How to build a search engine

4th September 2012

Contents

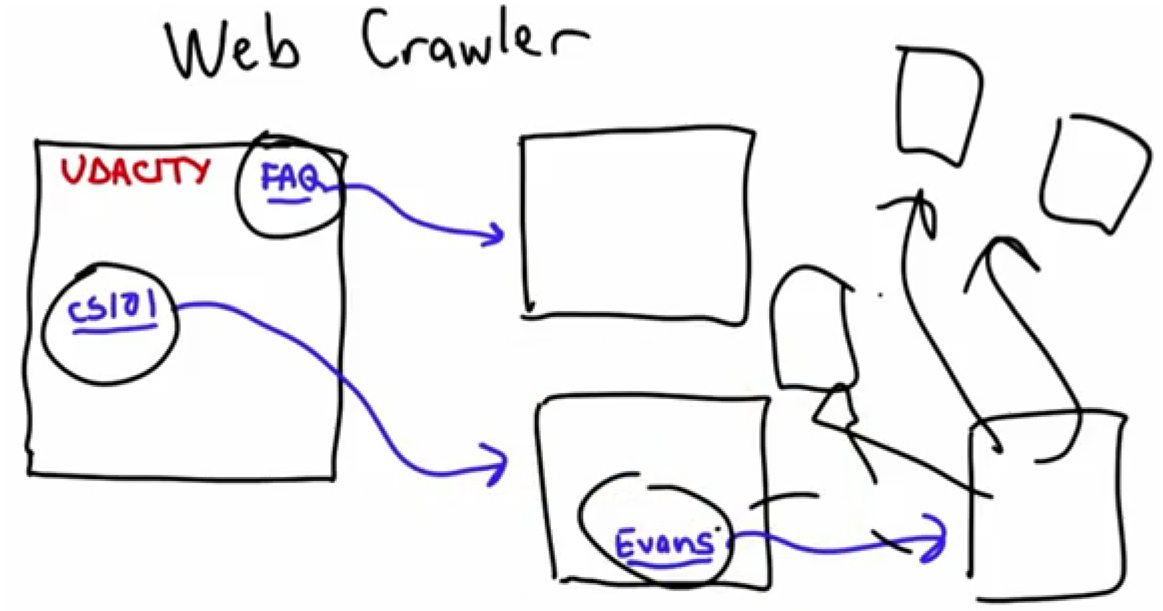
1 How to get started: your first program

1.1 Introducing the Web Crawler

Definition 1.1. A **web crawler** is a program that collects content from the web.

Algorithm 1.2 (The Process that a web crawler follows:)**.**

1. *Start* from *one preselected* page (the "**seed**" page).
2. *Extract all links* on page, and *follow* each *link* to find *new pages.*
3. *Extract all links* from *all new pages*, and *follow links* to find new pages.
4. *Repeat Step 3* as long as there are new pages to find, or until it is stopped.



1.2 Programming

Programming is the core of computer science.

Definition 1.3. A **computer** is a machine that can execute a program.
With the right program, a computer can do any mechanical computation you can imagine.

Definition 1.4. A **program** describes a very precise sequence of steps.

Since the computer is just a machine, the program must give the steps in a way that can be *executed mechanically.*

Definition 1.5. A **programming language** is a language designed for producing computer programs.

A good programming language makes it easy for humans to read and write programs that can be executed by a computer.

Remark. We will be using **Python**.

1.3 Getting Started with Python Programming

1.4 Would You Rather (use natural language)

Disadvantages

- **Ambiguous** - people interpret phrases in different ways. Computer need exact statements in order to function accurately.
- **Verbose** (Using more words than needed) - to say something with the level of precision needed for a computer to be able to follow it mechanically would require an awful lot of writing.

1.5 Grammar

- Compared to a natural language, programming languages adhere to a strict grammatical structure.
- In English, even if a phrase is written or spoken incorrectly, it can still be understood with the help of context or other cues (body language, tone).
- When programming language grammar is not followed the interpreter will return a SyntaxError message.

1.6 Backus-Naur Form (*1950s by John Backus IBM)*

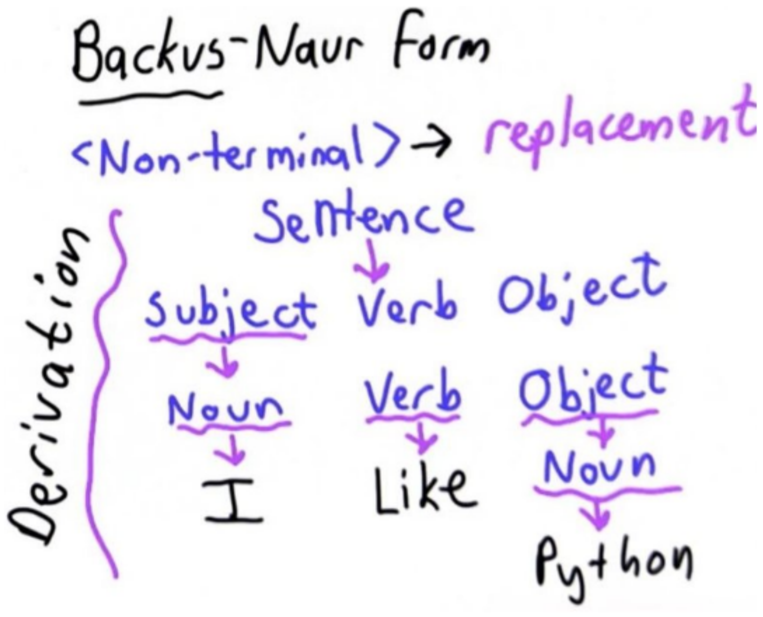
- The purpose of Backus-Naur Form is to describe a programming language in a simple and concise manner.

Structure:

<Non-Terminal> → replacement

- The **replacement** can be any sequence of zero or more non-terminals or terminals.
- **Terminals** never appear on the left side of a rule. Once you get to a terminal there is nothing else you can replace it with.

Example (Sentence following Replacement Rules).



Sentence → *Subject Verb Object*
→ *Noun Verb Object*
→ I Verb *Object*
→ I Like *Object*
→ I Like *Noun*
→ I Like *Python*

Remark. Replacement grammar is *important* as we describe an infinitely large language with a small **set** of precise rules.

1.7 Python Expressions

Definition 1.6. An **expression** is something that has a value.

A rule of the Python grammar for making expressions:
Expression → *Expression Operator Expression*

- The *Expression* non-terminal (LHS) can be replaced by an *Expression*, followed by an *Operator*, followed by another *Expression*.
  - E.g. 1 + 1 is an *Expression Operator Expression*.
- This **rule** that has *Expression* on both left and right side looks circular. However, we also have other rules for *Expression* that do not include *Expression* on the RHS.
- This is an example of a **recursive definition**.

1.7.1 Recursive definitions:

To make a good recursive definition you need at least two rules:

1. A rule that **defines something** in **terms** of **itself**.

   (a) *Expression* $\rightarrow$ *Expression Operator Expression*

2. A rule that **defines that thing** in **terms** of **something** else that we already know.

   (a) *Expression* $\rightarrow$ *Number*

Remark. These allow us to **define infinitely many things** using a **few simple rules** - a very powerful idea in computer science.

Example (Python grammar rules for arithmetic expressions)**.**

Expression $\rightarrow$ *Expression Operator Expression*
Expression $\rightarrow$ *Number*
Operator $\rightarrow$ +
Operator $\rightarrow$ *
Number $\rightarrow$ 0, 1, ...

Example (Derivation using this grammar)**.**

Expression $\rightarrow$ *Expression Operator Expression*
$\rightarrow$ *Expression* + *Expression*
$\rightarrow$ *Expression* + *Number*
$\rightarrow$ *Expression* + 1
$\rightarrow$ *Expression Operator Expression* + 1
$\rightarrow$ *Number Operator Expression* + 1
$\rightarrow$ 2 *Operator Expression* + 1
$\rightarrow$ 2 * *Expression* + 1
$\rightarrow$ 2 * *Expression Operator Expression* + 1
$\rightarrow$ 2 * *Number Operator Expression* + 1
$\rightarrow$ 2 * 3 *Operator Expression* + 1
$\rightarrow$ 2 * 3 * *Expression* + 1
$\rightarrow$ 2 * 3 * *Number* + 1
$\rightarrow$ 2 * 3 * 3 + 1

Example (Another Rule)**.**

Expression $\rightarrow$ *(Expression)*

Example (1,3,5 are valid Python expressions, 2,4 do not follow the rules)**.**

1. 3

2. ((3)

3. (1 * (2 * (3 * 4)))

4. $+ 3\ 3$

5. $(((7)))$

Definition 1.7. A **processor** (or CPU)**is** the part of the computer that carries out the steps specified in a computer program.

1.8 Admiral Grace Hopper (1906-1992)

- Grace Hopper was a pioneer in computing who was known for walking around with nano-sticks.
- Nano-sticks are pieces of wire that are the length light travels in a nanosecond, about 30 cm.
- Hopper wrote one of the first programming languages, COBOL, which was for a long time the world's most widely used programming language. Hopper built the first compiler.

Definition 1.8. A **compiler** is a program that takes as input a program in a programming language easy for humans to write and outputs a program in another language that is easier for computers to execute.

Remark. The difference between a compiler and an interpreter like Python is that a compiler does all the work at once and runs a new program, whereas, the interpreter converts the source code one step at a time as the program runs.

- When Grace Hopper started building the first compiler, most people did not believe it was possible for a computer program to produce other computer programs: "Nobody believed that I had a running compiler and nobody would touch it. They told me computers could only do arithmetic."

1.9 Variables

Definition 1.9. A **variable** is a name refers to a value.
To introduce a new variable, we use an assignment statement:
Name = *Expression*

Notation 1.10. We use the **hash mark** (#) to introduce a comment.

1.10 Variables Can Vary

1.11 Strings

Definition 1.11. A **string** is a sequence of characters surrounded by quotes, either *single* or *double.*

```
'I am a string!'
```

Remark. The only requirement is that the string must **start** and **end** with the **same kind of quote**.

```
"I prefer double quotes!"
```

Remark. This allows you to **include quotes inside of quotes** as a character in the string.

```
"I'm happy I started with a double quote!"
```

1.12 August Ada King, Countess of Lovelace, 1815-1852

Arguably the world's first computer programmer, she worked with Charles Baggage to design the first programmable machine.

1.12.1 Strings and Numbers

Notation 1.12. With string, + means **concatenation**.
$<string> + <string> \rightarrow$ outputs the concatenation of the two strings

Notation 1.13. You can multiply strings and integers! E.g. to return 12 exclamation points!

```
print '!' * 12
!!!!!!!!!!!!
```

1.13 Indexing Strings

Definition 1.14. Selecting sub-sequences from a string, it is called **indexing**.

Notation 1.15. Square brackets [] specifies which part of the string you want to select.

```
<string>[<expression>]
```

Remark. Like JavaScript, we start from the $0th$ position.

```
'udacity'[0]
'u'
'udacity'[1 + 1]
'a'
name = 'Dave'
name[0]
'D'
print name[-1]
'e'
```

1.14 Selecting Sub-Sequences

You can select a **sub-sequence** of a string by designating a *starting* position and *end* position.

```
<string>[<expression>] → a one-character string
<string>[<start expression>:<stop expression>] →
```

Example. `word = 'assume'`

```
print word[4:6]
'me'
print word[4:]
'me'
print word[:2]
```

```
'as'
print word[:]
'assume'
```

1.15 Finding Strings in Strings

The **find** method operates on strings, with the output of find being the *position* of the *first letter* of the string where the specified sub-string is found.

```
<search string>.find(<target string>)
```

Remark. If the target string is not found anywhere in the search string, then the output will be -1.

```
pythagoras = 'There is geometry in the humming of the strings, there is music in
the spacing of the spheres.  '
print pythagoras.find('string')
40
print pythagoras[40:]
'strings, there is music in the spacing of the spheres.'
print pythagoras.find('T')
0
print pythagoras.find('sphere')
86
print pythagoras[86:]
spheres.
print pythagoras.find('algebra')
-1
```

1.16 Find with Numbers (from the position of the number)

```
<search string>.find(<target string>, <number>)
```

The number input is the **position** in the **search** string where find will **start** looking for the target string.

Example. `danton = "De l'audace, encore de l'audace, toujours de l'audace."`

```
print danton.find('audace')
5
print danton.find('audace', 0)
5
print danton.find('audace', 5)
5
print danton.find('audace', 6)
25
print danton.find('audace', 25}
25
print danton.find('audace', 48)
-1
```

1.17 Extracting Links

Definition 1.16. A **web page** is really just a *long string of characters.* ('View Page Source')

Note. Your browser renders the web page in a way that looks more attractive than just the string of characters.

Remark. For our **web crawler**, the important thing is to find the links to other web pages in the page.
We find those links by looking for the anchor tags that match this structure:
`<a href="<url>">`

- To build our crawler, for each web page we want to find **all** the link target URLs on the page.
- We want to **keep track of them** and **follow them to find more content** on the web.

2 How to repeat. Finding all the links on a page

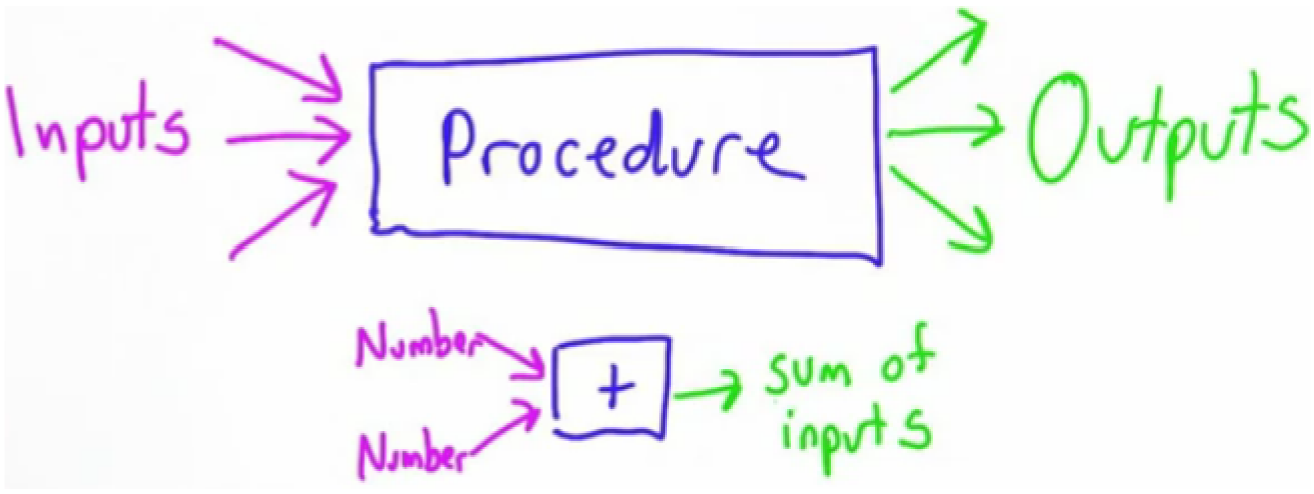
2.1 Introduction

In order to extract all of the links (the aim of this Unit), you need to know these 2 key concepts:

Definition 2.1. Procedures - a method to package code so it can be reused with different inputs. (Appear similar to functions, see below why we call them procedures.)

Definition 2.2. Control - a method to have the computer execute different instructions depending on the data.

Remark. "Different inputs, different behaviours."



2.2 Motivating Procedures

In order to extract all links, copying and pasting the code over and over may work:

```
page = page[end_quote:]
start_link = page.find('<a href=')
start_quote = page.find('"', start_link)
end_quote = page.find('"', start_quote + 1)
url = page[start_quote + 1:end_quote]
print url

page = page[end_quote:]
start_link = page.find('<a href=')
start_quote = page.find('"', start_link)
end_quote = page.find('"', start_quote + 1)
url = page[start_quote + 1:end_quote]
print url
```

Here the code will print out the next 2 links on the web page.

Disadvantages

- **Tedious** - the reason for computers is to avoid having to do tedious, mechanical work!
- **Inefficient** - some pages only have a **few links** while other pages have **more links than the number of repetitions**.

2.3 Introducing Procedures

Example. The + operator is a (built-in) procedure where the inputs are two numbers and the output is the sum of those two numbers.

2.3.1 Python Grammar for Procedures

```
def <name>(<parameters>):
    <block>
```

Notation 2.3. `def` - "define"

`<name>` is the name of a procedure.

`<parameters>` are the **inputs** to the procedure: a list of (**none**, "an empty set of closed parentheses: ()", or) **more names separated by commas**: `<name>`, `<name>`,...

Remark. When naming parameter, it is more beneficial to use descriptive names that remind you of what they mean. (Here we have replaced `page` with `s` because it can be used for any string.)

```
def get_next_target(s):
    start_link = s.find('<a href=')
    start_quote = s.find('"', start_link)
    end_quote = s.find('"', start_quote + 1)
    url = s[start_quote + 1:end_quote]
```

Note. A : (**colon**) ends the definition header (as above).

Notation 2.4. `<block>` - the body of the procedure is a `<block>`, which is the code that implements the procedure.

The block is **indented** inside the definition.

Remark. **Proper indentation tells** the **interpreter** when it has **reached** the **end** of the procedure definition - IMPORTANT IN PYTHON.

2.4 Return Statement

To finish the procedure, we need to **produce** the **outputs** - `return`.

Notation 2.5. `return <expression>, <expression>, ...`

Definition 2.6. A **return** statement can have *any number* of *expressions*, of which are the *outputs* of the procedure.

Remark. No expressions at all means no output. This can be useful as you can harness a procedure for their **side-effects**.

Definition 2.7. Side-effects are *visible*, such as the *printing* done by a *print* statement, but are NOT the outputs of the procedure.

2.5 Using Procedures

Notation 2.8. `<procedure>(<input>,<input>, ...)`

Remark. There are other forms (recall the `.find()` one; we learn about these later).

Example (rest_of_string procedure)**.**

```
def rest_of_string(s):
    return s[1:]
```

To use, we pass in one input, corresponding to the parameter s:

```
print rest_of_string('audacity')
    udacity
```

Summary. Many people call procedures in Python "functions." Refer to them as procedures because they are quite different from mathematical functions. The main differences are:

- A *mathematical* function always produces the *same output given the same inputs* - NOT necessarily the case for a Python procedure, which can produce different outputs for the same inputs depending on other state (we will see examples in Unit 3).
- A mathematical function *only* maps inputs to outputs. A Python procedure can also *produce side-effects,* like printing.
- A mathematical function is a *pure abstraction* that has no associated cost. The cost of executing a Python procedure depends on *how* it is *implemented.* (We will discuss how computer scientists think about the cost of procedures in Unit 5.)

2.6 Equality Comparisons

2.6.1 Comparison Operators

Python provides several operators for making comparisons:

$<$ less than
$>$ greater than
$\leq$ less than or equal to
$==$ equal to
$! =$ not equal to

Notation 2.9. `<number> <operator> <number>`

The output of a comparison is a **Boolean**: **True** or **False**.

2.7 If Statements

Notation 2.10. An `if` statement provides a way to control what code executes based on the result of a test expression.

```
if <''TestExpression''>:
      <''block''>
```

Remark. Similar to procedures, the end of the `if` statement block is determined by the *indentation*.
(Recall the `block` only runs `if True`.)

2.8 Is Friend: Else Expressions

The trivial consequence.

```
def bigger(a, b):
    if a > b:
       return a
     else:
        return b
```

2.9 Or Expressions

Notation 2.11. An `or` expression gives the logical or (disjunction) of two operands.

`<Expression> or <Expression>`

Remark. If the first expression evaluates to True, the value is True and the second expression is *not evaluated.*

If the value of the first expression evaluates to False, the value is the value of the second expression.

```
print True or this_is_an_error
True
```

2.10 Alan Turing

What a legend!

He showed that

> "Every possible computer program can be made from:
>
> 1) Variables
>
> 2) Arithmetic
>
> 3) Procedures
>
> 4) `if` statements

2.11 While Loops

Same form as MATLAB

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

2.12 Factorial

The number of ways to order blocks.

2.13 Break

Notation 2.12. `Break` gives us a way to break out of a loop, even if the test condition is `True`.

```
while <''TestExpression''>:
      <''Code''>
      if <''BreakTest''>:
         break # stop executing the while loop
      <''More Code''>
<''After While''>
```

The break statement jumps out of the loop to <*After While*>.

2.14 Multiple Assignment

Definition 2.13. Assigning multiple values on the left side of an assignment statement is called **multiple assignment**.

Notation 2.14. `<name1>, <name2>, ... = <expression1>, <expression2>, ...` with name1 matching expression1, name2 matching expression2 ...

Example. get_next_target(page):

```
def get_next_target(page):
    start_link = page.find('<a href=')
    start_quote = page.find('"', start_link)
    end_quote = page.find('"', start_quote + 1)
    url = page[start_quote + 1:end_quote]
    return url, end_quote
```

In order to run the code successfully, we are required to state two output arguments (like MATLAB).

```
url, endpos = get_next_target(page)
```

2.15 No Links

We need to modify to ensure valid execution for all scenarios. (This has taken the form of the quiz. A standard `if` statement does the trick.)

2.16 Print All Links

We modify our `get_next_target` code to print all links until a value of `url = None` is returned.

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

    print get_page('http://xkcd.com/353')
```

3 How to manage data. Crawling the web

3.1 Data: Learning to Crawl

3.2 Introduction

The aim for this unit is to learn about **structured data**. We initially look at **lists**, which are more powerful than **strings** (which are a type of structured data as you can break it down into its characters and operate on sub-sequences) as they can:

- the elements can be **anything** compared to a string (where all elements must be characters).

3.3 Nested Lists

Lists within lists.

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

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

3.4 Mutation (a contrast to MATLAB = value semantics)

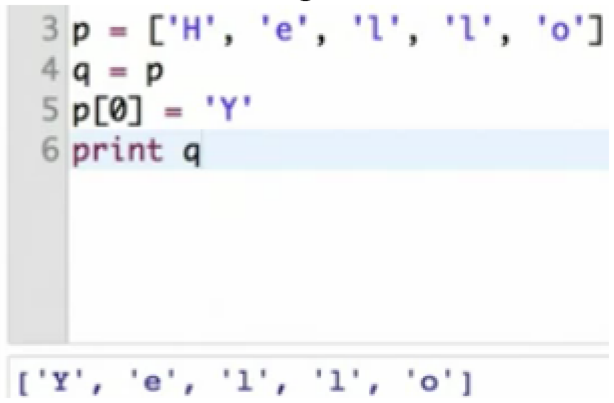
Definition 3.1. Mutation means *changing* the value of an object.

Remark. Lists support mutation. This is the second main difference between strings and lists.

Note. Strings *appear* to be able to mutate, however it really is just creating a new string.

3.5 Aliasing

Definition 3.2. Aliasing - when there are two (or more) names that refer to the *same object.*



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

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

Remark. “Modifying one modifies both - DIFFERENT TO MATLAB!”

3.6 List Operations

Notation 3.3. **Append** - a method that adds a new element to the *end of a list* - mutation not creation.

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

Notation 3.4. **Concatenation** - The + operator can be used with lists -mutation not creation.

```
<''list''> + <''list''> → <''list''>
```

Notation 3.5. **Length** - The `len` operator can be used to find out the length of an object.

```
len(<''list''>) → <''number''>
```

Remark. Not just for lists; includes strings and others.

3.7 How Computers Store Data

In order to store data you need two things:

1. Something that preserves state, and

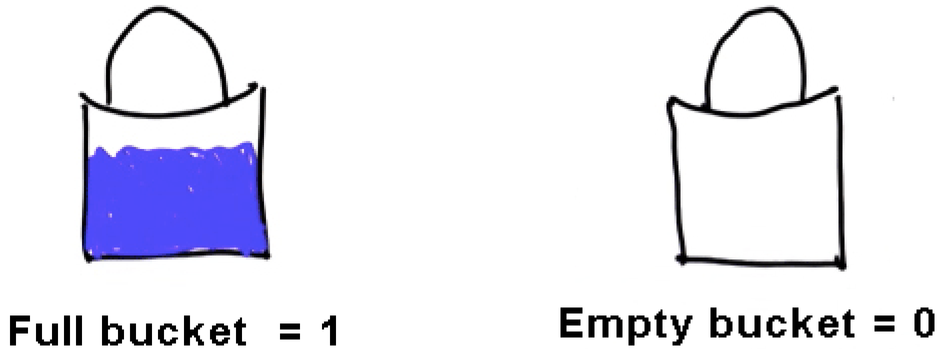
2. A way to read its state.

Definition 3.6. A **bit** is the fundamental unit of information.

Definition 3.7. Data that is stored directly in the *processor*, which is called the **register**, is stored like a switch (as a bit), which makes it very fast to change and read its state.

Remark. Like a light bulb, 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.

3.8 DRAM



Remark. 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.

Definition 3.8. 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**.

One byte is **8 bits**.
A **gigabyte** is 2^{30} bytes.

Notation 3.9. **Exponentiation** is denoted by ** (two asterisks)

```
print 2 ** 10
1024
```

Thus, the DRAM shown is like having $2^{30} \times 2^{8} = 17$ billion buckets, each one can store one bit.

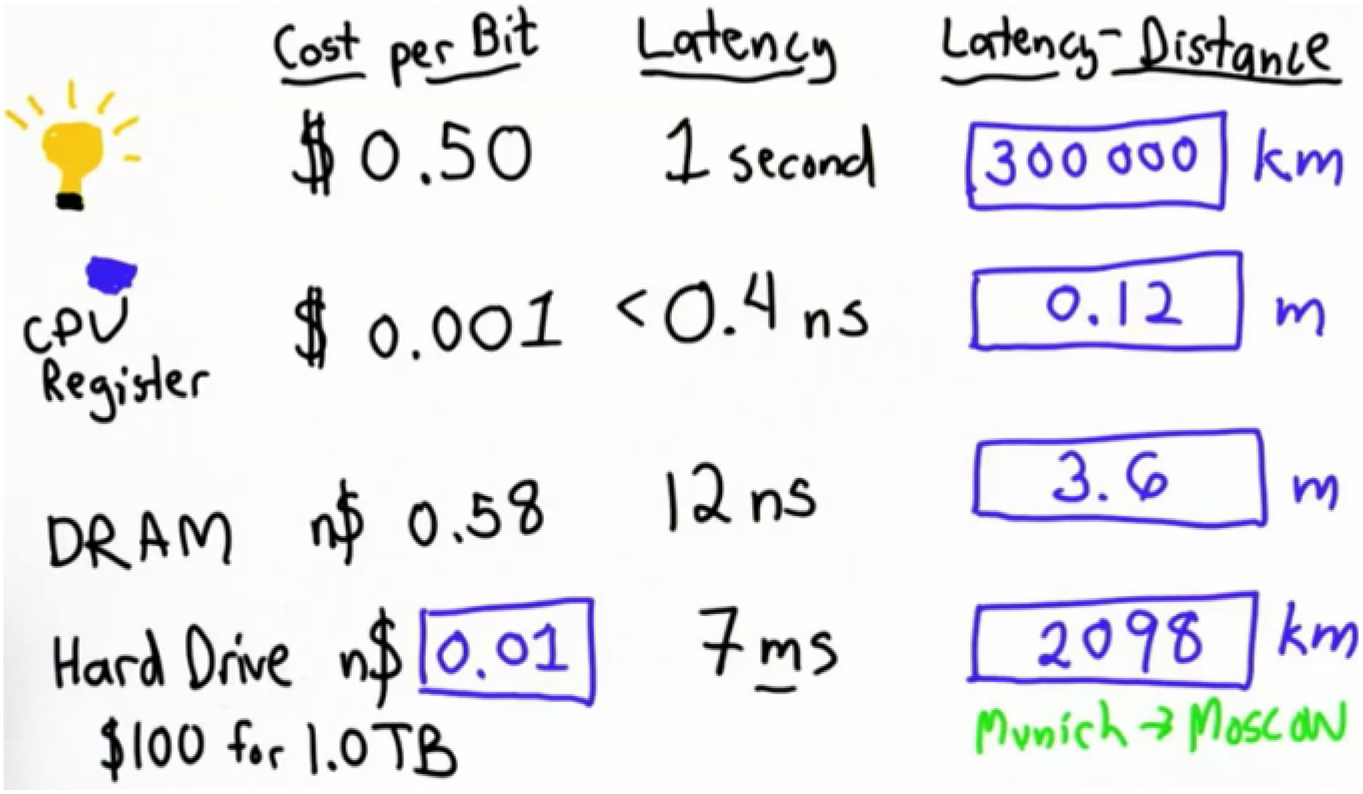
3.8.1 Memory

There are many different types of memory.
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**.

Example. For DRAM, the *latency* is about *12 nanoseconds.*

3.9 Memory Hierarchy



	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 $100 for 1.0TB	n$ 0.01	7 ms	2098 km Munich → Moscow

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

3.10 Hard Drives

- Much **slower** at storing data (in comparison to DRAM), since it involves spinning a physical disk and moving a read head - you have to wait for the disk to spin and reach the read-head (and move the read-head if necessary - isn't in the right place).
- Much **more** data can be stored at much lower cost (1TB in comparison to 2GB).
- Data **persists**. The data is not lost even when the power is *turned off*.

3.11 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.

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

3.12 For Loops

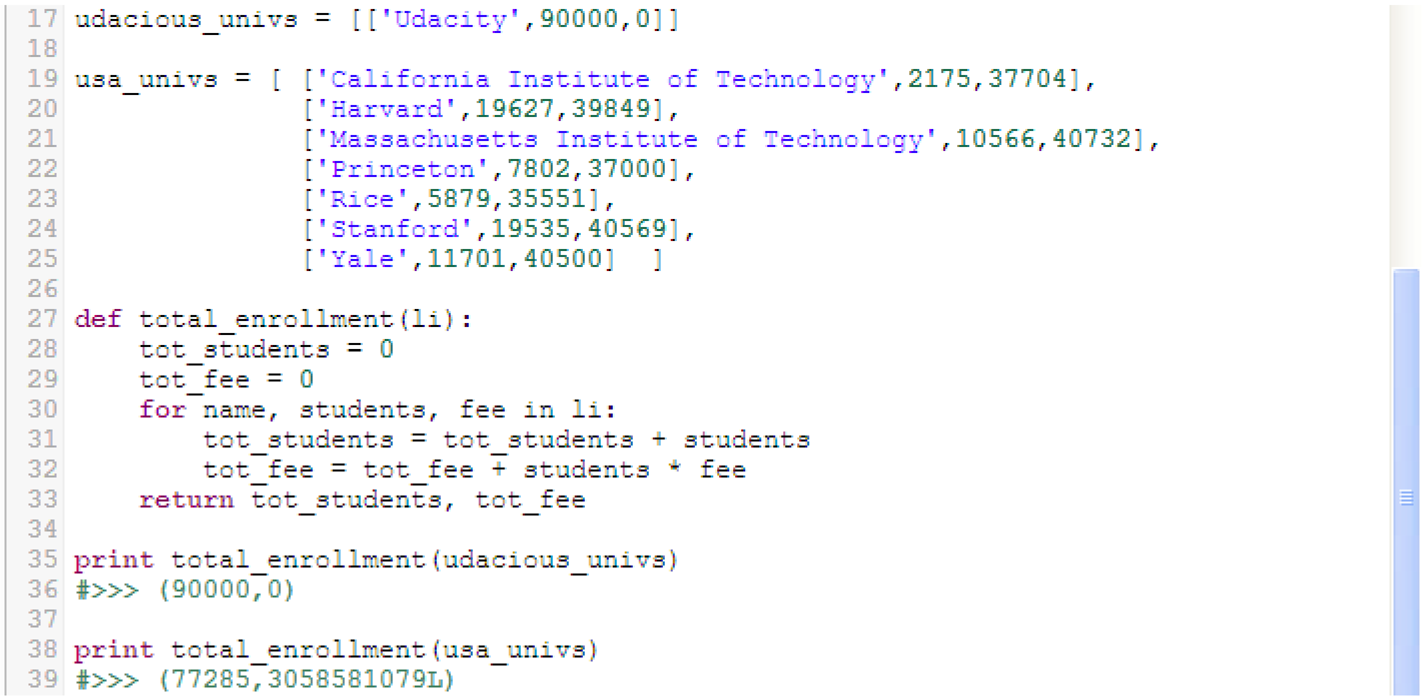
A more convenient way to loop through the elements of a **list** (or **string**): the `for` loop.

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

Example. `e` is assigned to each list value.

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

If we know what elements are in the list of a list, we can assign each a name:



```
udacious_univs = [['Udacity',90000,0]]

usa_univs = [ ['California Institute of Technology',2175,37704],
              ['Harvard',19627,39849],
              ['Massachusetts Institute of Technology',10566,40732],
              ['Princeton',7802,37000],
              ['Rice',5879,35551],
              ['Stanford',19535,40569],
              ['Yale',11701,40500]  ]

def total_enrollment(li):
    tot_students = 0
    tot_fee = 0
    for name, students, fee in li:
        tot_students = tot_students + students
        tot_fee = tot_fee + students * fee
    return tot_students, tot_fee

print total_enrollment(udacious_univs)
#>>> (90000,0)

print total_enrollment(usa_univs)
#>>> (77285,3058581079L)
```

Notation 3.10. `range(<start>, <stop>)`.

Remark. This can be useful for creating for loops. (For example when creating a hash table:

```
table = []
for unused in range(0, nbuckets)
      table.append([])
return table
```

3.13 Index

Notation 3.11. The **`index`** method is used on a `list` by passing in a value, and the output is the **first position** where that value sits in the list; otherwise an error is produce if the list does not contain any occurrences of the value

```
<''list''>.index(<''value''>) → <''position''> or error
```

Remark. 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.

Notation 3.12. `in` (outside a `for` loop) - tests if the element is anywhere in the list.

```
<''value''> in <''list''> → <''Boolean''>
```

Example. `in` (outside `for` loop)

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

Notation 3.13. We can also use `not in`:

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

3.14 Pop

Notation 3.14. `pop` mutates a list by **removing** its **last element** and returns the value of the element that was removed.

```
<''list''>.pop() → element
```

Example. `pop`

```
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 list [1, 2]
```

3.15 Collecting Links

- 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.

3.16 Get All Links

- 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 **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.

3.17 Links

We work with the test page at http://www.udacity.com/cs101x/index.html.

Here is how `get_all_links` should behave:

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

3.18 Finishing the Web Crawler

- The web crawler has 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**.

3.19 Crawling Process

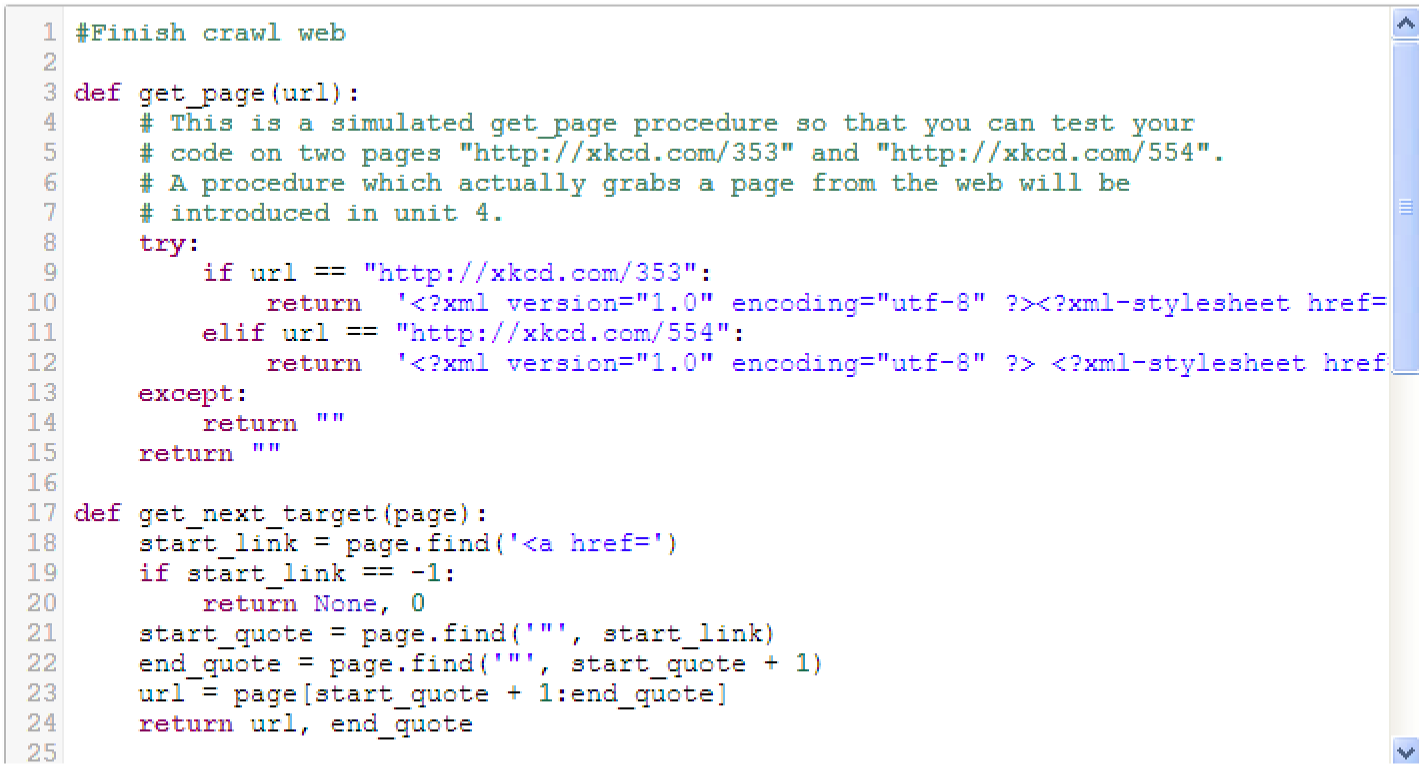
3.19.1 First Attempt

Here is the pseudo-code for the Crawling Process:

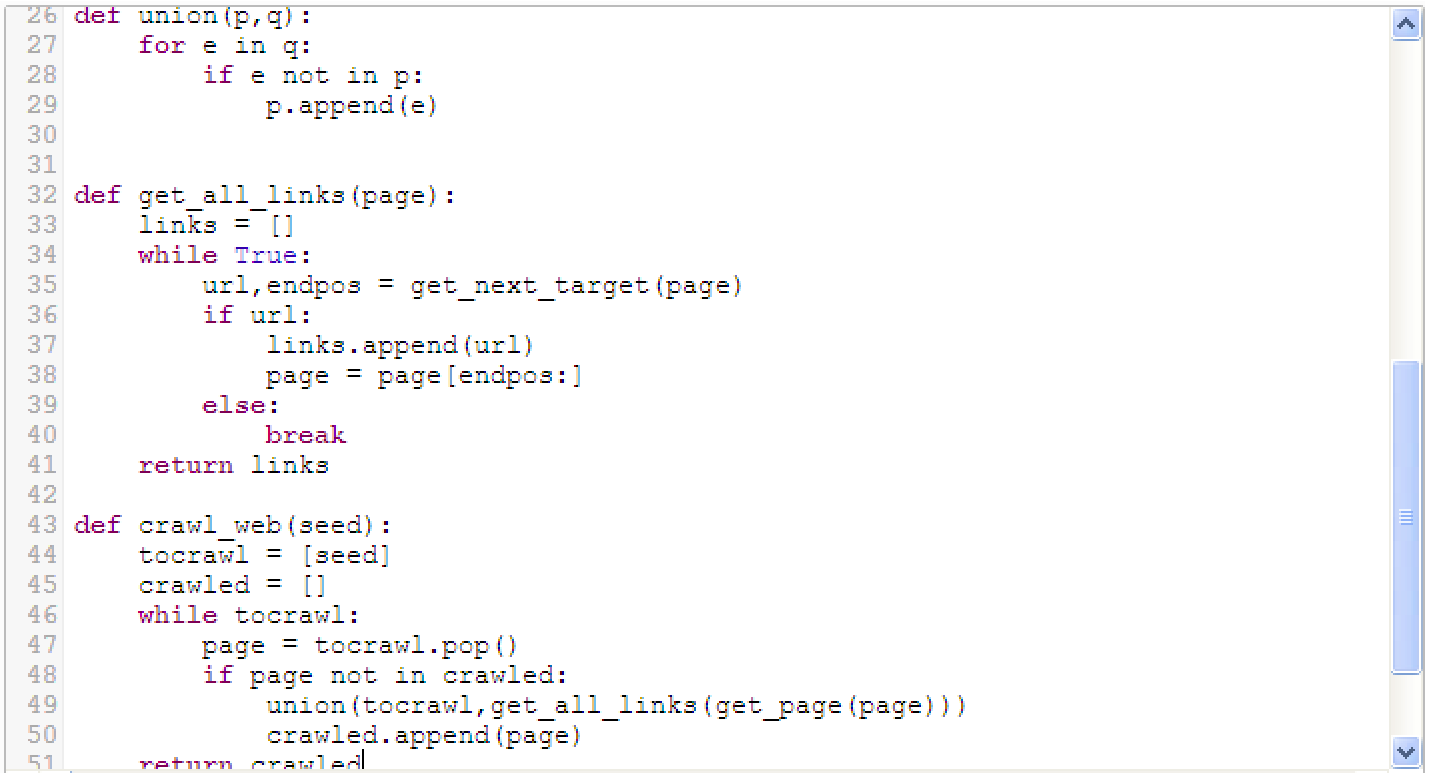
```
start with tocrawl = [seed] crawled = [] while there are more pages tocrawl:
    pick a page from tocrawl add that page to crawled add all the link targets on this page to tocrawl
return crawled
```

Remark. This can never return for some sites that have "circular links" (for example, our test site).

3.20 Conclusion



```python
#Finish crawl web

def get_page(url):
    # This is a simulated get_page procedure so that you can test your
    # code on two pages "http://xkcd.com/353" and "http://xkcd.com/554".
    # A procedure which actually grabs a page from the web will be
    # introduced in unit 4.
    try:
        if url == "http://xkcd.com/353":
            return  '<?xml version="1.0" encoding="utf-8" ?><?xml-stylesheet href=
        elif url == "http://xkcd.com/554":
            return  '<?xml version="1.0" encoding="utf-8" ?> <?xml-stylesheet href
    except:
        return ""
    return ""

def get_next_target(page):
    start_link = page.find('<a href=')
    if start_link == -1:
        return None, 0
    start_quote = page.find('"', start_link)
    end_quote = page.find('"', start_quote + 1)
    url = page[start_quote + 1:end_quote]
    return url, end_quote

```



```
def union(p,q):
    for e in q:
        if e not in p:
            p.append(e)


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

4 Building an Index (just like a book)

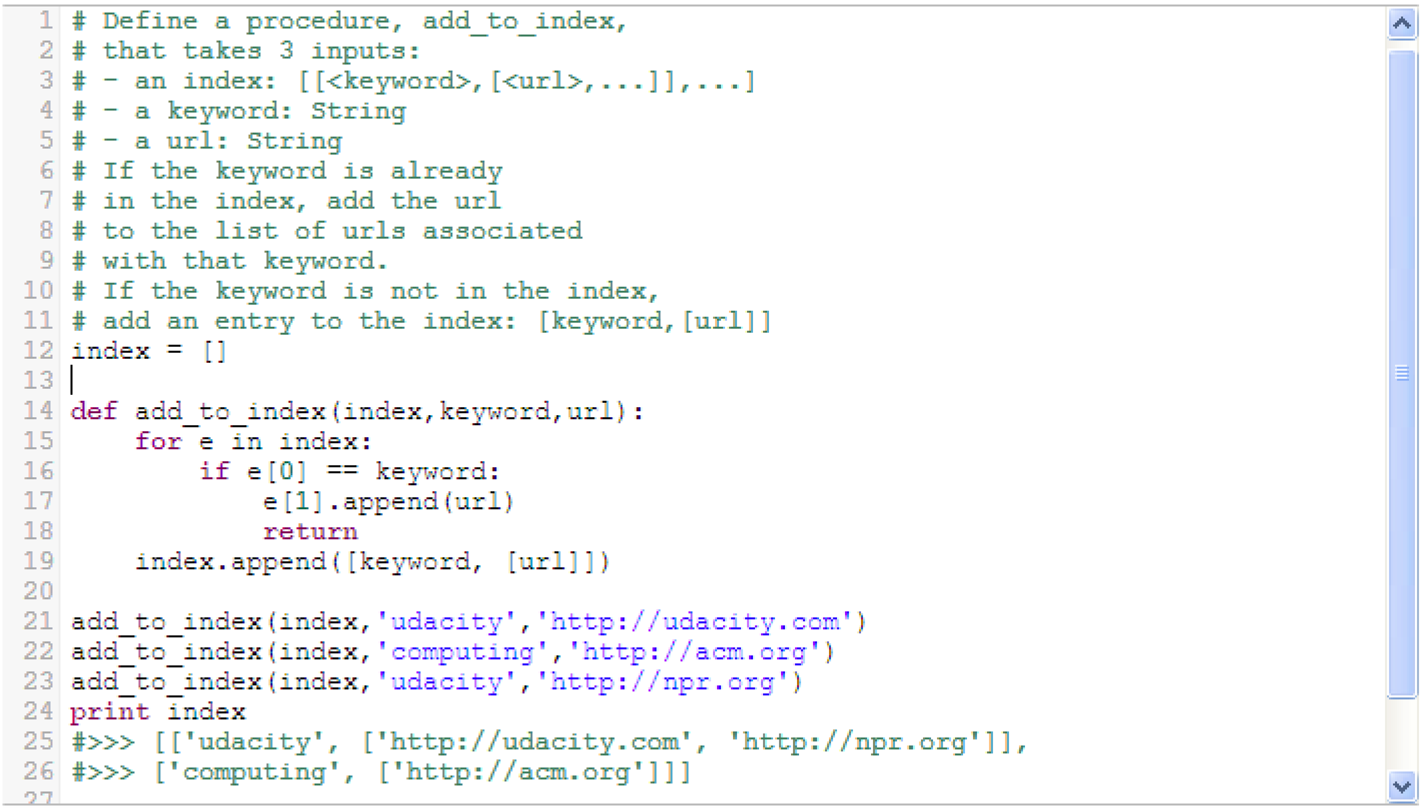
4.1 CS101: Building a Search Engine

4.2 Introduction

In unit 4 you are going to learn:

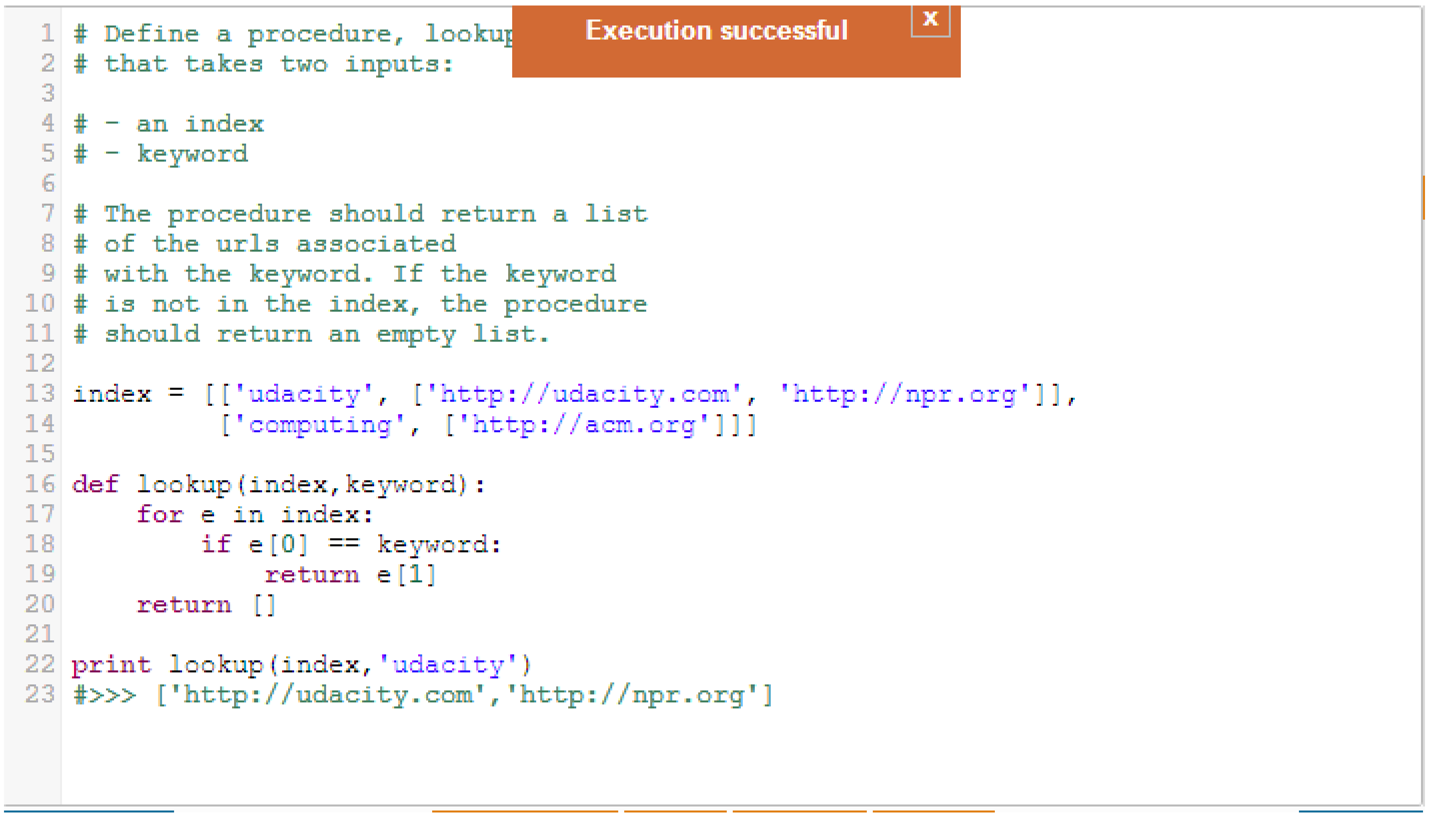
- how to finish the code for your search engine and how to **respond** to a **query** when someone wants the given web pages that correspond to a **given keyword**.
- about **how networks** and the **world wide web work** to understand more about how you can build up your search index.

4.3 Add to Index



```python
# Define a procedure, add_to_index,
# that takes 3 inputs:
# - an index: [[<keyword>,[<url>,...]],...]
# - 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 the index: [keyword,[url]]
index = []

def add_to_index(index,keyword,url):
    for e in index:
        if e[0] == keyword:
            e[1].append(url)
            return
    index.append([keyword, [url]])

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

4.4 Lookup Procedure



```python
# Define a procedure, lookup
# that takes two inputs:

# - an index
# - keyword

# The procedure should return a list
# of the urls associated
# with the keyword. If the keyword
# is not in the index, the procedure
# should return an empty list.

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

def lookup(index,keyword):
    for e in index:
        if e[0] == keyword:
            return e[1]
    return []

print lookup(index,'udacity')
#>>> ['http://udacity.com','http://npr.org']
```

4.5 Building the Web Index

To build your web index, you want to find a way to **separate all the words on a web page** - Python has a built-in operation:

Notation 4.1. **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''>, … ]
```

Example. Using Split

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

Remark. Not perfect - notice the words with the commas.

Notation 4.2. **Triple quotes** (""") - allows you to define *one string over several lines*

4.5.1 add_page_to_index

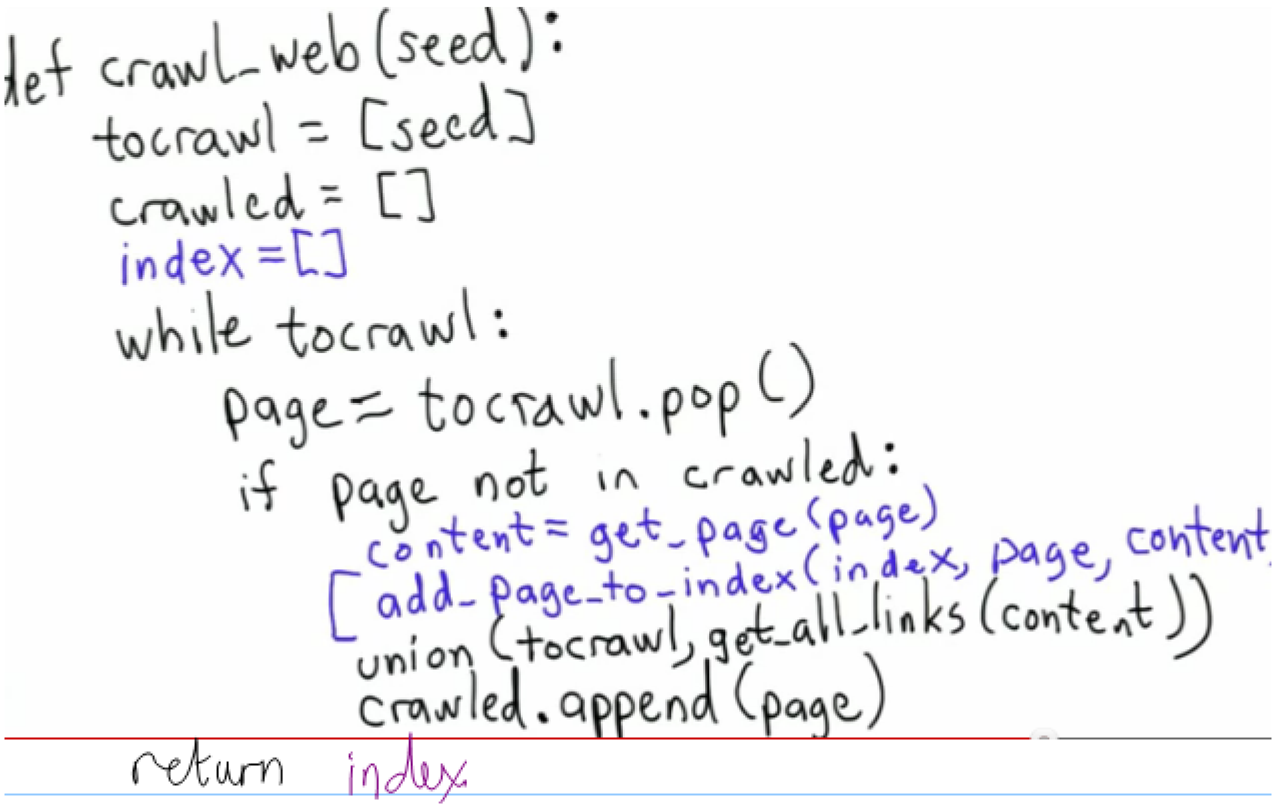
```
# Define a procedure, add_page_to_index,
# that takes three inputs:

#    - index
#    - url (String)
#    - content (String)

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

index = []

def add_page_to_index(index,url,content):
    words = content.split()
    for entry in words:
        index.append([entry, [url]])
    return


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

4.6 Finishing The Web Crawler

Recall the crawl_web code (3.20).

We now adapt the code so that you can use the information found on the pages crawled:

- First, we **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**.



```
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
```

4.7 Startup

A catchy name is key - Google, duckduckgo!.

We now have a functioning web crawler!

4.8 The Internet

We havee been using the `get_page function` where you **pass in a url** and it **passes out the content of the page:**

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

Remark. To read page on the WWW, we need to **import** from the Python library a function `urllib`.

Remark. The try and except work just like try and catch in MATLAB.

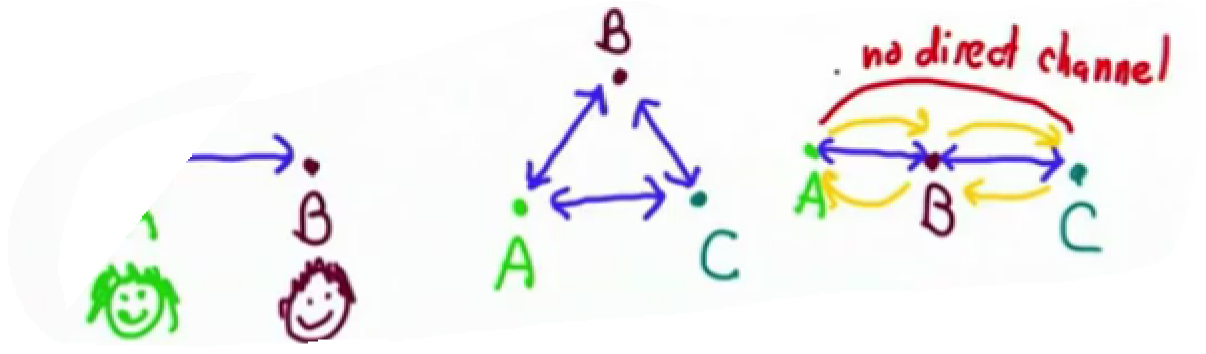
Notation 4.3. `.read()` -

Notation 4.4. `.urlopen()` -

4.9 Networks

The **Internet** is a **particular type** of **network**.

Definition 4.5. A **network** is a *group* of *entities* that *can communicate*, even though they are <u>*not directly* connected</u>.



4.10 Smoke Signals

The idea for networks have been around for over 3000 years - Greeks to ware when the Trojans are coming.

4.10.1 Making a Network

1. Method to encode and interpret messages.
2. Method to route messages.
3. Rules for deciding who gets to use resources.

4.11 Latency (measuring networks)

Definition 4.6. Latency is the *time* it takes a message to get from the source to the destination - measured in seconds/milliseconds.

Remark. If you care about your **ping** in online games, it's latency that matters.

4.12 Bandwidth (measuring networks)

Definition 4.7. Bandwidth is the *amount* of *information* that can be *transmitted per unit of time.*

"*The rate at which information is transferred.*"

Measured in units of *information/time*, such as **bits per second** - often measured in Mbps.

4.13 Bits

Definition 4.8. A **bit** is the smallest unit of information.

Remark. When asking yes-no (bit=on/off) questions, it is often better off asking a question that has an answer <u>**equally**</u> <u>**likely**</u> to be "Yes" as it is to be "No"

Remark. Anything that is **discrete,** (e.g.strings, lists, webpages...), can be converted into a number, and thus a bit and that number can then be transmitted.

Remark. 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 - **bandwidth.**

4.14 What Is Your Bandwidth

The common tests for testing bandwidth will try sending messages to figure out your bandwidth and depends on your location.

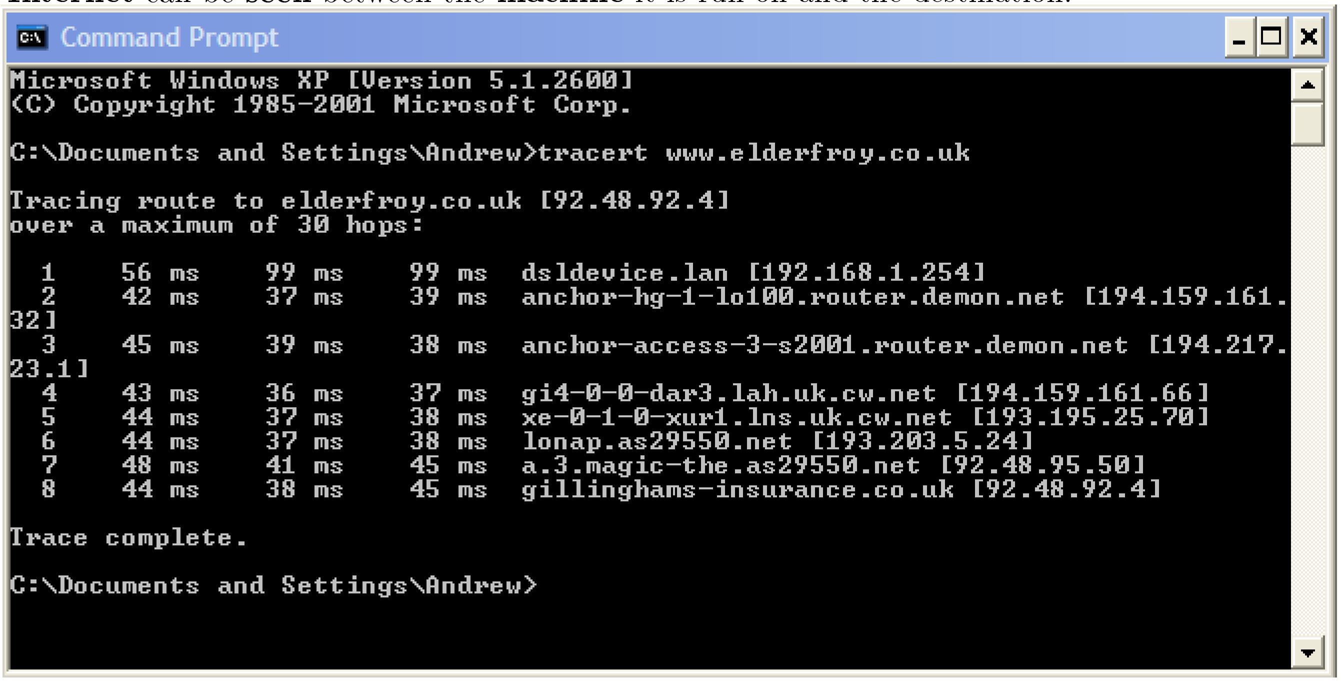
Remark. 10/07/2012 = Home Computer = 2.982 Mbps.

4.15 Traceroute(Mac)/Tracert(Windows)

You can learn a lot about the Internet by **measuring latency to different destinations**.

Example. For the map of Greece, there were hops on the way from Rhodes to Sparta:
Rhodes ->Naox -> Melos ->Sparta

Working with the command prompt and using an application called `traceroute`, the **hops** on the **Internet** can be **seen** between the **machine** it is run on and the destination.



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Andrew>tracert www.elderfroy.co.uk

Tracing route to elderfroy.co.uk [92.48.92.4]
over a maximum of 30 hops:

  1    56 ms    99 ms    99 ms  dsldevice.lan [192.168.1.254]
  2    42 ms    37 ms    39 ms  anchor-hg-1-lo100.router.demon.net [194.159.161.
32]
  3    45 ms    39 ms    38 ms  anchor-access-3-s2001.router.demon.net [194.217.
23.1]
  4    43 ms    36 ms    37 ms  gi4-0-0-dar3.lah.uk.cw.net [194.159.161.66]
  5    44 ms    37 ms    38 ms  xe-0-1-0-xur1.lns.uk.cw.net [193.195.25.70]
  6    44 ms    37 ms    38 ms  lonap.as29550.net [193.203.5.24]
  7    48 ms    41 ms    45 ms  a.3.magic-the.as29550.net [92.48.95.50]
  8    44 ms    38 ms    45 ms  gillinghams-insurance.co.uk [92.48.92.4]

Trace complete.

C:\Documents and Settings\Andrew>
```

Summary. **Tracert** 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. It returns the roundtrip (there and back) distance.

Here it took 8 hops, taking roughly $44ms$ (does three tests - $44ms, 38ms, 45ms$).

The first hop shows the **ip 192.168.1.254**. This is an **Internet address that refers to your router**. You will always start there.

Definition 4.9. A **packet** is a piece of information sent over the Internet.
It contains a *header* and a *message*.

- **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 - 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 tracert to send packets out different distances on the route to the destination. The hop limit is *only 1 byte long, which is 8 bits* - 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 distanc*e a packet can travel on the Internet at 255 hops. Fortunately, everywhere on the Internet is *connected by far fewer hops* than that!

Remark. The **three asterisks (*)** at the bottom of a `tracert` 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" - we may also need to change the time lapse.

4.16 Making a Network

Recall the 3 requirements for making a network:

1) Method to encode and interpret messages.
Internet: message → bits → electrons/photons
Any message can be encoded in bits and these can be encoded on a wire.

2) Method to route messages.
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.

3) Rules for deciding who gets to use resources.
Internet: best effort service
Unlike with the Greeks, there are no real rules on the Internet for who gets resources - it's a **wild west** where everywhere along the network gets to decide on its own what rules to apply.
If 2 messages are sent by a router at the same time, the router decides which one to send on. This means **packages might get dropped**. There is **no guarantee** that a package will reach its destination on the Internet.

4.17 Protocols

Definition 4.10. 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**.

Example. The protocol used on the **web** is called **Hypertext Transfer Protocol** which is abbreviated as HTTP.

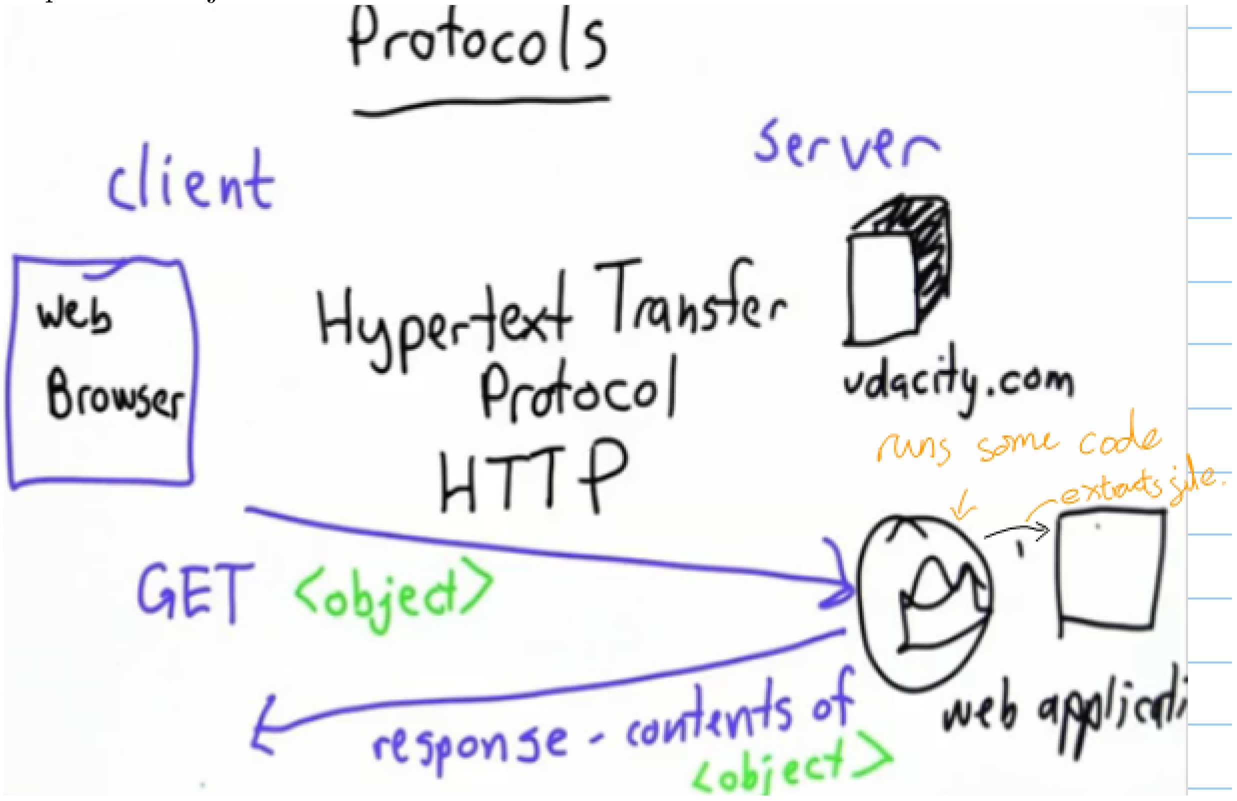
Almost all urls used start with http. That indicates that the **protocol to be used** to talk to the server is this Hypertext Transfer Protocol.

HTTP is a simple protocol and only has two main messages:

1) 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. Recall the python code for get_page:

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

2) The server receives 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>.



Remark. If you **click** on a **link**, 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.

4.18 Conclusion

A web browser, when requesting data over the Internet, the process involves:

- **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.

5 How Programs Run - Making things fast

5.1 Introduction

- With a **large index** and **lots** of **queries**, the previous 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***.

5.2 Making Things Fast

- What it **costs** to evaluate an execution is a very important - a fundamental problem in computer science (some spend their whole careers working on this.

Definition 5.1. Measuring what a algorithm cost to execute is called **algorithm analysis**.

Definition 5.2. An **algorithm** is a procedure that *always finishes* and *produces* the *correct result.*

Remark. We've already seen that it **isn't** an **easy** problem to **determine** if an algorithm always finishes - mathematical analysis come in here.

5.3 What is Cost?

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

```
Inputs → Algo 1 → Output
Inputs → Algo 2 → Output
```

Remark. For some `inputs`, `Algo 1` is cheaper than `Algo 2`, but for others, `Algo 2` might be cheaper.

We 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.

Remark. Computer scientists primarily talk about **cost** in terms of **input size** - as it is usually the main factor that determines the speed of the algorithm

*"Cost is measured in terms of **how time increases with input size**".*

Remark. If a certain amount of **memory** is needed to execute an algorithm, you have an indication of the size and cost of computer required to run the program.

5.4 Stopwatch

Algorithm 5.3. *The procedure, `time_execution`, is a way to **evaluate how long** it takes for some code to execute.*

We use the built-in `time.clock` in the time library.

```python
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 Python command
    run_time = time.clock() - start   # find difference in start and end time
    return result, run_time   # return the result of the code and time taken
```

Remark. The result is in seconds.

Remark. Running the timing through the web interpreter won't be accurate.

Remark. This doesn't tell you very much for short, fast executions.

5.5 Spin Loop

Algorithm. *spin_loop(n)*

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

Remark. It's **important** to notice that the **time changes depending on** the **input**.

5.6 Predicting Run Time (Quiz)

*Remark. "For `spin_loop`: Running Time is **linear** in the magnitude of n."*

5.7 Make Big Index

Idea - here we are creating a big index to test the speed of out `lookup` code.

Algorithm. *`make_big_index(size)`*

```
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
```

with `make_string` representing:

```
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')
```

which produces:

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

print make_big_index(100) # index with 100 keywords
[['aaaaaaaa', ['fake']], ['aaaaaaab', ['fake']], ['aaaaaaac', ['fake']], ['aaaaaaad
['fake']], ['aaaaaaaf', ['fake']],
... <snip> ...
['aaaaaadm', ['fake']], ['aaaaaadn', ['fake']], ['aaaaaado', ['fake']], ['aaaaaadp',
['fake']], ['aaaaaadr', ['fake']], ['aaaaaads', ['fake']], ['aaaaaadt', ['fake']], ['aaaaa
['fake']]]
```

When running `lookup(index, “udacity”)`, we establish a linear running time again.

Remark. Timings vary for many reasons:

1) Lots of other things run on the computer - so a program does not have total control over the processor.

2) Where things are in memory, it can take a longer or shorter time to retrieve.

5.8 Worst Case

- Usually, when analysing programs it’s the **worst-case execution time** that is **important**.

Definition 5.4. 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.*

E.g. For lookup, it’s when the keyword is the *last entry* in the index, or *not in* the index at all.

5.9 Making Lookup Faster

`lookup` is slow because it has to go through the whole of the for loop. When we look in a index of a book, the index is sorted alphabetically, and so is much quicker to search for a keyword.

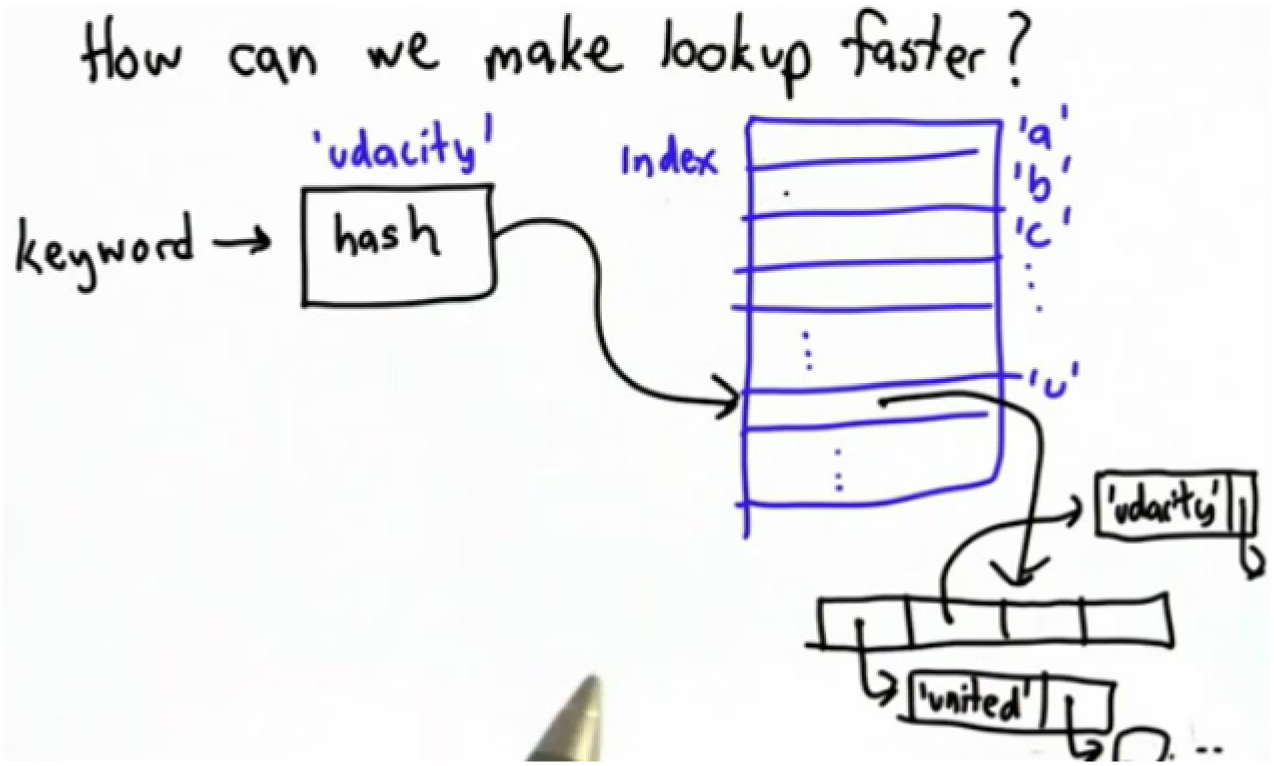
If a word isn’t where it belongs, it isn’t in the index.

Algorithm 5.5. *Given a keyword, the **hash function** will tell you where to look in the index.*

It maps the keyword to 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.

Hash Table - very useful **data structure** - so useful that it’s built into Python; a type 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.



Remark. **Not the best method** - 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*).

Remark. **Not great for English** - 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.*

Remark. If you have *millions* of keywords it would not fast enough.

Summary. There are 2 problems to fix:

1) Make a function depending on the **whole** word
2) Make the function **distribute** the keywords **evenly** between the buckets.

5.10 Hash Function (Our's to understand)

In creating our own hash function, we nee to be able to convert numbers in letter and visa versa:

Notation 5.6. `ord` and `chr`

```
ord(<one-letter string>) → Number
chr(<Number>) → <one-letter string>
```

Examples:

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

Remark. This numbers are based on ASCII character encoding.

Remark. `chr` and `ord` are inverse to one another:

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

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

5.11 Modulus Operator

The **modulus operator** (%) will be the tool used to change whatever values are calculated for strings into the range 0 to $b-1$ (where b is the number of buckets).

```
<''number''> % <''modulus''> → <''remainder''>
```

5.12 Bad Hash

The **hash function** takes 2 inputs, the *keyword* and the *number of buckets* and outputs a $n \in \{0,1,2,...,b-1\}$ which gives you the position where that string belongs.

Example. Bad has function:

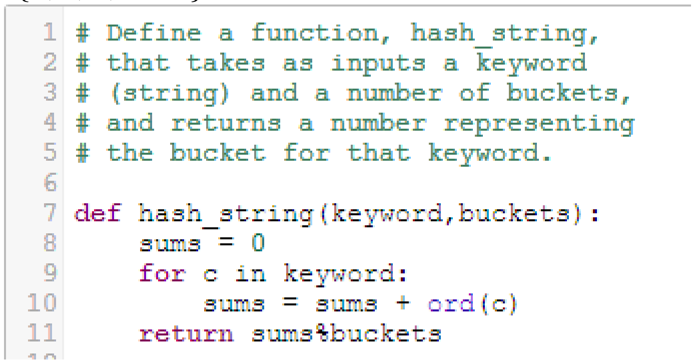
```
def bad_hash_string (keyword, buckets):
    return ord(keyword[0]) % buckets # output is the bucket based on the first lett
```

Remark. Bad as:

1) It produces an error for one input keyword - (the empty string “”).
2) If the keywords are distributed like words in English, some buckets will get too many words.
3) If the number of buckets is large, some buckets will not get any keywords.

5.13 Better Hash Function

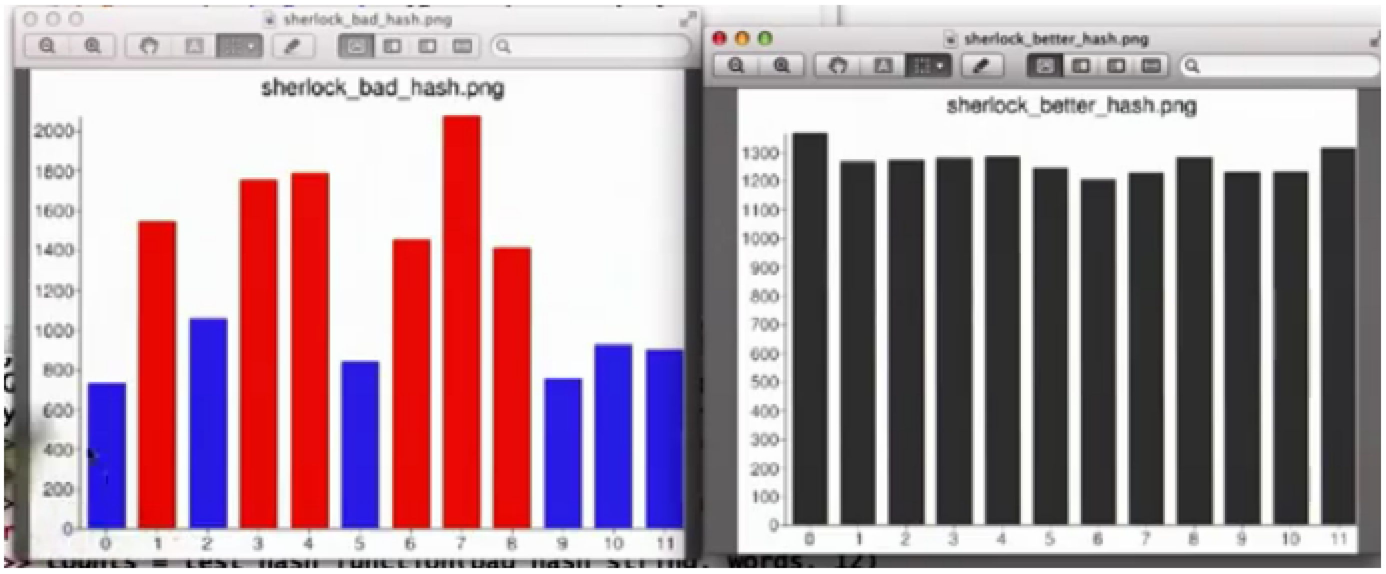
You are going to want to look at **more than just one letter** of the keyword - now we compute $n \in \{0,1,..,b-1\}$ based on all the letters to decide their bucket.



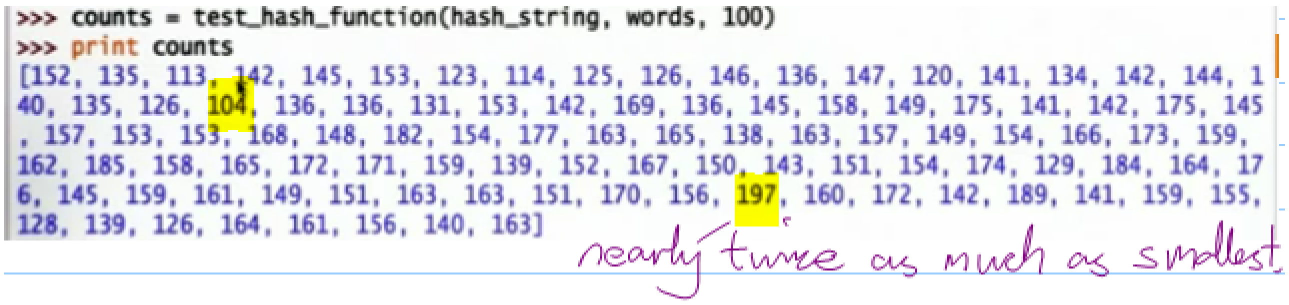
```
# Define a function, hash_string,
# that takes as inputs a keyword
# (string) and a number of buckets,
# and returns a number representing
# the bucket for that keyword.

def hash_string(keyword,buckets):
    sums = 0
    for c in keyword:
        sums = sums + ord(c)
    return sums%buckets
```

5.14 Testing Hash Functions

```
counts = test_hash_function(hash_string, words, 12) # find the distribution for the new
[1363, 1235, 1252, 1257, 1285, 1256, 1219, 1252, 1290, 1241, 1217, 1303]
```



If we **increase** the **number** of **buckets**:



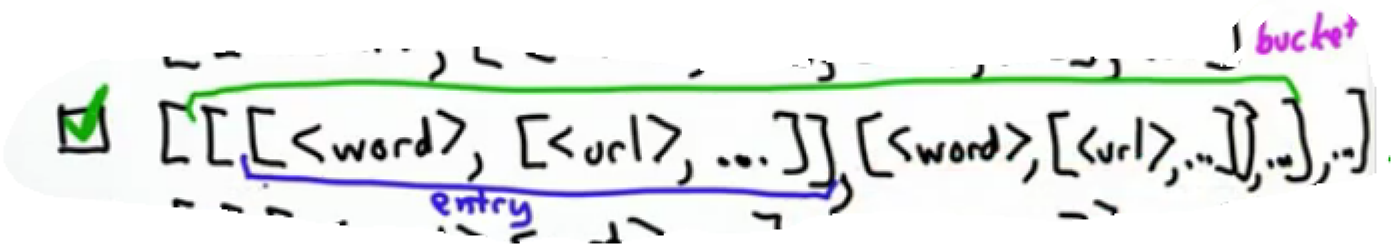
```
>>> counts = test_hash_function(hash_string, words, 100)
>>> print counts
[152, 135, 113, 142, 145, 153, 123, 114, 125, 126, 146, 136, 147, 120, 141, 134, 142, 144, 1
40, 135, 126, 104, 136, 136, 131, 153, 142, 169, 136, 145, 158, 149, 175, 141, 142, 175, 145
, 157, 153, 153, 168, 148, 182, 154, 177, 163, 165, 138, 163, 157, 149, 154, 166, 173, 159,
162, 185, 158, 165, 172, 171, 159, 139, 152, 167, 150, 143, 151, 154, 174, 129, 184, 164, 17
6, 145, 159, 161, 149, 151, 163, 163, 151, 170, 156, 197, 160, 172, 142, 189, 141, 159, 155,
128, 139, 126, 164, 161, 156, 140, 163]
```

nearly twice as much as smallest.

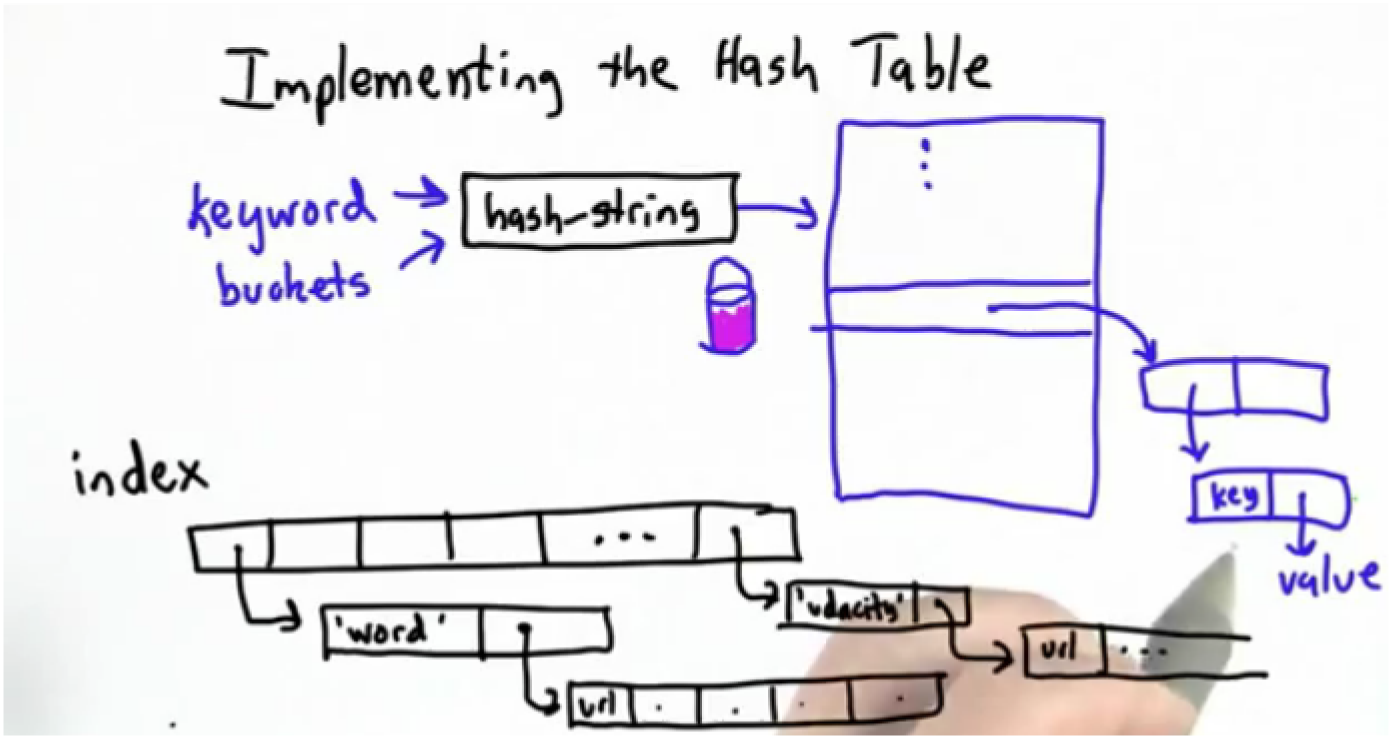
Remark. The time it take to index using a hash table depends on *the number of keywords per bucket* $\approx \frac{k}{b}$

5.15 Implementing Hash Tables

The data structure for our hash table follows:

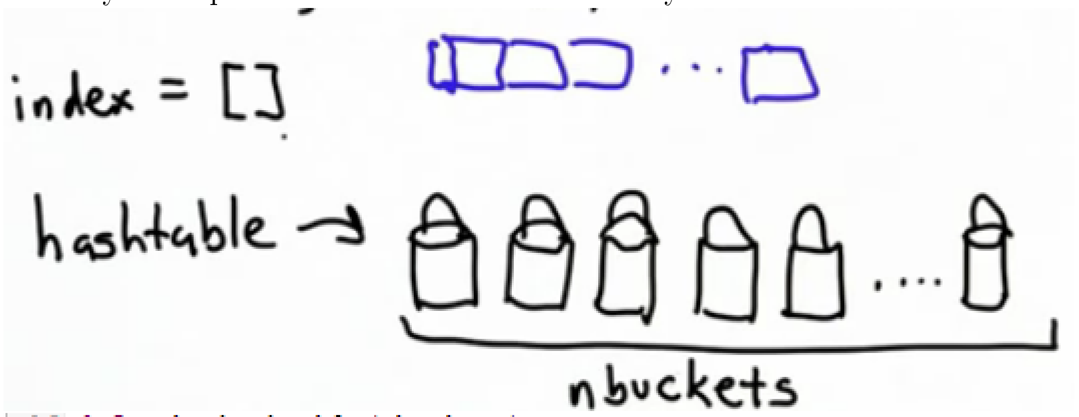

[[[<word>, [<url>, ...]], [<word>, [<url>, ...]], ...], ...]
bucket
entry

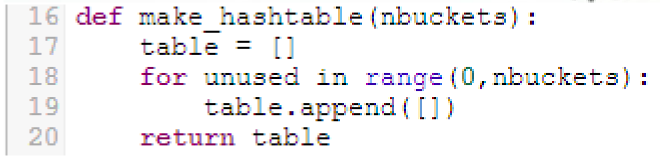
We can see this is true by seeing the more visual representation of the hash table.



5.16 Empty Hash Table

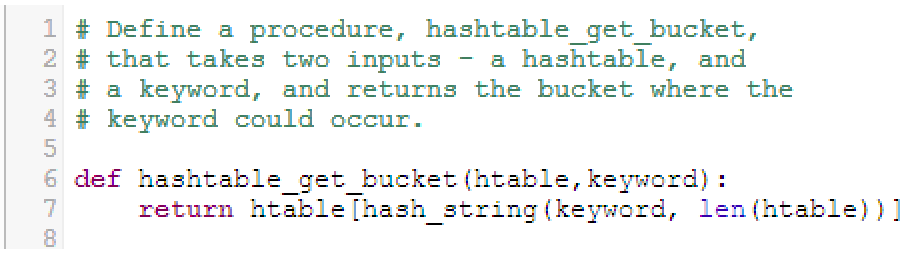
It's a good idea to create an empty hash table so Python knows what it's working with. More importantly we require it as then we can successfully add elements to it:





```
def make_hashtable(nbuckets):
    table = []
    for unused in range(0,nbuckets):
        table.append([])
    return table
```

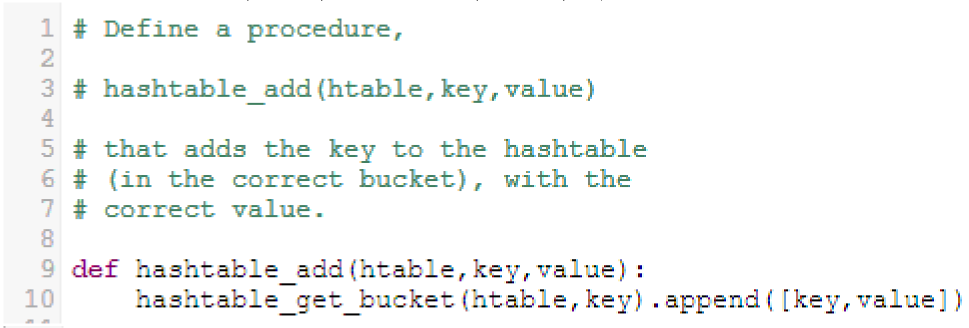
5.17 Finding Buckets



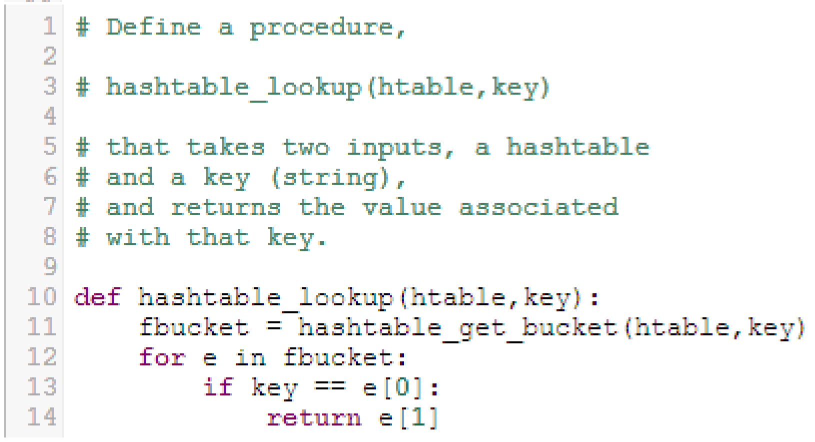
```python
# Define a procedure, hashtable_get_bucket,
# that takes two inputs - a hashtable, and
# a keyword, and returns the bucket where the
# keyword could occur.

def hashtable_get_bucket(htable,keyword):
    return htable[hash_string(keyword, len(htable))]

```

5.18 Operations on Hash Table

You wish to **lookup (read)** and **add (write)**, (depend on first being able to find the right bucket):



```python
# Define a procedure,

# hashtable_add(htable,key,value)

# that adds the key to the hashtable
# (in the correct bucket), with the
# correct value.

def hashtable_add(htable,key,value):
    hashtable_get_bucket(htable,key).append([key,value])
```



```python
# Define a procedure,

# hashtable_lookup(htable,key)

# that takes two inputs, a hashtable
# and a key (string),
# and returns the value associated
# with that key.

def hashtable_lookup(htable,key):
    fbucket = hashtable_get_bucket(htable,key)
    for e in fbucket:
        if key == e[0]:
            return e[1]
```

To prevent duplication we use `hashtable_update`:

```
# Define a procedure,

# hashtable_update(htable,key,value)

# 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.

# Hint: Use hashtable_lookup as a
# starting point.

def hashtable_update(htable,key,value):
    bucket = hashtable_get_bucket(htable,key)
    for entry in bucket:
        if entry[0] == key:
            entry[1] = value
            return
    bucket.append([key, value])
```

5.19 Dictionaries (using the built-in hash function - dictionary type)

- Created using curly brackets { }, and consists of **key:value** pairs.
- The **keys** can be any ***immutable type***, and the **values** can be of ***any type***.
- **Dictionaries** are also ***mutable*** and its **key:value** pairs are updated like in our update function for hash tables.

5.20 Using Dictionaries

Example. Using the dictionary type:

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

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

Remark. **No order** - as in the hash table, the order depends on the order of buckets (the hash function).

Remark. Printing dictionaries returns the values:

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

Remark. Unlike our hash table, printing elements not in a dictionary returns an error as opposed to the value `None`.

```
print elements['lithium']
Traceback (most recent call last):
 File "C:\Users\Sarah\Documents\courses\python\dummy.py", line 5, in <module>
        print elements['lithium']
KeyError: 'lithium'
```

To prevent this error, you can check if a key is in the dictionary using `in`, just like with lists - returns `True` or `False`.

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

Remark. Recall dictionaries are mutable - can add and change easily:

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

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

print element['nitrogen']
8
```

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

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

5.21 A Noble Gas

Note. Values don't have to be numbers or strings - they can be anything.

They can even be **other dictionaries**. The next example will have atomic symbols as keys with associated values which are dictionaries.

```
elements = {}
elements['H'] = {'name': 'Hydrogen', 'number': 1, 'weight': 1.00794}
elements['He'] = {'name': 'Helium', 'number': 2, 'weight': 4.002602,
                  'noble gas': True}
```

```
print elements
{'H': {'name': 'Hydrogen', 'weight': 1.00794, 'number': 1},
'He': {'noble gas': True, 'name': 'Helium', 'weight': 4.002602, 'number': 2}}
```

5.22 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** (as long as we increase the number of buckets as we increase the list of keywords) rather than linear time.

5.23 Changing Lookup (to work with dictionaries)

```
# Change the lookup procedure
# to now work with dictionaries.

def lookup(index, keyword):
    if keyword in index:
            return index[keyword]
    return None

```

5.23.1 Conclusion

We 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.

6 How to Have Infinite Power

6.1 Infinite Power

We introduce **recursive definitions**, a method for increasing your page ranking – being able to find the best page to respond to the query.

6.2 Long Words

"*There's no such things as the longest word in the English dictionary - I can always make a bigger one.*"

Definition 6.1. A **Word** is something that has a meaning.

6.3 Counter

There is a rule in English that says that for a word, you can make a new word by adding **counter** in front of the old one.

$$Word \longrightarrow counter - Word$$

Remark. The meaning counter-word is something that goes against, or is counter to the original word.

Remark. This provides a method to create an infinitely long word.

$$\begin{gathered} intelligence \\ counter - intelligence \\ counter - counter - intelligence \\ counter - counter - counter - intelligence \\ add\,infintum \\ Q.E.D \end{gathered}$$

6.4 Recursive Definitions

A recursive definition has 2 parts:

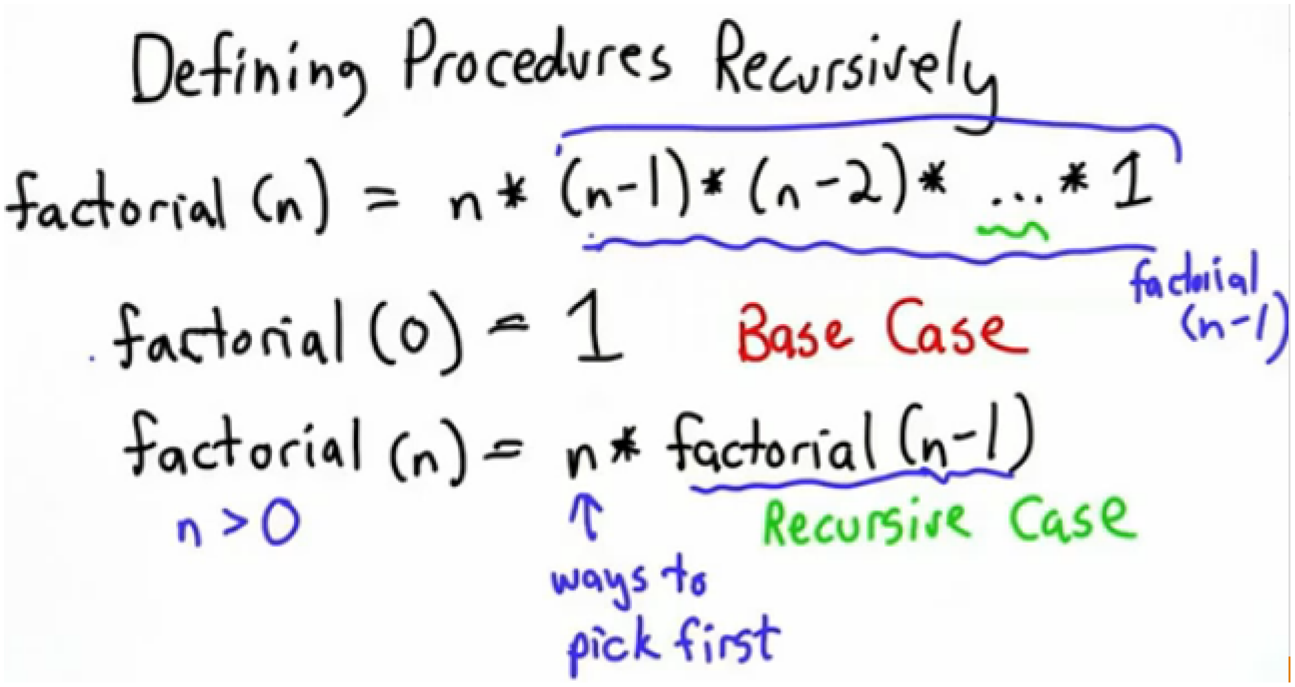
1. **Base case**:
   (a) A *starting* point and is *not* defined in terms of itself.
   (b) Usually the *smallest* or *simplest* input - already know how to define.

2. **Recursive case**:
   (a) Defined in terms of itself, but *not* itself exactly.
   (b) Defined in terms of a *smaller version* of itself, as *progress* is *made towards* the *base* case.

6.5 Ancestors (example of recursion)

Your parents are your ancestors, but they are not your only ancestors. Your parents have parents – your grandparents, who are also your ancestors. Your grandparents also have parents who are your ancestors too, and so on.

6.6 Recursive Procedures


Defining Procedures Recursively
factorial (n) = n * (n-1) * (n-2) * ... * 1
factorial (n-1)
factorial (0) = 1
Base Case
factorial (n) = n * factorial (n-1)
n > 0
ways to pick first
Recursive Case

Remark. There are n ways to pick the first item, and then there are $n-1$ items remaining. There are $factorial(n-1)$ ways to arrange these $n-1$ items.

6.7 Palindromes

Definition 6.2. A **palindrome** is a string that *reads* the *same way forwards* and *backwards.*

Example. “stanley yelnats”, “”, “a” etc.

Algorithm. *is_palindrome(s)*

```python
# Define a procedure is_palindrome, that takes as input a string, and returns a
# Boolean indicating if the input string is a palindrome.

# Base Case: '' => True
# Recursive Case: if first and last characters don't match => False
# if they do match, is the middle a palindrome?

def is_palindrome(s):
    if "" == s:
        return True
    if s[0] != s[-1]:
        return False
    else:
        return is_palindrome(s[1:-1])

print is_palindrome('')
#>>> True
print is_palindrome('abab')
#>>> False
print is_palindrome('abba')
#>>> True
```

6.8 Recursive v. Iterative

Remark. Any procedure that you write recursively can also be written iteratively.

```python
def iter_palindrome(s):
    for i in range(0, len(s) / 2):
        if s[i] != s[-(i + 1)]:
            return False
    return True
```

Remark. This recursive version makes a **new string** every time you make a recursive call.
This step is pretty expensive.

Note. **Recursive calls** themselves are **fairly expensive**.

There are languages which make recursive calls really cheap - Python is not one of them; thus for large inputs, defining procedures iteratively are generally more efficient.

6.9 Bunnies

Remark. **Fibonacci Numbers** are one of the most interesting things in mathematics.

Once you know about them you will start to see them all over the place, both in nature and design.

Abstract

The name comes from Leonardo da Pisa, who is also known as Fibonacci. In 1202 he published a book called, Liber Abaci. The root, abaci, is the same for the word abacus, the calculating machine. Liber Abaci is loosely translated as the "book of calculation." The book introduced Indian mathematics to the West, particularly, Arabic numerals. Arabic numerals soon replaced the Roman numeral system, which had been widely used. In his book, Fibonacci showed how much easier it is to do calculations using numbers in the decimal system where the position of the number indicates its value. He showed this by introducing problems and using calculation to solve them.

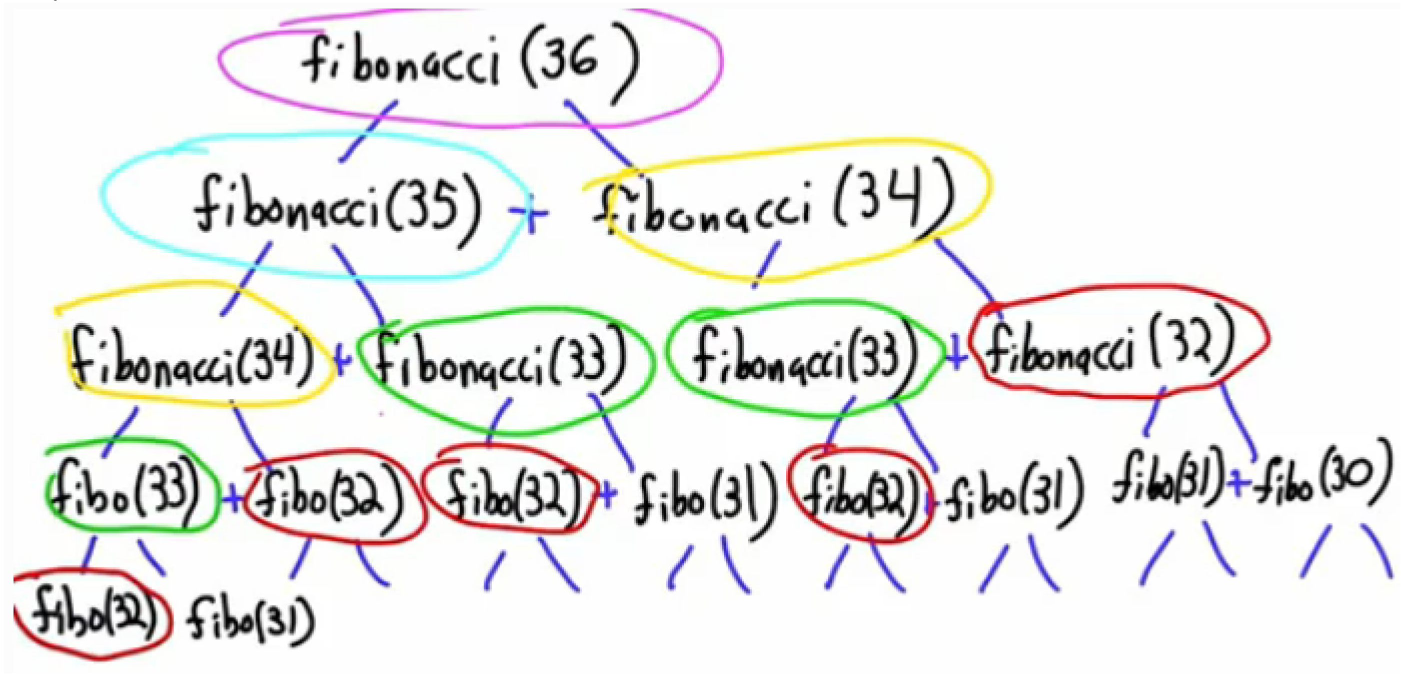
Algorithm. *Identifying a Fibonacci Number:*

```python
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

6.10 Divide and Be Conquered

Remark. The previous algorithm is inefficent. Computing `fibonacci(36)` (number of rabbits in 3 years) makes Python time out.

Why?


fibonacci (36)
fibonacci(35) + fibonacci (34)
fibonacci(34) + fibonacci(33) fibonacci(33) + fibonacci (32)
fibo(33) + fibo(32) fibo(32) + fibo(31) fibo(32) fibo(31) fibo(31) + fibo (30)
fibo(32) fibo(31)

Lots of redundant computations take place (notice a pattern?):
We need to evaluate `fibonacci(32)` 5 times.
We need to evaluate `fibonacci(33)` 3 times.
We need to evaluate `fibonacci(34)` 2 times.
We need to evaluate `fibonacci(35)` 1 time.
We need to evaluate `fibonacci(36)` 1 time.

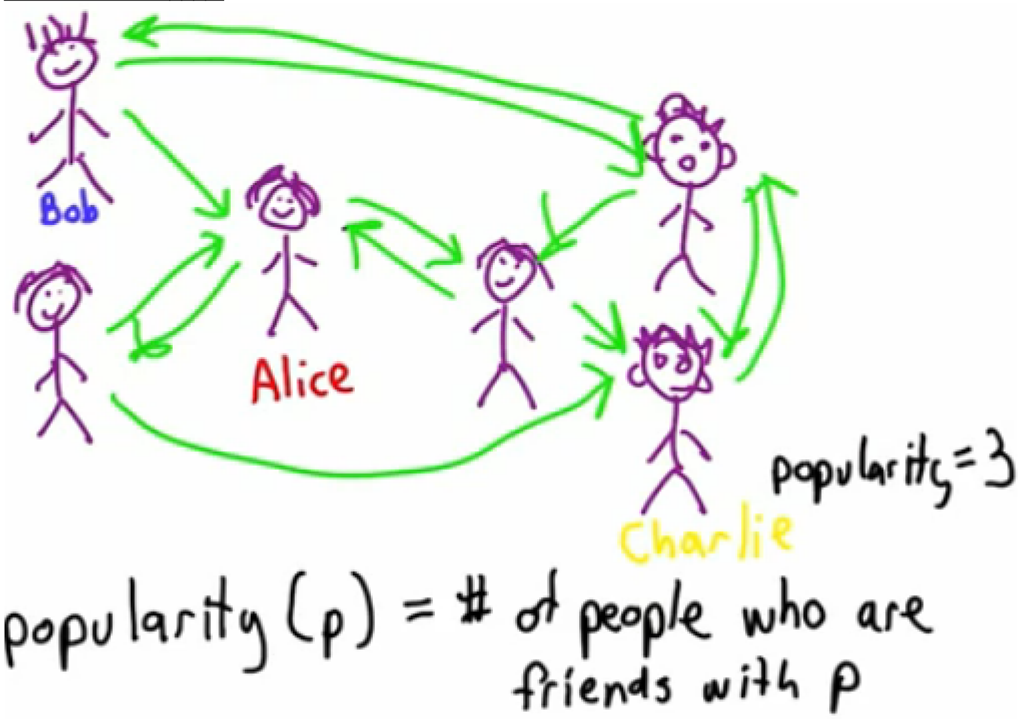
6.10.1 Faster Fibonacci

```python
#Define a faster fibonacci procedure that will enable us to computer
#fibonacci(36).

def fibonacci(n):
    current = 0
    after = 1
    for i in range(0,n):
        current, after = after, current+after
    return current
```

6.10.2 Ranking Web Pages

A good search engine ranks the pages so that the one at the front of the list is the one the user most likely wants.

Popularity - 1



Consider a typical group of friends in middle school. One way to decide popularity is to look at **friendship links**. Friendship links are go in one direction. Just because Bob is friends with Alice does not mean Alice is friends with Bob.

For the diagram above:

- `popularity(Charlie) = 3`
- `popularity(Alice) = 3`
- `popularity(Bob) = 1`

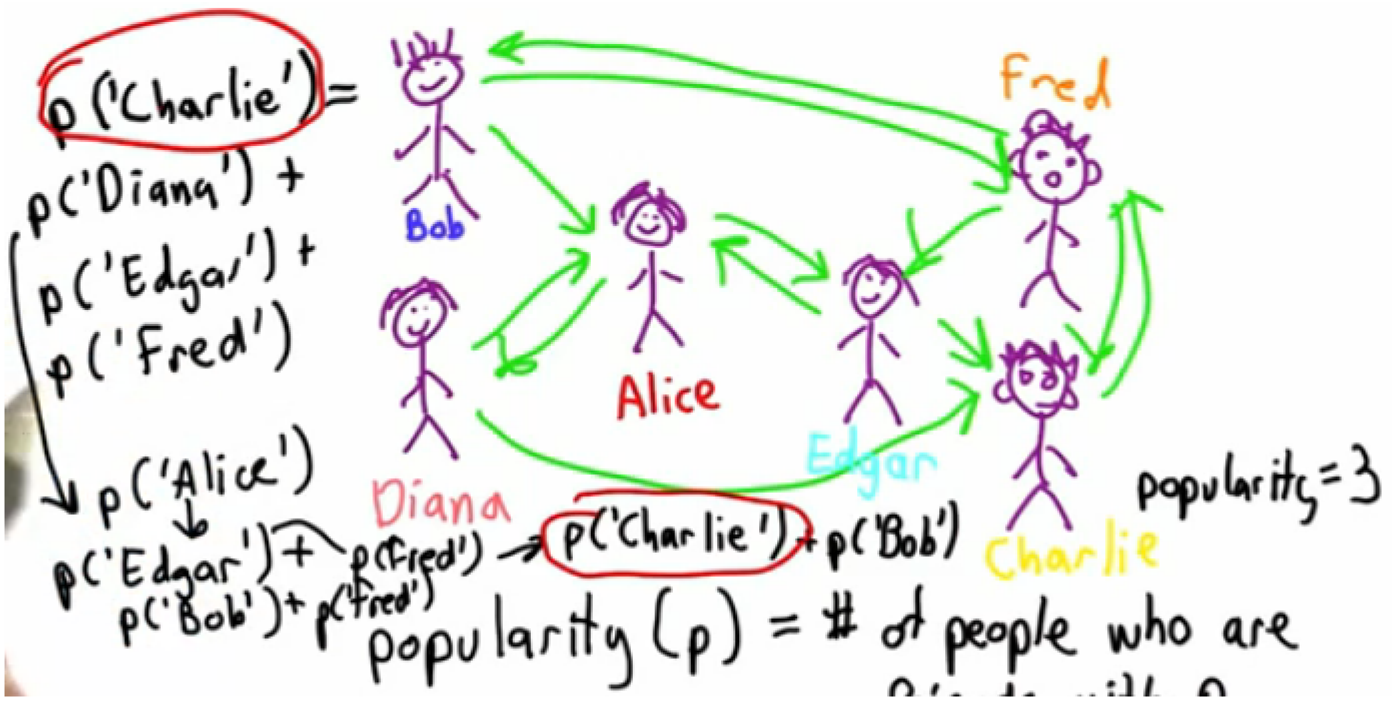
6.10.3 Popularity - 2

It's no good to have lots of friends with no friends (geeks), you have to have friends who are popular. Popularity is about having l**ots of friends who have lots of friends**. So:

$$popularity\,(p) = \sum_{f \in friends(p)} popularity\,(f) \tag{6.1}$$

6.11 Circular Definitions

6.1 is not well-defined recursively - we have no base case.



6.12 Relaxation

There is no sensible base case that provides a good recursive definition. Instead, an algorithm called the **relaxation algorithm** can be used.

Algorithm 6.3 (**Relaxation algorithm**).
__Start__ with a __guess__ and then loop where you do something to __improve__ the __guess__.
Each time you go through the loop, the __guess is refined__.
At __some point__ you'll stop and take that to be the result you want.

```
# start with a guess
while not done:
    make the guess better
```

Note. This procedure requires an extra parameters:

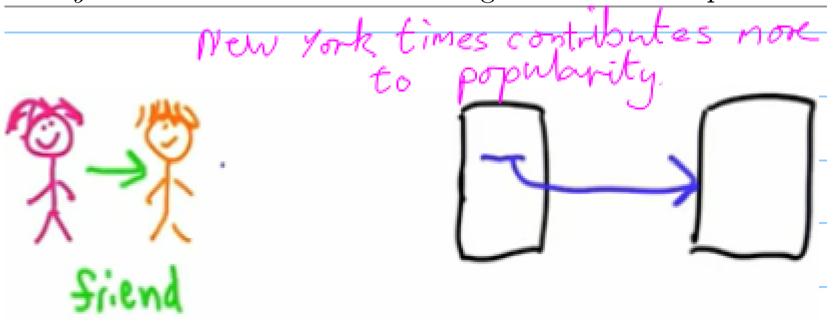
```
popularity(<time step> ,<person>) --> score
```

Remark. **Base case** - the popularity for *everyone* at time 0 to 1.
$popularity\,(0, p) = 1$
Recursive step - for $t > 0$, the popularity of each of their friends at the *previous time step*, $t - 1$ is summed.
$popularity\,(t, p) = \sum_{f \in friends(p)} popularity\,(t-1, p)$

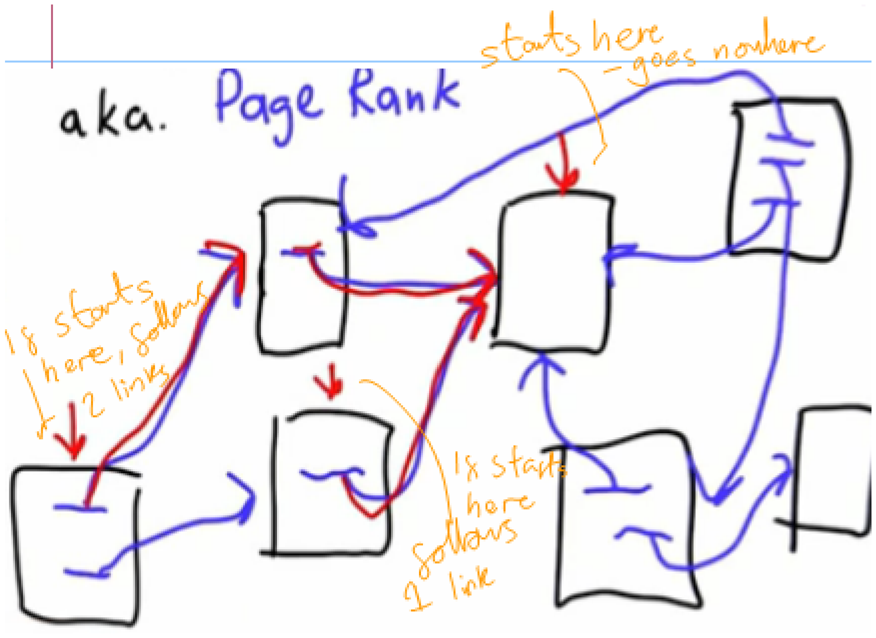
```
def popularity(t,p):
    if t == 0:  # base case, at time step 0
        return 1 # the score is always 1
    else:
        score = 0
        for f in friends(p): # summing over the friends
            score = score + popularity(t-1,f) # adding the popularity at the time ste
        return score
```

6.13 Page Rank

Summary. Links on the web are analogous to friendships



- This model is a **random** web surfer who **starts** at a **random page** and then **follows** the **links** at **random**.
  - The **popularity** of a **page** is the **probability** that the **random** surfer **reaches** that **page**.

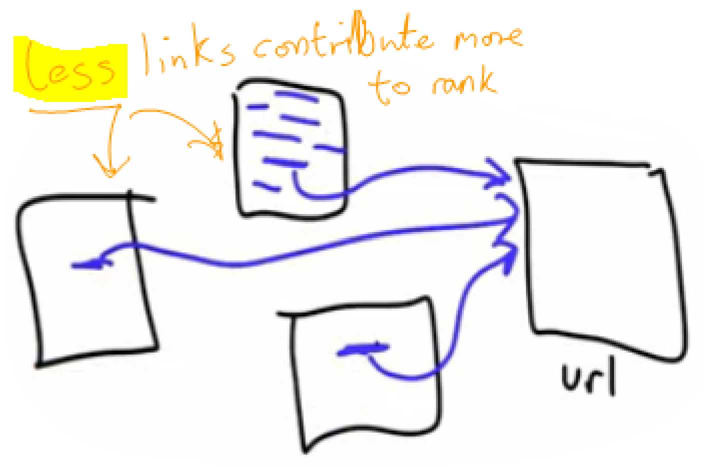

aka. Page Rank
starts here
-goes nowhere
1/8 starts here, follows 2 links
1/8 starts here follows 1 link

- The rank function is defined recursively over time:

$$rank\,(<timestep>, <page>) \longrightarrow value$$
$$\text{Base case: } rank\,(0, url) \longrightarrow 1$$
$$rank\,(t, url) \longrightarrow \sum_{p \in inlinks[url]} \frac{rank\,(t-1, p)}{outlinks\,[p]}$$

Remark. The rank **contributed** by each page is **inversely weighted** by the **number** of **outlinks** from that page.

"So we **divide** each rank in the sum by the number of outlinks from that page."



6.14 Altavista - (a more popular search engine in 1988)

- **Pages with no links have a rank of 0**, which makes it very hard to start a new page (also we will be dividing by 0 is the recursive step above). So, instead each page will have some starting rank greater than 0.
- The model represents the **probability** that a random surfer reached a given page - the ranks should be a **probability distribution**.
  - The sum of ranks for all pages will sum up to 1.
  - So at time 0, instead of 1, the rank is $\frac{1}{N}$ for each page, where N is the number of pages.
- A **damping constant** is used to **diminish** the **raw values** of the ranking algorithm. In the model, this represents the probability that a page was reached via following a link. Set the damping constant, d, to 0.8 for now.
- At time t, **add** $\frac{(1-d)}{N}$ to **each rank** that will represent the probability that the page was not reached via following a link. Multiply the first term, the sum, by d. This will make the ranks a probability distribution at any given time step.

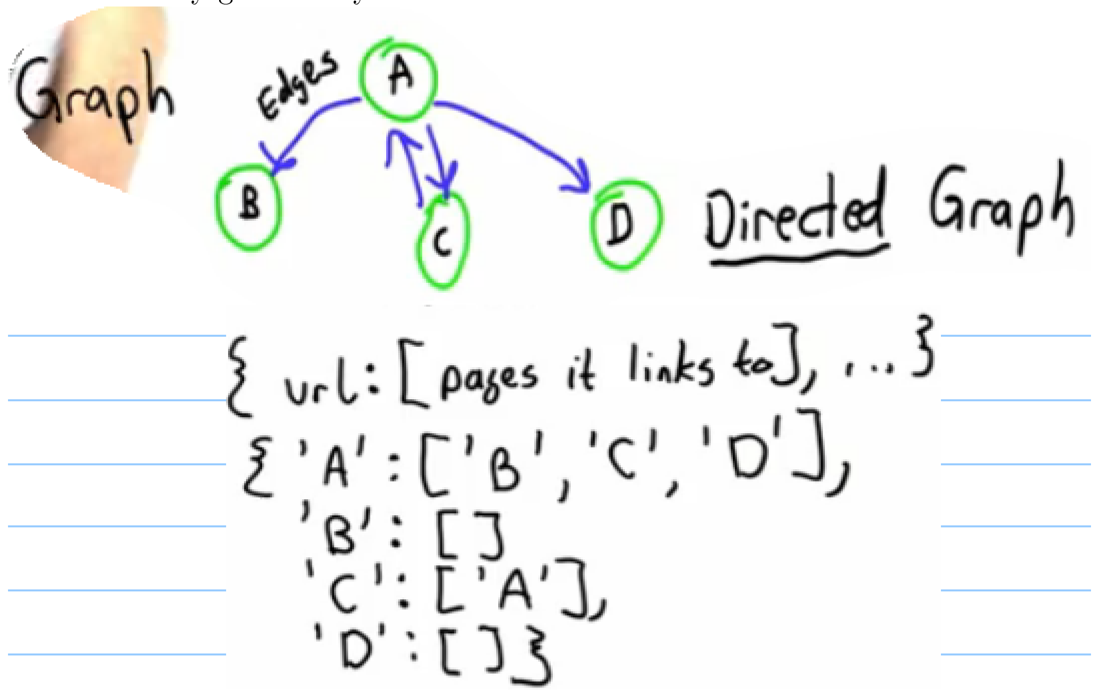
$$rank\left(t, url\right) \longrightarrow d \sum_{p \in inlinks[url]} \frac{rank\left(t-1, p\right)}{outlinks\left[p\right]} + \frac{1-d}{N}$$

6.15 Urank

Since **PageRank is a registered trademark of Google**, the algorithm will be called **URank** instead.

- **URank** needs to **keep track** of **which pages link to which pages**
- The data structure we use is a **directed graph**.

Definition 6.4. A **directed graph** is a data structure where nodes are linked to other nodes, and the links only go one way.



So `crawl_web` will now produce a graph in addition to an `index`, where the `graph` gives a mapping from each page to all the pages it links to:

```python
# Modify the crawl_web procedure so that instead of just returning the
# index, it returns an index and a graph. The graph should be a
# Dictionary where the key:value entries are:

#  url: [list of pages url links to]


def crawl_web(seed): # returns index, graph of outlinks
    tocrawl = [seed]
    crawled = []
    graph = {}  # <url>:[list of pages it links to]
    index = {}
    while tocrawl:
        page = tocrawl.pop()
        if page not in crawled:
            content = get_page(page)
            add_page_to_index(index, page, content)
            outlinks = get_all_links(content)
            graph[page] = outlinks
            union(tocrawl, outlinks)
            crawled.append(page)
    return index, graph
```

6.16 Computing Page Rank

The output of `compute_ranks` is a dictionary mapping each URL to its rank, which is a number.

6.17 Formal Calculations

```
rank(0, url) = 1/npages
rank(t, url) = (1-d)/npages +
  sum([for each page 'p' that links to URL,
    d*rank(t-1, p)/(number of outlinks from p)])
```

- Since the ranks should **not depend** on the **order** that the **pages** were examined by the algorithm, you need to **keep track of the ranks at the last time step**.
- We keep 2 separate dictionaries, `ranks` and `newranks`, where `newranks` is the working space for each time step (and `ranks` is the ranks at time $t-1$).

6.18 Computer Ranks

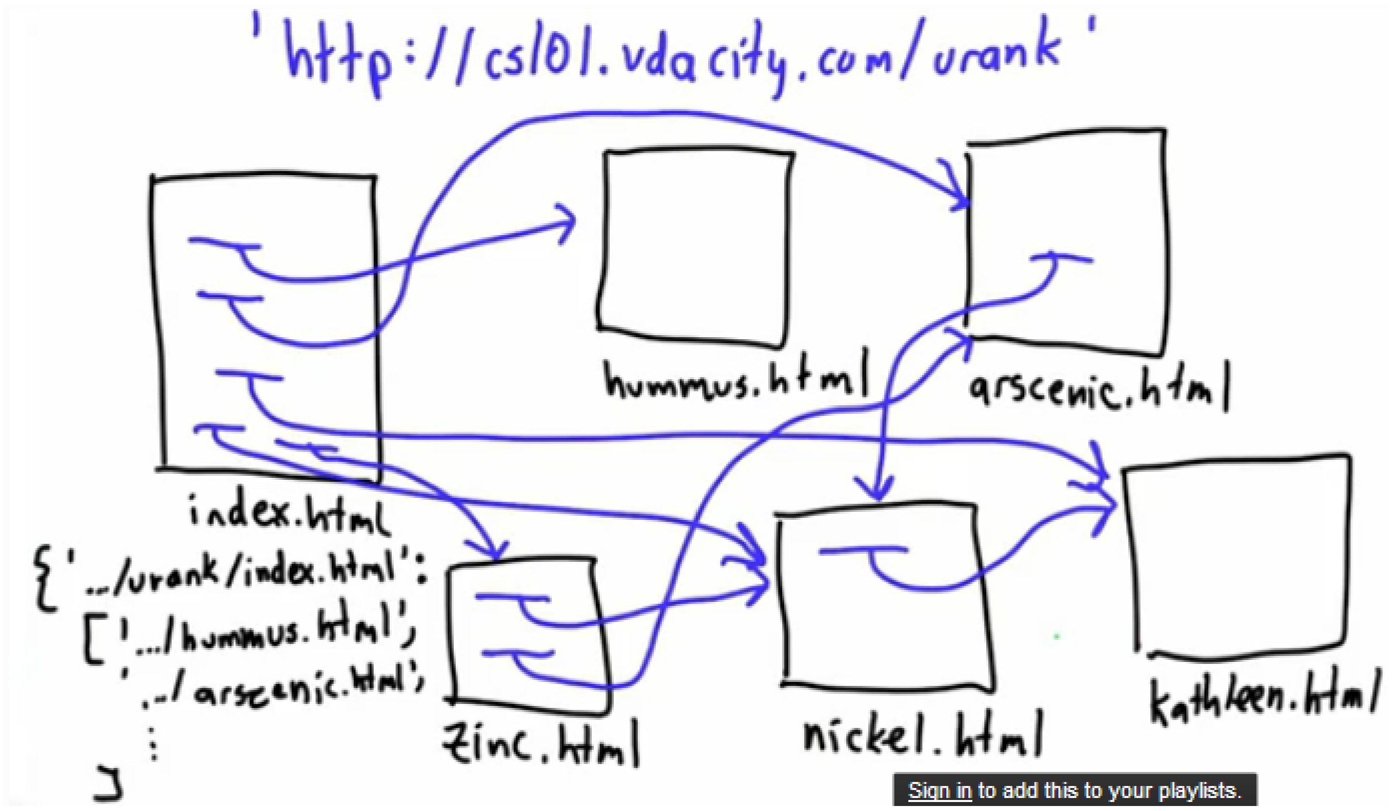
See subsection 6.14 for our description for what we want from our PageRank:

```
#Finishing the page ranking algorithm.

def compute_ranks(graph):
    d = 0.8 # damping factor
    numloops = 10

    ranks = {}
    npages = len(graph)
    # Defining ranks as we said in AltaVista subsection.
    for page in graph:
        ranks[page] = 1.0 / npages

    for i in range(0, numloops):
        newranks = {}
        for page in graph:
            newrank = (1 - d) / npages
            # Summation as in AltaVista subsection.
            for node in graph:
                if page in graph[node]:
                    newrank = newrank + d * (ranks[node] / len(graph[node]))
            newranks[page] = newrank
        ranks = newranks
    return ranks
```

Testing our `compute_ranks` code:

```
index, graph = crawl_web('http://udacity.com/cs101x/urank/index.html')
ranks = compute_ranks(graph)
print ranks

#>>> {'http://udacity.com/cs101x/urank/kathleen.html': 0.11661866666666663,
#'http://udacity.com/cs101x/urank/zinc.html': 0.038666666666666655,
#'http://udacity.com/cs101x/urank/hummus.html': 0.038666666666666655,
#'http://udacity.com/cs101x/urank/arsenic.html': 0.054133333333333325,
#'http://udacity.com/cs101x/urank/index.html': 0.033333333333333326,
#'http://udacity.com/cs101x/urank/nickel.html': 0.09743999999999997}
```

Comparing our results to what is expected:



Note. The kathleen.html page is more popular than the nickel.html page.

Remark. Even though the nickel page has **more links** (more friends), the kathleen page is linked to more popular pages.

6.19 Search Engine

We are done!