loops continued and coding efficiently

Genome 559: Introduction to Statistical and Computational Genomics

Prof. James H. Thomas

Review

Increment operator

```
x += y      # adds y to the value of x
x *= y      # multiplies x by the value y
x -= y      # subtracts y from the value of x
```

Explicit program exit

```
sys.exit()   # exit program immediately
```

Use to terminate when something is wrong - best to use `print` to provide user feedback before exit

Smart loop use

- if you don't know how many times you want to loop, use a **`while`** loop (indeterminate).

- e.g. finding all matches in a sequence

- e.g. looping through a list until you reach some list value

```
lastVal = "Gotterdammerung"
i = 0
while i < len(someList) and someList[i] != lastVal:
    <do something with someList[i]>
    i += 1
if i == len(someList):
    print lastVal, "not found"
    print "World not ended yet"
```

Smart loop use

Read a file and print the first ten lines

```
import sys
infile = open(sys.argv[1], "r")
lineList = infile.readlines()
counter = 0
for line in lineList:
    counter += 1
    if (counter > 10):
        break
    print line
infile.close()
```

Does this work?

YES

Is it good code?

NO

What if the file has a million lines? (not uncommon in bioinformatics)

```
import sys
infile = open(sys.argv[1], "r")
lineList = infile.readlines()
counter = 0
for line in lineList:
    counter += 1
    if (counter > 10):
        break
    print line
infile.close()
```

this statement reads all million lines!!

How about this instead?

```
import sys
infile = open(sys.argv[1], "r")
for counter in range(10):
    print infile.readline()
infile.close()
```

this version reads only the first ten lines, one at a time

This while loop does the same thing:

```
import sys
infile = open(sys.argv[1], "r")
counter = 0
while counter < 10:
    print infile.readline()
    counter += 1
infile.close()
```

- The original `readlines()` approach not only takes much longer on large files it also has to store ALL the data in memory.

- I ran original version and efficient version on a very large file.

- Original version ran for 45 seconds and crashed when it ran out of memory.

- Improved version ran successfully in << 1 sec.

What if the file has fewer than ten lines?

```
import sys
infile = open(sys.argv[1], "r")
for counter in range(10):
    print infile.readline()
infile.close()
```

hint - when `readline()` reaches the end of a file, it returns `""`
but a blank line in the middle of a file returns `"/n"`

It prints blank lines repeatedly - not ideal

Improved version:

```
import sys
infile = open(sys.argv[1], "r")
for counter in range(10):
    line = infile.readline()
    if len(line) == 0:
        break
    print line
infile.close()
```

test for end of file

Memory allocation efficiency

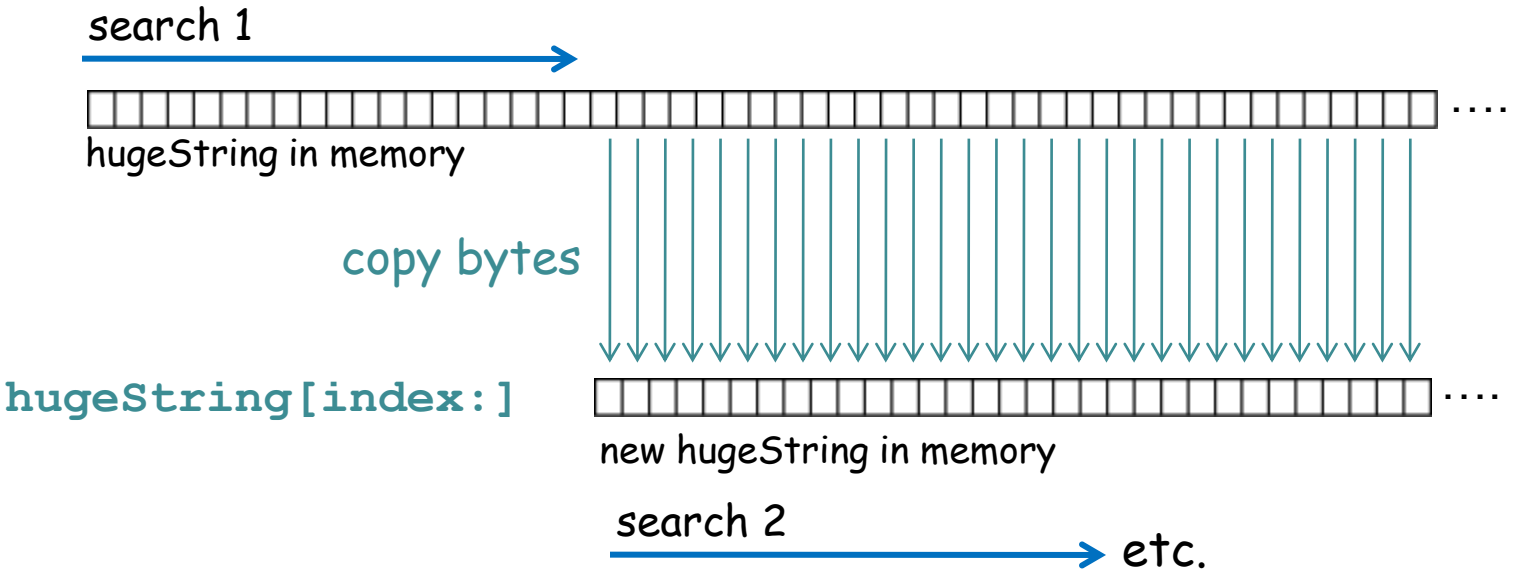
```
index = 0
curIndex = 0
while True:
    curIndex = hugeString[index:].find(query)
    if curIndex == -1:
        break
    index += curIndex
    print index
    index += 1    # move past last match
```

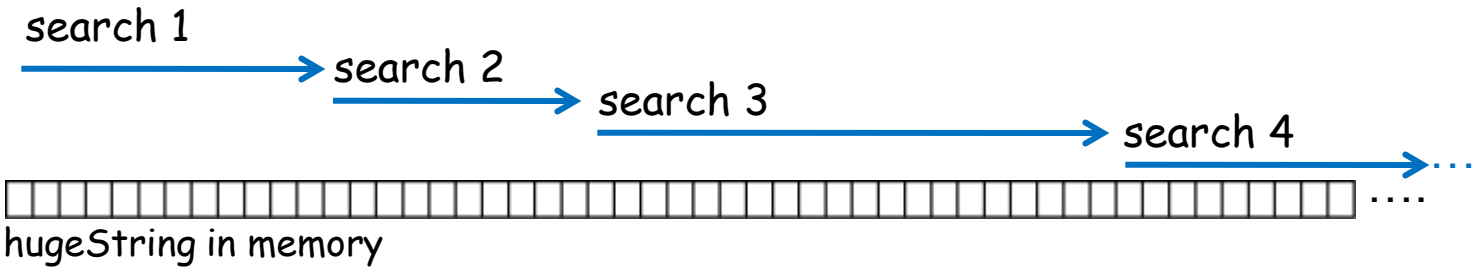
be wary if you are splitting strings a lot

```
index = 0
while True:
    index = hugeString.find(query, index)
    if index == -1:
        break
    print index
    index += 1    # move past last match
```

First version makes a NEW large string in memory every time through the loop - slow!

Second version uses the same string every time but starts search at different points in memory. Ran 10x to 1000x faster in test searches.





`hugeString.find(queryString, index)`

To figure out where to start this search, the computer just adds `index` to the position in memory of the 0th byte of hugeString and starts the search there.

Sequential splitting of file contents

Many problems in text or sequence parsing can employ this strategy:

- First, split file content into chunks (lines or fasta sequences etc.)
- Second, from each chunk extract the needed data
- This can be repeated - split each chunk into subchunks, extract needed data from subchunks.

```
import sys
lineList = open(sys.argv[1], "r").readlines()
for line in lineList:
    fieldList = line.strip().split("\t")
    for field in fieldList:
        <do something with field>
```

How many levels of splitting does this do? 2

General points for improving your code

- Write compactly as long as it is clear to read

- Consider whether you do things that are unnecessary (e.g. reading all the lines in a file when you don't need to)

- Consider user feedback if something unexpected arises (we will learn how to do this more elegantly soon).

- Don't waste memory by keeping information you don't need to use.

Sample problem #1

Write a program `read-N-lines.py` that prints the first N lines from a file (or all lines if the file has fewer than N lines), where N is the first argument and filename is the second argument. Be sure it handles very short and very long files correctly and efficiently.

```
>python read-N-lines.py 7 file.txt
this
file
has
five
lines

>
```

Solution #1

```
import sys
infile = open(sys.argv[2], "r")
max = int(sys.argv[1])
counter = 0
while counter < max:
    line = infile.readline()
    if len(line) == 0:   # we reached end of file
        break
    print line.strip()
    counter += 1
```

Sample problem #2

Write a program **`find-match.py`** that prints all the lines from the file **`cf_repmask.txt`** (linked from the web site) in which the 11th text field exactly matches "CfERV1", with the number of lines matched and the total number of file lines at the end. Make the file name, the search term, and the field number command-line arguments.

The file is an annotation of all the repeat sequences known in the dog genome. It is ~4.5 million lines long. Each line has 17 tab-delimited text fields.

You will know you got it right if the example match count is 1,168.

(If you use the smaller file cfam_repmask2.txt, the count should be 184)

Your program should run in about 10-20 seconds.

Solution #2

```
import sys
if (len(sys.argv) != 4):
    print("USAGE: three arguments expected")
    sys.exit()
query = sys.argv[1]                 # get the search term
fnum = int(sys.argv[2]) - 1         # get the field number
hitCount = 0                  # initialize hit and line counts
lineCount = 0
f = open(sys.argv[3])               # open the file
for line in f:                      # for each line in file
    lineCount += 1
    fields = line.split('\t')
    if fields[fnum] == query:       # test for match
        print line.strip()
        hitCount += 1
f.close()
print hitCount, "matches,", lineCount, "lines"
```

Remark - in Solution #2 it is a bad idea to read all the lines at once with `f.readlines()`.

Even though the problem requires you to read every line in the file, the best solution uses minimal memory because it never stores more than one line at a time.

Challenge problem 1

Extend sample problem 2 so that there is an optional 4th argument that specifies a minimum genomic length to report a match.

In the file, fields 7 and 8 are integers that indicate the genomic start and end positions of the repeat sequence.

You should get 341 matches for the query "CfERV1" and a minimum genomic length of 1000. (If you use the smaller file cfam_repmask2.txt, there should be 63 matches)

Solution to challenge problem 1

```
import sys
if (len(sys.argv) < 4):
    print("USAGE: at least three arguments expected")
    sys.exit()
query = sys.argv[1]
fnum = int(sys.argv[2]) - 1
minSpan = 0                 # set a default so that any match passes
if len(sys.argv) == 5:
    minSpan = int(sys.argv[4])
hitCount = 0
lineCount = 0
f = open(sys.argv[3])
for line in f:
    lineCount += 1
    fields = line.split('\t')
    if fields[fnum] == query:
        span = int(fields[7]) - int(fields[6])
        if span >= minSpan:
            print line.strip()
            hitCount += 1
f.close()
print hitCount, "matches,", lineCount, "lines"
```

Challenge problem 2

Modify sample problem 2 so that the number of matches and number of lines prints BEFORE the specific matches.

The trick is simple - make a list that will hold the matched lines, rather than printing them as you go. Print the list at the end.

```
import sys
if (len(sys.argv) != 4):
    print("USAGE: three arguments expected")
    sys.exit()
query = sys.argv[1]
fnum = int(sys.argv[2]) - 1
lineCount = 0
matchLines = []      # initialize the list to hold match lines
f = open(sys.argv[3])
for line in f:
    lineCount += 1
    fields = line.split('\t')
    if fields[fnum] == query:
        matchLines.append(line.strip())  # put line in list
f.close()
print len(matchLines), "matches,", lineCount, "lines"
for line in matchLines:
    print line
```

(note also that `matchLines` implicitly gives the number of matched lines)

One possible problem is that, if the number of matched lines is huge, you could run out of memory.