CS101: Building a Search Engine

Unit 3: Data: Learning to Crawl

Table of Contents

Contents

Introduction

The main new topic for Unit 3 is structured data. By the end of this unit, you will have finished building a simple web crawler.

The closest thing you have seen to structured data so far is the string type introduced in Unit 1, and used in many of the procedures in Unit 2. A string is considered a kind of structured data because you can break it down into its characters and you can operate on sub-sequences of a string. This unit introduces lists, a more powerful and general type of structured data. Compared to a string where all of the elements must be characters, in a list the elements can be anything you want such as characters, strings, numbers or even other lists!

The table below summarizes the similarities and differences between strings and lists.

String vs. List Table

Quiz: Stooges

Define a variable, stooges, whose value is a list of the names of the Three Stooges: "Moe", "Larry", and "Curly."

Answer

Quiz: Days in a Month

Given the variable:

```
days_in_month = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
```

define a procedure, **how_many_day**s, that takes as input a number representing a month, and outputs the number of days in that month.

```
how_many_days(1) â†’ 31
how_many_days(9) â†’ 30
```

Answer

Nested Lists

So far, all of the elements in our lists have been of the same type: strings, numbers, etc. However, there are no restrictions on the types of elements in a list. Elements of a list can be any type you want, you can also mix and match different types of elements in a list.

For example:

```
mixed_up = ['apple', 3, 'oranges', 27,
                     [1, 2, ['alpha', 'beta']]]
```

or a more useful example:

```
beatles = [['John', 1940],
               ['Paul', 1942],
               ['George', 1943],
               ['Ringo', 1940]]
```

This list provides information about the names of the Beatles band members, as well as when they were born. Try putting this into your interpreter. When you are typing your code into the interpreter and you want to separate data onto two lines, do so after a comma to make it clear to the interpreter that this is still one list.

```
beatles = [['John', 1940], ['Paul', 1942],
               ['George', 1943], ['Ringo', 1940]]

print beatles
'John', 1940], ['Paul', 1942], ['George', 1943], ['Ringo', 19

print beatles[3]
['Ringo', 1940]
```

You can also use indexing again on the list that results to obtain an inner element:

```
    print beatles[3][0]
    Ringo
```

Quiz: Countries

Given the variable countries defined as:

```
countries = [['China', 'Beijing', 1350],
                 ['India', 'Delhi', 1220],
                 ['Romania', 'Bucharest', 21]
                 ['United States', 'Washington', 307]]
```

Each list contains the name of a country, its capital, and its approximate population in millions.

Write code to print out the capital of India.

Answer

Quiz: Relative Size

What multiple of Romania's population is the population of China? To solve this, you need to divide the population of China by the population of Romania.

Answer

Mutation

Mutation means changing the value of an object. Lists support mutation. This is the second main difference between strings and lists.

It might have seemed like we could change the value of strings:

```
s = 'Hello'
s = 'Yello'
```

However, this expression changes the value the variable s refers to, but does not change the value of the string **Hello**. As another example, consider string concatenation:

```
s = s + 'w'
```

This operation may look like it is changing the value of the string, but that's not what happens. It is not modifying the value of any string, but instead is creating a new string, **Yellow**, and assigning the variable **s** to refer to that new string.

Lists can be mutated, thus changing the value of an existing list.

Here is a list:

```
p = ['H', 'e', 'l', 'l', 'o']
```

Mutate a list by modifying the value of its elements:

```
p[0] = 'Y'
```

This expression replaces the value in position 0 of p with the string 'Y'. After the assignment, the value of p has changed:

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

p[4] = '!'
print p
['Y', 'e', 'l', 'l', '!']
```

Quiz: Different Stooges

Previously, you defined:

```
stooges = ['Moe', 'Larry', 'Curly']
```

In some Stooges films, though, Curly was replaced by Shemp. Write one line of code that changes the value of stooges to be:

```
['Moe', 'Larry', 'Shemp']
```

but does not create a new list object.

Answer

Aliasing

Now that you know how a mutation modifies an existing list object, you will really be able to see how this is differs from strings when you introduce a new variable.

```
p = ['H', 'e', 'l', 'l', 'o']
p[0] = 'Y'
q = p
```

After this assignment, p and q refer to the same list: **['Y', 'e', 'l', 'l', 'o']**.

Suppose we use an assignment statement to modify one of the elements of q:

```
q[4] = '!'
```

This also changes the value of p:

```
print p
['Y', 'e', 'l', 'l', '!']
```

After the **q = p** assignment, the names p and q refer to the same list, so anything we do that mutates that list changes that value both variables refer to.

It is called aliasing when there are two names that refer to the same object. **Aliasing** is very useful, but also can be very confusing since one mutation can impact many variables. If something happens that changes the state of the object, it affects the state of the object for all names that refer to that object.

Strings are Immutable. Note that we cannot mutate strings, since they are immutable objects. Try mutating a string in the interpreter: #!highlight python s = 'Hello' s[0] = 'Y' 'str' object does not support item assignment } **Mutable and Immutable Objects**. The key difference between mutable and immutable objects, is that once an object is mutable, you have to worry about other variables that might refer to the same object. You can change the value of that object and it affects not just variable you think you changed, but other variables that refer to the same object as well.

Here is another example:

```
p = ['J', 'a', 'm', 'e', 's']
q = p
p[2] = 'n'
```

Both **p** and **q** now refer to the same list:

```
['J', 'a', 'n', 'e', 's']
```

What happens if you assign **p** a new value, as in:

```
p = [0, 0, 7]
```

In this case, the value of p will change, but the value of q will remain the same. The assignment changes the value the name p refers to, which is different from mutating the object that p refers to.

Quiz: Aliasing

What is the value of agent[2] after running the following code:

```
spy = [0, 0, 7]
agent = spy
spy[2] = agent[2] + 1
```

Answer

Quiz: Replace Spy

NOTE: This explains what happens when a mutable object like a list is passed in a procedure.

Define a procedure, **replace_spy**, that takes as its input a list of three numbers and increases the value of the third element of the list to be one more than its previous value. Here is an example of the behavior that you want:

```
spy = [0,0,7]
replace_spy(spy)
print spy
[0, 0, 8]
```

Answer to Q-7

List Operations

There are many built-in operations on lists. Here are a few of the most useful ones here.

Append. The append method adds a new element to the end of a list. The append method mutates the list that it is invoked on, it does not create a new list. The syntax for the append method is:

```
    <''list''>.append(<''element''>)
```

For example, assume you want to end up with four stooges in your list, instead of just three:

```
stooges = ['Moe', 'Larry', 'Curly']
stooges.append('Shemp')
['Moe', 'Larry', 'Curly', 'Shemp']
```

Concatenation. The + operator can be used with lists and is very similar to how it is used to concatenate strings. It produces a new list, it does not mutate either of the input lists.

```
<''list''> + <''list''> â†’ <''list''>
```

For example,

```
[0, 1] + [2, 3] â†’ [0, 1, 2, 3]
```

Length. The **len** operator can be used to find out the length of an object. The **len** operator works for many things other than lists, it works for any object that is a collection of things including strings. The output from len is the number of elements in its input.

```
                 len(<''list''>) â†’ <''number''>
```

For example, **len([0,1]) â†’ 2**. Note that len only counts the outer elements:

```
len(['a', ['b', ['c', 'd']]]) â†’ 2
```

since the input list contains two elements: **'a**' and [**'b**', [**'c**', **'d**']].

When you invoke len on a string, the output is the number of elements in the string.

```
len("Udacity") â†’ 7
```

Quiz: Len Quiz

What is the value of **len(p)** after running the following code:

```
p = [1, 2]
p.append(3)
p = p + [4, 5]
len(p) â†’ ?
```

Answer to Q-8

Quiz: Append Quiz

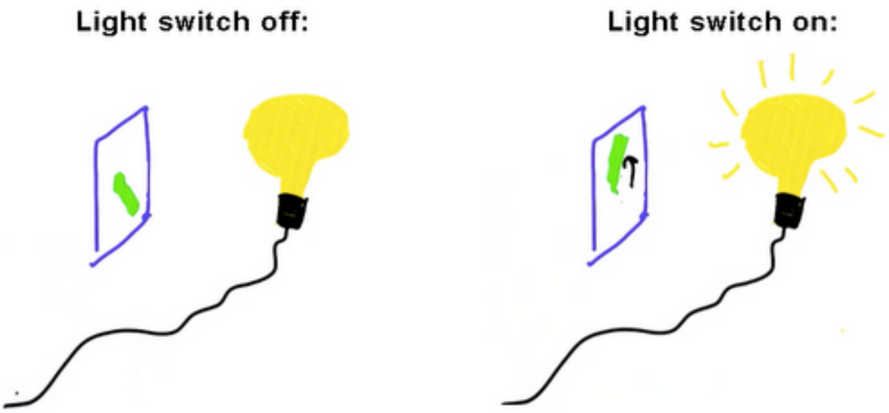
What is the value of len(p) after running:

```
p = [1, 2]
q = [3, 4]
p.append(q)
len(p) â†’ ?
```

Answer to Q-9

How Computers Store Data

In order to store data you need two things: (1) something that preserves state, and (2) a way to read its state. Our storage mechanism needs to have more than one state, but two states is enough. We can think about this like a light switch, which is connected to a light bulb through some power source. When you turn the light switch on, the light bulb turns on:

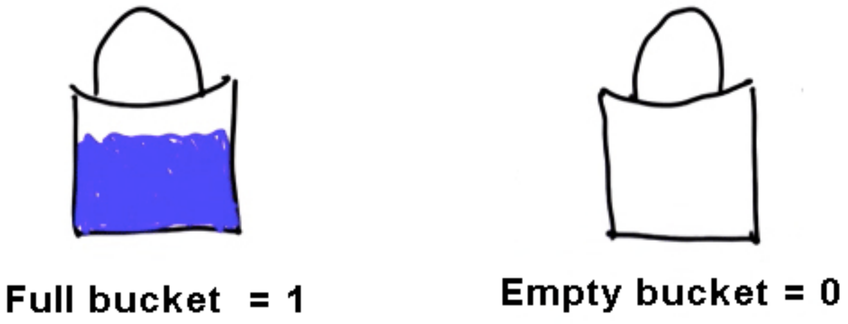


Flipping the switch changes the state of the light bulb. The light bulb has two different states: it can be on or off. This is what we need to store one bit of data. A **bit** is the fundamental unit of *information*. One bit is enough to decide between two options (for example, on or off for the light bulb). If you had enough light bulbs you could store many bits, which would be enough to be able to store any amount of digital data.

In addition to something that can change state, to read memory you also need something that can sense the state. In terms of a light bulb, that could be an eye or a light sensor, which could see if the light bulb was on or off. This is very similar to the way computers store data, but computers use much less energy and much less space than a light bulb to store one bit.

The fastest memory in your computer works like a switch. Data that is stored directly in the processor's memory, which is called the **register**, is stored like a switch, which makes it very fast to change and read its state. However, a register is like a light bulb in that when you turn the power off, you lose the state. This means that all the data stored in registers is lost when the computer is turned off.

Another way that computers store data is similar to a bucket. We could represent a one by a full bucket and represent a zero with an empty bucket. To check the state of the bucket, we could weigh the bucket or look at it to figure out whether it is full or empty.



The difference between buckets and light bulbs is that buckets leak a little, and water evaporates from the bucket. If you want to store data using a bucket, it will not last forever. Eventually, when all the water evaporates you will be unable to tell the difference between a zero and a one. Computers solve this problem using the digital abstraction. There are infinitely many different amount of water that could be in the bucket, but they are all mapped to either a 0 or a 1 value. This means it is okay if some water evaporates, as long as it does not drop below the threshold for representing a **1**.

In computers, the buckets are holding electrons instead of water, and we call them **capacitors**. The memory in your computer that works this way is called **DRAM**.

DRAM

Below is a two gigabytes (GB) of DRAM taken out of a computer.

image

A gigabyte means approximately a billion bytes. One byte is 8 bits.

A gigabyte is actually 2^30 bytes. This is very close to one billion, but in computing it is usually more convenient to use powers of two.

In Python, the exponentiation operator is denoted with two asterisks:

```
<''base''> ** <''power''> â†’ <''base''><''power''>
```

For example,

```
print 2 ** 10
```

```
1024
```

One kilobyte is 1024 bytes.

```
print 2 ** 20 # one megabyte
1048576

print 2 ** 30 # one gigabyte
1073741824

print 2 ** 40 # one terabyte
1099511627776
```

Kilobytes, megabytes, gigabytes, and terabytes are the main units we use to talk about computer memory.

Now, back to the DRAM, which is two gigabytes of memory. Since one gigabyte is 2^30 bytes, we can compute the total number of bits by multiplying that by 2 (since there are two gigabytes) and 8 (the number of bits in a byte):

- 2\^30*2*8[?]17 billion light switches
- 1 byte = 8 bits
- 1 bit light switch (two states)

Thus, the DRAM shown is like having 17 billion buckets, each one can store one bit.

There are many different types of memory inside your computer, for example, registers, that were mentioned earlier as the fastest memory that is built right into the processor. What distinguishes different types of memory is the time it takes to retrieve a value (this is called **latency**), the cost per bit, and how long it retains its state without power.

For DRAM, the latency is about 12 nanoseconds (recall that there are one billion nanoseconds in a second). The cost of the 2 GB DRAM show is about 10 USD (approximately 7 euros).

Memory Hierarchy

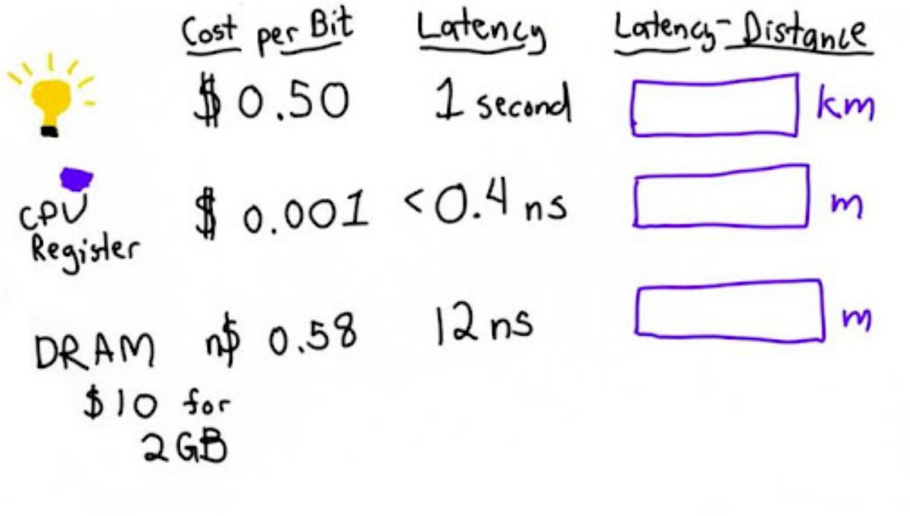
To get a better understanding of the different types of memory in the computer, let's compare them in terms of **Cost per Bit** and **Latency**. Since times in nanoseconds are hard to relate to, we will convert the latencies into how far light travels in the time it takes to retrieve a stored bit.

Since the costs per bit get pretty low, we introduce a new money unit: one nanodollar (n$) is one billionth of a US dollar, and truly not worth the paper on which it is printed!

image

Quiz: Memory Hierarchy

Fill in the Latency-Distance for the light bulb, CPU register and DRAM using the information provided. Keep in mind that you will be finding the answers in different units. As a reminder the speed of light is about 300,000 km/sec.



Hard Drives

Another type of memory in your computer is a **hard drive**. Inside the hard drive there are several disks that spin. The disks store data magnetically, and there is a read-head that can read data from the disks as well as write new data from the disk. Compared to DRAM, this is a very slow way of storing data since it involves spinning a physical disk and moving a read head, but it can store data for far less cost than DRAM. The other advantage of storing data on a hard drive is that it persists. The data is not lost even when the power is turned off.

Where our DRAM was two gigabytes, this hard drive can store one terabyte, which is 500 times as much memory. A terabyte is close to a trillion bytes:

- 8*2\^40 bits [?] 8.8trillion bits

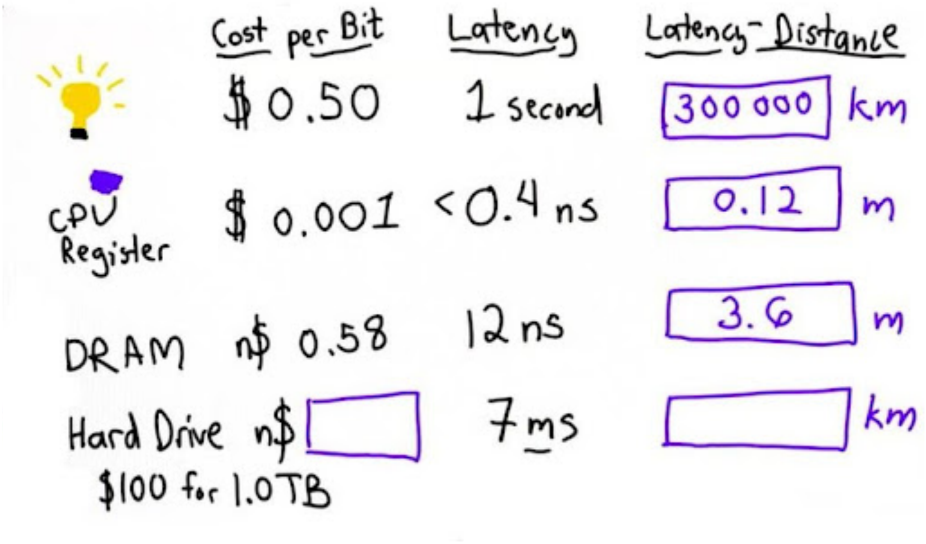
This is enough memory to store about 100 hours of high quality video.

The latency for a hard drive is much higher than it is for DRAM. This is because the hard drive is moving physical things. It operates using disks, so you have to wait for the disk to spin and reach the read-head. Also, if the disk isn't in the right place then you might have to wait for the read-head to move. The average latency for a hard drive is about seven milliseconds (1 millisecond = 1/1000 of a second = 1 million nanoseconds).

The cost of this 1.0 terrabyte hard drive is 100 USD (75 Euros), so the cost per bit is much lower than it is for DRAM memory.

Quiz: Hard Drives

Add the hard drive data to your unit table. Include how many nanodollars it costs per bit and its latency-distance.



	Cost per Bit	Latency	Latency-Distance
💡	$0.50	1 second	300 000 km
CPU Register	$ 0.001	<0.4 ns	0.12 m
DRAM	n$ 0.58	12 ns	3.6 m
Hard Drive $100 for 1.0TB	n$ □	7 ms	□ km

Loops on Lists

Since lists are collections of things, it is very useful to be able to go through a list and do something with every element.

In Unit 2, we introduced the **while** loop:

```
while <'''TestExpression'''>:
        <'''Block'''>
```

If the test expression is True, the <**Block**> is executed. At the end of the block, execution continues by re-evaluating the test expression, and continuing to execute the block as long as the test expression evaluates to true.

Quiz: Loops on Lists

Define a procedure called **print_all_elements** that takes as input a list p and prints out every element of that list. Here is a start:

NOTE: This procedure is not passing the variable through reference. It is just passing the value (list). If the list is changed, then the list that the original variable before procedure is called is also affected. It DOES NOT change what the variable is referring to

```
def print_all_elements(p):
    i = 0
    while __________:
          print p[i]
          i = i + 1
```

Answer

For Loops

In Python there is a more convenient way to loop through the elements of a list: the for loop. The syntax looks like this:

```
for <''name''> in <''list''>:
        <''block''>
```

The loop goes through each element of the list in turn, assigning that element to the <**name**> and evaluating the <**block**>. Using the for loop, we can use less code than we needed using the while loop to define the procedure **print_all_elements**:

```
def print_all_elements(p):
    for e in p:
        print e
```

Let's walk-through what happens when you apply this **for** loop to a list:

```
mylist = [1, 2, 3]
print_all_elements(mylist)
```

When you pass in mylist to **print_all_elements** the variable **p** will refer to the list that contains the three elements, **1**, **2** and **3**. When the loop is executed, the variable **e** is assigned to the first element in the list, and the body of the loop will print the first element. So, for the first iteration the value of **e** will be **1**. The block is executed, printing out **1**. Since there are more elements in the list, execution continues, assigning **2** to the **e**. Again, the block is executed, but this time it prints out **2**. Execution continues for the third iteration, which prints out **3**. There are no more elements in the list, so the for loop is complete and execution continues with the next statement (in this case, there is no following statement, so execution finishes).

Quiz: Sum List

Define a procedure, **sum_list**, that takes as its input a list of numbers, and produces as its output the sum of all the elements in the input list.

For example,

```
sum_list([2, 7, 4]) â†’ 13
```

Answer

Quiz: Measure Udacity

Define a procedure, **measure_udacity**, that takes as its input a list of strings, and outputs a number that is a count of the number of elements in the input list that start with an uppercase letter 'U'.

For example,

```
measure_udacity(['Dave', 'Sebastian', 'Katy'])
0

measure_udacity(['Umika', 'Umberto'])
2
```

Answer

Quiz: Find Element

Define a procedure, **find_element**, that takes as its input a list and a value of any type, and outputs the index of the first element in the input list that matches the value. If there is no matching element, output -1.

Examples:

```
find_element([1, 2, 3], 3) â†’ 2
find_element(['alpha', 'beta'], 'gamma') â†’ -1
```

Answer

Index

There are many other ways to define find_element. A built-in list operation that we have not yet introduced that makes it easier to write find_element is the index method:

```
<''list''>.index(<''value''>) â†’ <''position''> or error
```

The index method is invoked on a list by passing in a value, and the output is the first position where that value sits in the list. If the list that does not contain any occurrences of the value you pass in, index produces an error (this is different from the find method for strings which we used in Unit 1, that returns a -1 when the target string is not found).

Examples:

```
p = [0, 1, 2]
print p.index(2)
2

p = [0, 1, 2, 2, 2]
print p.index(2)
2
```

Even though there are many 2s in the list, the output is the first position where 2 occurs. #!highlight python p = [0, 1, 2] print p.index(3) ValueError: list.index(x): x not in list } Since the requested behavior of find_element is to output -1 when the input element is not found, we cannot use index directly to implement find_element since index produces an error when the element is not found. Instead, we can use another list operation, in, to first test if the element is anywhere in the list. We have already seen in used in the for loop, however outside of a for loop header it means something different:

```
<''value''> in <''list''> â†’ <''Boolean''>
```

The output is **True** if the list contains an element matching value, and **False** if it does not.

Examples:

```
p = [0, 1, 2]
print 3 in p
False
print 1 in p
True
```

Similarly, you can use **not in**, which has the opposite meaning of **in**:

```
<''value''> not in <''list''>
```

If the value is not in the list the result of <*value*> not in <*list*> is **True**, and if the <*value*> is in the <*list*> than the result is **False**.

These two expressions are equivalent:

```
<''value''> not in <''list''>  not <''value''> in <''list''>
```

Quiz: Index

Define **find_element**, this time using index.

Answer

Quiz: Union

Define a procedure, **union**, that takes as inputs two lists. It should modify the first input list to be the set union of the two lists.

Examples:

```
a = [1, 2, 3]
b = [2, 4, 6]
union(a, b)
print a
[1, 2, 3, 4, 6]
print b
[2, 4, 6]
```

Answer to Q-17

Pop

The pop operation mutates a list by removing its last element. It returns the value of the element that was removed.

```
<''list''>.pop() â†’ element
```

Example:

```
a = [1, 2, 3]
b = a # both a and b refer to the same list
x = a.pop() # value of x is 3, and a and b now refer to the l
```

Quiz: Pop Quiz

Assume p refers to a list with at least two elements. Which of these code fragments does not change the final value p.

1.

2. x = p.pop() y = p.pop() p.append(x) p.append(y)

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

4. x = p.pop() y = p.pop() p.append(y) p.append(x)

Answer

Collecting Links

Now we are ready to finish our web crawler!

You need to start by finding all the links on the seed page, but instead of just printing them like you did in Unit 2, you need to store them in a list so you can use them to keep going. Go through all the links in that list to continue our crawl, and keep going as long as there are more pages to crawl.

The first step to define a procedure **get_all_links** that takes as input a string that represents the text on a web page and produces as output a list containing all the URLs that are targets of link tags on that page.

Get All Links

Here is a recap of the code from Unit 2:

```
def print_all_links(page):
    while True:
        url, endpos = get_next_target(page)
        if url:
            print url
            page = page[endpos:]
        else:
            break
```

We defined a procedure, **get_next_target**, that would take a page, search for the first link on that page, return that as the value of **url** and also return the position at the end of the quote is so we know where to continue.

Then, we defined the procedure, **print_all_links**, that keeps going as long as there are more links on the page. It will repeatedly find the next target, print it out, and advance the page past the end position.

What we want to do to change this is instead of printing out the URL each time we find one, we want to collect the URLs so we may use them to keep crawling and find new pages. To do this, we will create a list of all of the links we find. We change the **print_all_links** procedure into **get_all_links** so that we can use the output, which will be a list of links, which will correspond to the links we were originally printing out.

Links

As an example of how this should work, there is a test page at https://www.udacity.com/cs101x/index.html. It contains three link tags that point to pages about crawling, walking, and flying (you can check them out for yourself by clicking on links on the test page in your web browser).

Here is how **get_all_links** should behave:

- links = get_all_links(get_page('http://www.udacity.com/cs101x/index.html') print links ['http://www.udacity.com/cs101x/crawling.html', 'http://www.udacity.com/cs101x/walking.html', 'http://www.udacity.com/cs101x/flying.html']

Because the result is a list, we can use it to continue crawling pages. Think on your own how to define **get_all_links**, but if you get stuck, use the following quizzes to step through the changes we need to make.

== Quiz: Starting **get_all_links**

What should the initial value of **links** be? Remember, your goal for **get_all_links** is to return a list of all the links found on a page. You will use the links variable to refer to a list that contains all the links we have found.

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

Answer

Quiz: Finishing Links

For this last quiz on **get_all_links**, figure out how to get the output:

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

Answer

Finishing the Web Crawler

At this point we are ready to finish the web crawler. The web crawler is meant to be able to find links on a seed page, make them into a list and then follow those links to new pages where there may be more links, which you want your web crawler to follow.

In order to do this the web crawler needs to keep track of all the pages. Use the variable **tocrawl** as a list of pages left to crawl. Use the variable crawled to store the list of pages **crawled**.

Crawling Process - First Attempt

Here is a description of the crawling process. We call this **pseudocode** since it is more precise than English and structured sort of like Python code, but is not actual Python code. As we develop more complex algorithms, it is useful to describe them in pseudocode before attempting to write the Python code to implement them. (In this case, it is also done to give you an opportunity to write the Python code yourself!)

- start with **tocrawl** = [seed] **crawled = []** while there are more pages **tocrawl**:

    - pick a page from **tocrawl** add that page to **crawled** add all the link targets on this page to **tocrawl**

  return **crawled**

Quiz: Crawling Process- First Attempt

#!wiki What would happen if we follow this process on the test site, starting with the seed page http://www.udacity.com/cs101x/index.html ?

1. It will return a list of all the urls reachable from the seed page.

2. It will return a list of some of the urls reachable from the seed page.
3. It will never return.

Answer

The next several quizzes implement your web crawling procedure, **crawl_web**, that takes as input a seed page url, and outputs a list of all the urls that can be reached by following links starting from the seed page.

Quiz: Crawl Web

To start the **crawl_web** procedure, provide the initial values of tocrawl and crawl:

```
def crawl_web(seed):
    tocrawl = ______ - initialize this variable
    crawl = ________ - initialize this variable
```

Answer

Quiz: Crawl the Web Loop

The next step is to write a loop to do the crawling, where you keep going as long as there are pages to crawl. To do this, you will use a **while** loop, with **tocrawl** as your test condition. You could use **len(tocraw) == 0** to test if the list is empty. There is an easier way to write this using just **tocrawl**. An empty list (a list with no elements) is interpreted as false, and every non-empty list is interpreted as true.

Inside the loop, we need to choose a page to crawl. For this quiz, your goal is to figure out a good way to do this. There are many ways to do this, but using things we have learned in this unit you can do it using one line of code that both initializes **page** to the next page we want to crawl and removes that page from the **tocrawl** list.

```
def crawl_web(seed):
    tocrawl = [seed]
    crawled = []
    while tocrawl:
        page = ____________
```

Answer

Quiz: Crawl If

The next step is to manage the problem of cycles in the links. We do not want to crawl pages that we've already crawled, so what we need is someway of testing whether the page was crawled.

To make a decision like this, we use if. We need a test condition for if that will only do the stuff we do to crawl a page if it has not been crawled before.

```
def crawl_web(seed):
    tocrawl = [seed]
    crawled = []
    while tocrawl:
          page = tocrawl.pop()
          if __________________
```

Answer

Quiz: Finishing Crawl Web

Now you're ready to finish writing our crawler. Write two lines of code to update the value of tocrawl to reflect all of the new links found on page and update the value of crawled to keep track of the pages that have been crawled.

```
def crawl_web(seed):
    tocrawl = [seed]
    crawled = []
    while tocrawl:
          page = tocrawl.pop()
          if page not in crawled:
             ______________________
             ______________________
    return crawled
```

Answer

Conclusion

Anna Patterson has been working on search engines for more than a decade. She led the development of the world's largest web index for the http://recall.archive.org/ project. For more of her insights on building search engines, see her article, Why Writing Your Own Search Engine Is Hard (ACM Queue, April 2004). She is currently a Director of Research at Google, working on Android.