CS101: Building a Search Engine

Unit 4: Building an Index

Table of Contents

Contents

Introduction

In unit 4 you are going to learn how to finish the code for your search engine and how to respond to a query when someone wants the given web pages that correspond to a given keyword. You will

also learn about how networks and the world wide web work to understand more about how you can build up your search index.

The main new computer science idea you will learn is how to build complex data structures. You will learn how to design a structure that you can use so that you can respond to queries without needing to rescan all the web pages every time you want to respond to a query. The structure you will build for this is called an index. The goal of an index is to map a keyword and where that keyword is found. For example, in the index of a book you can see a page number which serves as a map to where a term or concept can be found. The key ideas in index will allow us to find references to what we want. With a search engine the index gives you a way for a keyword to map to a list of web pages, which are the urls where those particular keywords appear. Once you have done the work of building an index, then the look-ups are really fast.

Quiz: Data Structures

Deciding on data structures is one of the most important parts of building software. As long as you pick the right data structure, the rest of the code will be a lot easier to write.

Which of these data structures would be a good way to represent the index for your search engine?

1. [<keyword1>, <url1, 1="">, <url1, 2="">, <keyword2>, ...]
2. [[<keyword1>, <url1, 1="">, <url1, 2="">], [<keyword2>, <url2, 1="">, ...]
3. <url1, 1="">, [<keyword1>, <keyword23>, ... , [<url2, 1="">, [<keyword2>, ...]]
4. <keyword1>, [<url1, 1="">, <url1, 2="">, [<keyword2>, [<url2, 1="">]], ...]

Answer

Add to Index

Quiz: Add to Index

Define a procedure, **add_to_index**, that takes three inputs:

- an index [<keyword>, [<url>, ...](/wiki/%3Ckeyword%3E%2C

%20%5B%3Curl%3E%2C%20%E2%80%A6), ...]
- a keyword string
- a url string

If the keyword is already in the index, add the url to the list of urls associated with that keyword.

If the keyword is not in the index, add an entry to to the index: **[keyword, [url]]**

For example:

```
index = []
add_to_index(index, 'udacity', 'http://udacity.com')
add_to_index(index, 'computing', 'http://acm.org')
```

This code starts with the empty list index. After the two lines of code the empty list will contain two lists beginning with the keywords, udacity and computing.

```
index = []
add_to_index(index, 'udacity', 'http://udacity.com')
add_to_index(index, 'computing', 'http://acm.org')
add_to_index(index, 'udacity', 'http://npr.org')
```

In this code, udacity is already in the index and you don't want to add a new entry to the index itself. Since udacity is already in the index, what you want to do is add the new url to the list already associated with that keyword.

IMAGE 1

Answer

Lookup Procedure

Quiz: Lookup

Define a procedure, lookup, that takes two inputs:

- An index: A list where each element of the list is a list containing a keyword and a list as its second element. The second list element is a list of urls where that keyword appears.
- The keyword to lookup

The output should be a list of the urls associated with the keyword. If the keyword is not in the index, the output should be an empty

list.

For example:

```
lookup('udacity') â†’ ['http://udacity.com', 'http://npr.org'
```

Answer

Building the Web Index

To build your web index, you want to find a way to separate all the words on a web page. It is possible to use the concepts you've already seen to build a procedure to do this, however, Python has a built-in operation that will make this much simpler.

Split. When you invoke the split operation on a string, the output is a list of the words in the string.

```
<''string''>.split()
[<''word''>, <''word''>, ... ]
```

For example:

```
quote = "In Washington, it's dog eat dog. In academia, it's e
print quote.split()
['In', 'Washington,', "it's", 'dog', 'eat', 'dog.', 'In', 'ac
    "it's", 'exactly', 'the', 'opposite.', '---', 'Robert', '
```

This operation does a pretty good job of separating out the words in the list so that they will be useful. However, in the case of **'Washington,**' which was followed by a comma in the quote, for the keyword you would not want to include the comma. While this isn't perfect, it is going to good enough for now.

Here is another example of how **split** works, using triple quotes (""""). Using the triple quotes you can define one string over several lines:

```
quote = """
     There's no buisiness like show business,
      but there are several businesses like accounting.
          (David Leterman)
          """
print quote.split()
["There's", 'no', 'buisiness', 'like', 'show', 'business,', '
'there', 'are', 'several', 'businesses', 'like', 'accounting.
'(David', 'Leterman)']
```

This still has similar problems to the first example, where the parentheses are included in the word **'(David**'.

Quiz: Add Page to Index

Define a procedure, **add_page_to_index**, that takes three inputs:

- index
- url (string)
- content (string)

It should update the index to include all of the word occurrences found in the page content by adding the url to the word's associated url list.

For example:

```
index = []

add_page_to_index(index, 'fake.test', "This is a test")
print index
['This', ['fake.test'], ['is', ['fake.test']], ['a', ['fake.tes
['test', ['fake.test']]]
```

Printing at **index[0]**:

```
index = []

add_page_to_index(index, 'fake.test', "This is a test")
print index[0]
['This', ['fake.test']]
```

Printing at **index[1]** :

```
index = []

add_page_to_index(index, 'fake.test', "This is a test")
print index[1]
['is', ['fake.test']]
```

Now, add a page called **real.test**:

```
index = []

add_page_to_index(index, 'fake.test', "This is a test")
add_page_to_index(index, 'real.test', "This is not a test")

print index
```

```
'This', ['fake.test', 'real.test'], ['is', ['fake.test',
'real.test']], ['a', ['fake.test']], ['test', ['fake.test',
'real.test']], ['not', ['real.test']]]
```

Have a look at the entries when you **index[1]**:

```
index = []

add_page_to_index(index, 'fake.test', "This is a test")
add_page_to_index(index, 'real.test', "This is not a test")

print index
print index[1]
'This', ['fake.test', 'real.test'], ['is', ['fake.test',
'real.test']], ['a', ['fake.test']], ['test', ['fake.test',
'real.test']], ['not', ['real.test']]]
['is', ['fake.test', 'real.test']]
```

Have a look at the entries when you **index[4]**:

```
index = []

add_page_to_index(index, 'fake.test', "This is a test")
add_page_to_index(index, 'real.test', "This is not a test")

print index

print index[1]

print index[4]
'This', ['fake.test', 'real.test'], ['is', ['fake.test',
'real.test']], ['a', ['fake.test']], ['test', ['fake.test',
'real.test']], ['not', ['real.test']]]

['is', ['fake.test', 'real.test']]

['not', ['real.test']]
```

You have already defined a procedure for responding to a query, so check out if it works on this index, searching for the keyword 'is':

```
index = []

add_page_to_index(index, 'fake.test', "This is a test")
add_page_to_index(index, 'real.test', "This is not a test")

print lookup(index, 'is') # you should expect to see that thi
['fake.test', 'real.test']
```

Now try searching for a keyword, 'udacity,' that is not in either of the urls:

```
index = []

add_page_to_index(index, 'fake.test', "This is a test")
add_page_to_index(index, 'real.test', "This is not a test")

print lookup(index, 'udacity') # you should expect to see an
[]
```

Well, that was successful! Try to define this procedure.

Answer

Finishing The Web Crawler

Returning to the code you wrote before for crawling the web, make some modifications to include the code you've just written.

First, a quick recap on how the **crawl_web** code below works before incorporating the indexing:

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

First, you defined two variables **tocrawl** and **crawled**. Starting with the **seed** page, **tocrawl** keeps track of the pages left to crawl, whereas **crawled** keeps track of the pages that have already been crawled. When there are still pages left to crawl, remove the last page from **tocrawl** using the **pop** method. If that page has not been crawled yet, get all the links from the page and add the new ones to **tocrawl**. Then, add the page that was just crawled to the list of crawled links. When there are no more pages to crawl, return the list of **crawled** pages.

Adapt the code so that you can use the information found on the pages crawled. The changed code is below. First, add the variable **index**, to keep track of the content on the pages along with their associated urls. Since you are really interested in the **index**, this is what we will return. It is possible to return both **crawled** and

index, but to keep it simple just **return index**. Next, add a variable, **content** to replace **get_page(page)**. This variable will be used twice, once in the code already there and once in the code to be filled in for the quiz. The procedure **get_page(page)** is expensive as it requires a web call, so we don't want to call it more often than is necessary. Using the variable **content** means that the call to the procedure **get_page(page)** only needs to be performed once and then the result is stored and can be used over and over without having to go through the expensive call again.

Quiz: Finishing the Web Crawler

Fill in the missing line using the variable content.

```
def crawl_web(seed):
    tocrawl = [seed]
    crawled = []
    index = []
    while tocrawl:
        page = tocrawl.pop()
        if page not in crawled:
            content = get_page(page)
            # FILL IN THE MISSING LINE BELOW HERE

            union(tocrawl,get_all_links(content))
            crawled.append(page)
    return index
```

Answer

Startup

You now have a functioning web crawler! From a seed you can find a set of pages; for each of these pages you can add the content from that page to an index, and return that index. Additionally, since you have already written the code you can do the look-up that will return the pages for that keyword.

But you're not completely done yet. In unit 5, you will see how to make a search engine faster and in unit 6 you will learn how to find the best page for a given query rather than returning all the pages.

Before then, you need to understand more about how the Internet works and what happens when you request a page on the world wide web.

The Internet

So far, you've been using the get_page function where you pass in a url and it passes out the content of the page. This is the python code that does that:

```
def get_page(url):
    try:
        import urllib
        return urllib.urlopen(url).read()
    except:
        return ""
```

The table below shows what each line of the code does.

CODE/EXPLANATION TABLE

Looking at the code for **get_page** doesn't help you understand what is going on when you request a page on the web because it is all hidden in the Python library, **urllib**. To understand more, read about how the Internet works.

Networks (Networks)

The Internet is a particular type of network. So what is a network?

A network is a group of entities that can communicate, even though they are not directly connected.

Meet Alice and Bob. They can communicate directly, so this is not a network by our definition:

IMAGE2

Next, Carol comes along. Alice, Bob and Carol can all communicate directly so this is still not a network.

IMAGE3

In this example, Alice and Carol can not communicate directly but they can still communicate via Bob, so this is a network.

IMAGE4

Many people would consider all three examples of a network, but you want to think about a more precise definition. A message that has to go through two hops, so that people who are not directly

connected can still communicate, is a network. It must have at least three people.

Quiz: Networks

This quiz tests your understanding of what a network is, and your knowledge of world history!

How old is the idea of a network?

1. 10 years old
2. 30 years old
3. 100 years old
4. 1000 years old
5. over 3000 years old

Answer

Smoke Signals

Humans have been using networks since at least as long as we can record. They might have just been networks where people just talked to each other and passed on message but even 3000 years ago, there were sophisticated networks using technology. In the Illiad, written about 800 BC, Homer describes a network from approximately 1200 B.C.

Thus, from some far-away beleaguered island, where all day long the men have fought a desperate battle from their city walls, the smoke goes up to heaven; but no sooner has the sun gone down than the light from the line of beacons blazes up and shoots into the sky to warn the neighbouring islanders and bring them to the rescue in their ships. Iliad, Homer written \~800 BC describing events \~1200BC

The Ancient Greeks had a line of beacons spread over the Greek Islands, which could be lit to announce an oncoming attack. When the soldiers at one of those beacons saw another beacon lit they would light their own beacon, spreading the message across the islands.

What does it take to make a network like this work? Suppose a message is to be sent from Rhodes to Sparta. There are many points along the way where smoke signals can be sent. First, the beacon is lit at Rhodes to be seen by the places near Rhodes. To reach Sparta, Naxos needs to resend the message by lighting its own beacon,

which can be seen at Melos, Cydonia and Athens. When Melos lights its beacon, Sparta receives the message.

Several different things are needed in order to make a network like this work:

- a way to encode and interpret messages - Greeks: "Agamennon is arriving" â†’ specific smoke signal
- a way to route messages, that is, a way to indicate which nodes should forward the message. Most likely the Ancient Greeks did this by lighting all the beacons, which is very wasteful. For the Greeks, a message such as 'the enemies are arriving' would be fine to spread throughout. This is called "flooding the network." To send a message between two points is a harder problem.
- rules for deciding who gets to use resources â€" who gets to send a message. Greeks: Generals have priority.

All these things are needed on the Internet too.

Latency

Latency is the time it takes a message to get from the source to the destination. NOTE: Latency can also mean round_trip latency(like the traceroute).

Latency is measured in seconds, or for a fast network today, in milliseconds. There are 1000 milliseconds in 1 second. (If you care about your ping in online games, it's latency that matters.)

Quiz: Latency

Zeus, the all powerful Greek god, wants to send a message from Rhodes to Sparta. He thinks that the latency of the smoke signalling system is too high, which means it takes too long for a message to be sent.

How could Zeus (remember, he's all powerful) reduce the latency between Rhodes and Sparta? (Choose one or more answers.)

1. Make the signalling nodes further apart, so it takes fewer hops.
2. Threaten the soldiers to make the signalling fires quicker.
3. Add colours to the smoke so there are more messages per signal.
4. Increase the speed of light.

Answer

Bandwidth

Bandwidth is the amount of information that can be transmitted per unit of time. Bandwidth does not consider the start up time (that's the latency) to get the first bit across, but the amount of information that can be got across once the first bit has arrived. In other words, it's the rate at which information is transferred.

Bandwidth is measured in units of [information/time], such as bits per second. On the Internet it is often measured in Mbps which is megabits per second.

Bits

A **bit** is the smallest unit of information.

IMAGE5

For example, if you ask, "Is the gold star in the green box?" you get one bit of information back. This bit of information takes you from having two choices to just one. If the answer is yes, you know the green box is the box you should open. If the answer is no, you know it's the blue box. (So 1 bit is the result to one yes or no question) A bit can be yes/no,true/false,1/0.

In computing, rather than talking about bits in terms of yes or no, you talk about them as 0 and 1, where 0 more readily maps to no and 1 to yes.

The diagram below shows how the question, "Is the gold star in the green box?" reduces the two choices to one with just one bit of information. If the answer is yes, that is 1, then the choice labelled with that arrow leads to the green box. If the answer is no, which is 0, the arrow leads to the blue box.

IMAGE6

Therefore, if there are two choices then knowing which to choose is one bit of information. For the Ancient Greeks' network, they could only transmit one bit of information by lighting the fire. If the fire was lit, then yes (1) or not lit (0). If that were the case, they would only have been able to send one message - the enemy is coming.

If all that can be sent is 0s and 1s, can anything more interesting be sent? Can just one thing be sent?

Yes! Anything you want can be sent!

Suppose that instead of two boxes, there are four boxes: green, blue, purple and red. There is still only one gold star.

IMAGE7

You could ask:

- Is the star in the green box?
- Is the star in the purple box?
- Is the star in the blue box?

As long as you know the gold star is in one of the boxes, you don't need to ask "Is the star in the red box?" because a negative answer to the other three questions already tells you it is. In order to be sure about where the gold star is, you need to ask at most, three questions, which takes in three bits. Is it possible to do better?

Quiz: Bits

How many bits are needed to find the gold star?

- 1
- 2
- 3

Answer

Before, these were the questions:

- Is the star in the green box?
- Is the star in the purple box?
- Is the star in the blue box?

Three questions might be needed, but two questions are enough if the answers to the questions give more information. The problem with asking "Is the star in the green box?" is that the answer is "No" three quarters of the time and "Yes" only one quarter. A guess of "No" would be right more than half the time. This means that the question does not provide a full bit's worth of information.

Instead, you would be better off asking a question that has an answer equally likely to be "Yes" as it is to be "No". So, an initial question could be "Is the gold star in either the green or purple box?" If the answer is yes, then the number of choices is cut in half and the gold star is in the green or purple box. If the answer is no,

again, the number of choices has halved and the gold star is in the blue or the purple box. After that question, only one more question is needed to determine exactly which box the star is in.

The diagram below illustrates that only a total of two questions is needed:

IMAGE8

This example shows that you can encode four things in bits. There are four colours, and two bits are needed to find an answer.

Any number of things can be encoded with more bits. In the diagram below, sixteen numbers are encoded using four bits.

IMAGE9

For every extra bit added, double the number of things can be encoded as there is one more yes/no question.

IMAGE10

This diagram tells you:

- For one bit, 2 numbers can be encoded.
- For two bits, it's double that number so, 2*2 = 22 =4 numbers.
- For three bits, it's double again, which means a total of 2*2*2 = 23=8numbers encoded.
- For four bits, as in the example above, 2*2*2*2=24=16 numbers are encoded.
- In general, the number of things which can be encoded in n bits is 2^n.

Once a number of things can be distinguished, information(data) can be sent. Anything that is discrete, such as strings or lists, can be converted into a number and that number can then be transmitted. Once bits can be sent, anything can be sent. The number of bits needed is a measure of the amount of information that can be sent.

Bits are directly related to bandwidth, which is a measure of the amount of information that can be sent. Bandwidth is measured in bits per second, each bit being a yes/no question â€" 0s and 1s. Those 0s and 1s could be encoding a string of text, like in a web page, or an image.

What Is Your Bandwidth

There are different ways to measure the bandwidth over your

Internet connection. One way is to use http://www.cnet.co.uk/broadband-speedtest/. This site is not necessarily that accurate. It will try sending messages to figure out your bandwidth and depends on your location. In the image below, the bandwidth shown is 6385 kbps (kilobits per second) which is 6.385 Mbps (megabits per second). If you do the test several times, you'll find the values vary.

IMAGE11

Bandwidth does vary since you are sharing the net with other people who may be doing different things. There are many reasons why bandwidth varies.

Quiz: What Is Your Bandwidth

This is more of a survey than a quiz.

What is your bandwidth?

- less than 1 kbps
- 1-10 kbps
- 10-100 kbps
- 100-1000 kbps 1000 kbps = 1Mbps
- 1-10 Mbps
- 10-20 Mbps
- 20-40 Mbps
- 40-60 Mbps
- 60-100 Mbps
- 100-200 Mbps

Answer

Traceroute

You can learn a lot about the Internet by measuring latency to different destinations. For the map of Greece, there were hops on the way from Rhodes to Sparta.

- Rhodes ->Naox -> Melos ->Sparta

Using an application called **traceroute**, the hops on the Internet can be seen between the machine it is run on and the destination.

Mac: On a mac, you can create a shell and run **traceroute** directly.

IMAGE12

Windows: 7 and Vista: Click on the start menu. Type cmd in the search bar and hit enter.

IMAGE13

Earlier versions of Windows: Click on the start menu and select run. In the open: box, type cmd and hit enter.

You'll get up a command prompt window.

IMAGE14

Note that the command in Windows is tracert and not traceroute.

Unix: Run **traceroute** from a terminal window.

Once you've typed **traceroute** (or tracert for Windows), pick where you want the destination to be. For the first example, **traceroute www.udacity.com**, you can see all the hops along the way. **Traceroute** is sending packets across the network looking at all the intermediate hops to figure out the route it takes to get a packet from the current location to www.udacity.com. A **packet** is a piece of information sent over the Internet. It contains a header and a message. The header contains information about, among other things, the packet source, destination and its "hop limit". Hop limit is the amount of hops the packet is allowed to go through. This is so that a packet can not get stuck in an infinite loop, jumping back and forth between the same places. The hop limit is used in **traceroute** to send packets out different distances on the route to the destination. The hop limit is only 1 byte long, which is 8 bits. From the previous discussion on bits, you saw that the amount of information which can be encoded by 8 bits is 2\^8=256, so the maximum number of hops is 255 (since it goes from 0 to 255 inclusive.) This puts a maximum distance a packet can travel on the Internet at 255 hops. Fortunately, everywhere on the Internet is connected by far fewer hops than that! IMAGE15

Here, you can see each hop. It took fifteen steps to get to www.udacity.com and the total time was about 39 milliseconds. You can see there are several different times (see circled figures) because the application is doing multiple tests and the time might vary from test to test. Although the **traceroute** is to www.udacity.com, from the first line of the **traceroute** you can see that the site is being served by a server at google.com. IMAGE16

The first hop shows the ip 192.168.3.1. This is an Internet address that refers to your router. You will always start there.

IMAGE17

After that you can see all the time amounts it takes to get to all the different sites. For example, to get to the Comcast site in Santa Clara took between 11.8 and 37.5 milliseconds.

IMAGE18

Each hop on the way took a few milliseconds, taking a about 40 milliseconds for 15 hops.

The next hop is somewhere further away at MIT, which is in Boston, Massachuettes, the other side of the country from Palo Alto, California. To do this, the command is **traceroute www.mit.edu**.

IMAGE19

Getting to udacity.com was not very far, geographically. To get to MIT, data travels through Santa Clara, to San Jose and then across the country to New York and finally to Boston. The locations are marked on this Google map: http://g.co/maps/u9ssk.<
>

You can see that Palo Alto, Santa Clara and San Jose are all on the West Coast of the USA and then there is a huge jump across the country to New York and Boston on the East Coast.

Sometimes from the host names you can guess the locations and sometimes you cannot. Unlike with the Ancient Greek fires, where the distance between hops was limited, the distance between hops on the Internet can be thousands of miles. There is probably a fiber optic cable between San Jose and New York and therefore no need for any routing directions between them â€" the data just goes straight through. The big time distance is from San Jose to New York, as highlighted on the screenshot above.

In total it took the packets about 100 ms to get across the country.

The three asterisks (*) at the bottom of the **traceroute** application's output indicate the packet is not actually getting to the final destination because there is no response from the web server at MIT. When this happens, you can try to repeat the **traceroute** with different time out options. If you want to terminate the traceroute before it's finished, you can use ctrl-c.

It takes longer to run **traceroute** than it does to send a message since the application is sending many messages and trying to find out all the points on the way.

About the furthest place from Palo Alto is Madagascar. Below is the output from **traceroute** www.mairie-antananarivo.mg.

The packet starts in Palo Alto, then goes from Santa Clara to Dallas, and then through many servers from the same company.

IMAGE20 & 21

Finally, it escapes from Texas at hop 18 and arrives in Washington DC (149.6.56.30), where it leaves the USA to head for France (217.16.0.2) at hop 19. It already takes nearly twice the time it took to get to MIT without having reached Madagascar yet. (Note: the information about the locations come from http://whatismyipaddress.com and may not be accurate.)

Quiz: Traveling Data

We observed that the latency between Palo Alto and Cambridge, MA is 100 milliseconds. The distance between Palo Alto, CA and Cambridge, MA is 4300 kilometers. At what fraction of the speed of light (\~300 000 km/s) did my data travel?

1. 1/7000
2. 1/70
3. 1/3
4. 1/700
5. 1/7
6. 0.8

Answer

Making a Network

Several different things are needed in order to make a network like this work.

1. A Way to encode and interpret messages

Greeks: "Agamennon is arriving" â† ' specific smoke signal Internet: message â† ' bits â† ' electrons/photons

Any message can be encoded in bits and then the bits can be encoded on the wire. How that encoding actually works is pretty complicated and it's not something that will be covered in this class. There are lots of different ways to do it.

1. A way to route messages

Greeks: directing smoke signals Internet: routers figure out next hops

For all the routers along the path, a message comes in and the router has to decide where to send it on. Maybe the router has a table saying where to send it next. Nirvada is on the way to Boston from California, so it could send the message on to Nirvada from California, but that wasn't what happened in the traceroute. The message was sent to San Jose where there is a big strong pipe straight across the country to New York and from there the message was sent on to Boston.

Routing is a challenging problem which won't be covered in more detail in this class.

1. Rules for deciding who gets to use resources

Greeks: generals have priority Internet: best effort service

Unlike with the Greeks, there are no real rules on the Internet for who gets resources. It's much more of a wild west where everywhere along the network gets to decide on its own what rules to apply. It's called a best effort service. If two messages need to be sent by a router at the same time, the router decides which one to send on. This means that your package might get dropped. There is no guarantee that a package will reach its destination on the Internet.

This class won't cover anything more on routing and rules, so you are encouraged to take a future networking class to learn more about these. However, we will look a little bit more about how messages work for the web.

Protocols

A **protocol** is a set of rules, that people agree to, which determines how two entities can talk to each other.

For the web, the protocol gives rules about how a client and a server talk to each other. The client is the web browser and the server is the web server.

IMAGE22

The protocol used on the web is called Hypertext Transfer Protocol which is abbreviated as HTTP. When you look in your browser, almost all the urls that you use start with http. That indicates that when the page is requested, the protocol to be used to talk to the

server is this Hypertext Transfer Protocol. It's a very simple protocol, and only has two main messages. One of those messages is **GET**. The client can send a message to the server which says **GET** followed by the name of the object you want to get, **GET <object>**. That's all the client does. The python code for **get_page**:

```
def get_page(url):
 try:
     import urllib
     return urllib.urlopen(url).read()
 except:
     return ""
```

calls a library function **urllib.urlopen(url)** which actually does this.

After the client sends the message, the server will receive it. The server runs some code on it, finds the file that was requested, perhaps runs some more code, and then sends back a response with the contents of the requested **<object>**. That's the whole protocol.

IMAGE23

If you click on a link in your browser, your web browser works out which url you are requesting and sends a **GET** message to the correct web server specified by that url. When it gets a response, it processes and then renders it.

If you'd like to learn more about what the server does, take the Web App course, CS253 Web Application Engineering and if you're interesting in how a web browser works, take the course CS262:Programming Languages.

Conclusion

Hopefully you understand at a high level what a web browser does when requesting data over the Internet. There is nothing magic about it. The process is all about sending messages across the Internet and receiving responses which are text. That text is processed by a browser, or even by the web crawler you've programmed.

The search engine so far works but it isn't fast or smart. In unit 5, you'll look at how to make the search engine scale and respond to queries faster. In unit 6, you'll learn how to find the best response to a query, that is, to respond with the best page rather than all the pages.