Computational Thought *with Prof. Dennis Shasha* scribe: George Wong
1 November 2011

We started out class by running a few of the programs that were produced by our classmates for the number guessing problem. It was noted that in the future, we should not be writing programs that perform random operations: rather there should be some algorithm(s) working in the background.

For the guessing problem, the number of guesses 'n' can be found by setting: n = $\log_2$(range). Of course, range can be found by: range = max-min. This is because each time that we guess and figure out whether the number is equal, LESS or GREATER, we can effectively eliminate at least half of the numbers in our range. As an example:

We start with the range [1,1000]. Guessing 500, the mean of the range, and saying that the number is basically less or greater than our guess eliminates either [1,500] or [500-1000]. We are left with either [501,1000] or [1,499] respectively. We find that this range is only half of the size of our first range. Each guess after that subsequently eliminates half of the range as well. Thus, the number of times we need to divide the original range by 2 until we end up with one number is the number of guesses we need. The function $\log_2$ returns this number of divisions.

We saw the picture that can be found at: http://cs.nyu.edu/cs/faculty/shasha/papers/new-york-rangers.jpeg. Our beloved professor apparently has season tickets to the New York Rangers ice-hockey team.

Aux programmes:

The first example of a program that we looked at was *zoo.py*, whose original code can be found at the end of this document. In this, we introduced the concepts of the 'array', in the form of a list of animals at a zoo. The programs original purpose was to allow the user to select one of the listed animals and delete that animal from the list.

We jumped right in and noted that there were two of the 'tiger' entries and sought out what would happen were we to elect to remove 'tiger'. We found that this course of action only removed one of the 'tiger' entries from the list. The *.remove()* function searches through an array and removes the first instance of the argument within the parenthesis that it finds.

To modify the program, we made a quick, slight modification: *if* to *while* (with the removal of *else*, because *else* makes no sense for *while* loops). Thus we ended up with the program *zoonat.py*, also reproduced below. Running this program removed (and continues to remove) every instance of indicated animal from the array.

Note that this works because the *while* loop keeps running until the animal we specified, *new_animal* is no longer *in zoo*. Consider the corresponding physical case: we have a zookeeper whom we tell to remove all of the tigers. He protests, saying that he can only handle one tiger at a time. We revise, saying, 'While there is a tiger in the zoo, remove a tiger.'

We decided to modify the program one last time: for each animal removed, we wish to add an elephant. To do this, we simply added a line to the *while* loop, indicating that 'elephant' should be added. Note that this function comes in the form *.append()*, where the argument 'elephant' is written between the two parenthesis. The code of this program, named *zooravi.py*, is also reproduced below.

Breaking away from the family of *zoo* programs, we entered the world of file manipulation.

Our first foray introduced us to the concept of *open()*, as a way to read the contents of a file. The exact syntax used was *open("ourownfile","r")*, wherein the first term of the argument specifies the name of the file to be opened (and for now, this file needs to be in our working directory); the second term indicates that the file is only going to be read (as opposed to being written to).

Notice that we set this *open()* equal to a term, which we name 'fileicareabout'. From acts as an easier way to refer to the contents of the file; however, it cannot yet truly be manipulated, so we thus port all of the information stored within it to a quasi-string 'text'. For posterity reasons (and to avoid nasty worm-like issues), we close the file we opened with the code *fileicareabout.close()*.

To actually show the user what was in the file, we use a simple *for* loop. Our quasi-string 'text' has different lines (that correspond to the lines of the original file). Within our *for* loop, we count the number of characters that the line contains (by use of the *len()* function), print that value, and then print the line itself. Remember that what we actually call each of the lines in the large 'text' file does not matter. Here we started with the name 'line' and later swapped to 'george' to indicate this variability.

When we first ran the program, something interesting happened. After each line, the program skipped a line when printing the text. This happened because of the way that information is stored in files, specifically information regarding changing from one line to the next. At the end of each of the lines in the file, there is the character '\n' or CR… *carriage return*, indicating that we should switch to a new line.

A simple 'fix' or work-around for this simply has us writing each of the lines in the file, minus the last character. To do this, we add the range *[0:len(name)-1]* to the end of the *print* command, directly after our line/string (which we here have named george). Were we to swap out 'line' for 'george' again, we would end up with *print len(line), line[0:(len(line))-1)]*. The second part of that code essentially tells the computer 'read the string *line* from its start position *0* to one character before its end position, or it length *len(line)-1*.

The question was posed, "What if we specific a range and the line we're asking for is not that long?" Here we came across the fact that the range is not inclusive: if we give the program the range [0:5] to write, it will only write the 0,1,2,3, and 4 characters, not the fifth. (We should note that this exclusiveness enables us to use the *len()* function to signify the end-character of a string. Of course, when we count, we start with one, though the string's characters are indexed starting with the number zero. It therefore stands to reason that the *len()* function should return the a value one more than the highest index of the string whose length it is computing. Again, the non-inclusiveness allows us to simply write *len()* as opposed to *len()-1* when signifying a range.)

In any case, we found out that asking for the range of characters [0,5] when there are only two characters will not cause a problem. The program will simply read the first two, notice there are no more and go on to the next line.

Finally, we got into the heavy stuff, querying Google. It was indicated to us that we did not really need to understand the intimate workings of the code, but to be somewhat familiar with the main concepts that it relies upon.

For next time, we should be prepared with the next assigned Doctor Ecco problem: *Puzzle-Mad Kidnapper*; run the *googlesearcher.py* program a few times to familiarize ourselves with it; and read the *Gerald Sussman: Building a Billion Biocomputers* chapter if we have time.

---

zoo.py

```
#!/usr/bin/env python
#
# Deleting elements from a list

zoo = ["monkey", "tiger", "eagle", "parrot", "tiger"]
print "The zoo has the following", len(zoo), "animals:", zoo

new_animal = raw_input("Which animal would you like to delete? \
Input the name of the animal: ")
if new_animal in zoo:
      zoo.remove(new_animal)
      print new_animal, "has been deleted"/
else:
      print new_animal, "cannot be deleted because it is not in the zoo"
print "The zoo has the following", len(zoo), "animals:", zoo
```

zoonat.py

```
#!/usr/bin/env python
#*
# Deleting elements from a list

zoo = ["monkey", "tiger", "eagle", "parrot", "tiger"]
print "The zoo has the following", len(zoo), "animals:", zoo

new_animal = raw_input("Which animal would you like to delete? \
Input the name of the animal: ")
while new_animal in zoo:
      zoo.remove(new_animal)
      print new_animal, "has been deleted"

print "The zoo has the following", len(zoo), "animals:", zoo
```

zooravi.py

```
#!/usr/bin/env python
#
# Deleting elements from a list

zoo = ["monkey", "tiger", "eagle", "parrot", "tiger"]
print "The zoo has the following", len(zoo), "animals:", zoo

new_animal = raw_input("Which animal would you like to delete? \
Input the name of the animal: ")
while new_animal in zoo:
      zoo.remove(new_animal)
      zoo.append('elephant')
      print new_animal, "has been deleted"

print "The zoo has the following", len(zoo), "animals:", zoo
```

filein.py

```
#!/usr/bin/env python
#
# Program to read and print the length and
# every line of a file
#
fileicareabout = open("ourownfile","r")
text = fileicareabout.readlines()
fileicareabout.close()


for george in text:
    print len(george), george[0:(len(george)-1)]
    print len(george), george[0:4]
print
```

googlesearcher.py

```
import urllib, urllib2
import json
import sys

print sys.argv

api_key, userip = None, None
query = {'q' : 'great climbing in new york'}

# by default, you learn about python, but if you give an argument, you learn
# about other stuff.
if 1 < len(sys.argv):
  query['q'] = sys.argv[1]
referrer = "http://cs.nyu.edu/q/3900610"
```

```
if userip:
    query.update(userip=userip)
if api_key:
    query.update(key=api_key)

url = 'http://ajax.googleapis.com/ajax/services/search/web?v=1.0&%s' % (
    urllib.urlencode(query))

request = urllib2.Request(url, headers=dict(Referer=referrer))
json = json.load(urllib2.urlopen(request))

results = json['responseData']['results']
for r in results:
  print r['title'] + ": " + r['url']
  print "----------"
```