loops continued and coding efficiently

Genome 559: Introduction to Statistical and Computational Genomics

Prof. James H. Thomas

Review

`for` vs. `while` loops

- `for` is natural to loop through all elements of something of <u>determinate</u> size.

- `while` is natural to loop an <u>indeterminate</u> number of times until some condition is met.

Mnemonic: Four-D (for determinate), Whindy (while indeterminate)

Smart loop use

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

- loop through a list until you reach some list value

YOUR PLAN? `for` with break, `while` with test

```
queryVal = "Gotterdammerung"
i = 0
while i < len(someList) and someList[i] != queryVal:
    i += 1
print "value at list position", i+1
```

ensures loop doesn't run past end of list

tests whether element matches query

Read files efficiently

Read a file and print the first ten lines

```
import sys
infile = open(sys.argv[1], 'r')
lineList = infile.readlines()
infile.close()
for i in range(10):
    print lineList[i].strip()
```

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()
infile.close()
for i in range(10):
    print lineList[i].strip()
```

this statement reads all million lines!!

How about this instead?

```
import sys
infile = open(sys.argv[1], "r")
for i in range(10):
    print infile.readline().strip()
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 takes much longer on large files AND it 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 <u>fewer</u> than ten lines?

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

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

The program above prints blank lines repeatedly - not ideal

Improved version:

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

test for end of file

Keep only needed data

Suppose you want to work with a subset of file lines (say those with a match to some string):

```
lines = openFile.readlines()
for line in lines:
   if "Elizabeth Bennet" in line:
      <do something>
```

OR

```
for line in openFile:
   if "Elizabeth Bennet" in line:
      <do something>
```

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
```

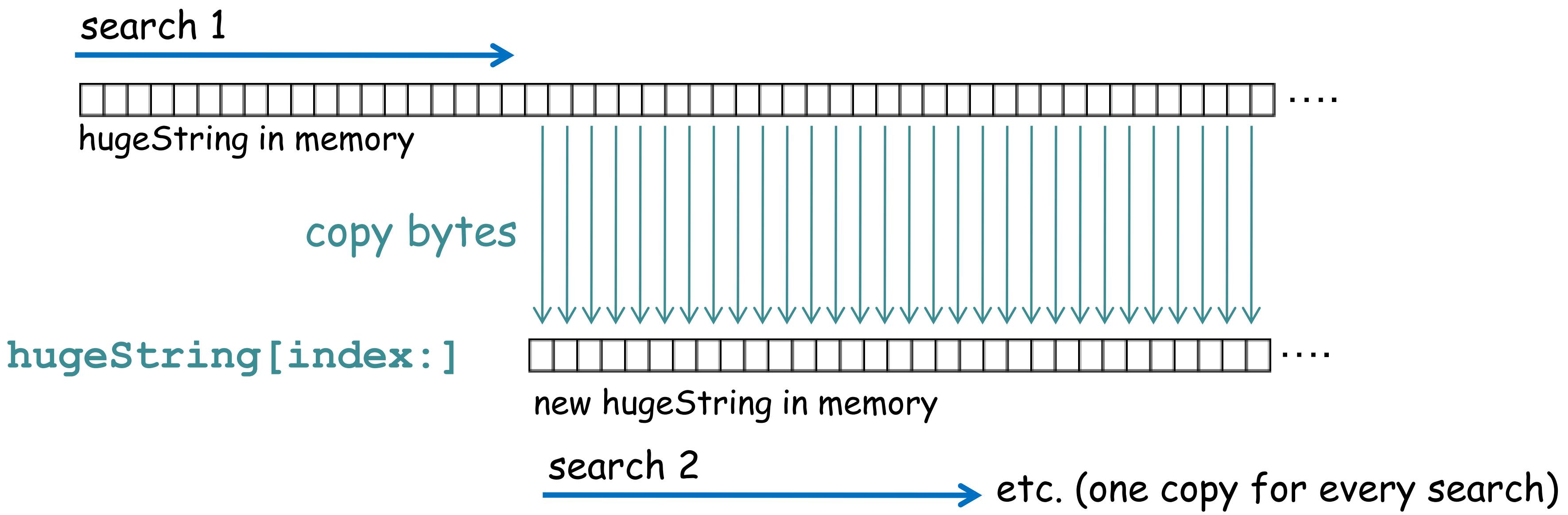
when query is not found, find() returns -1

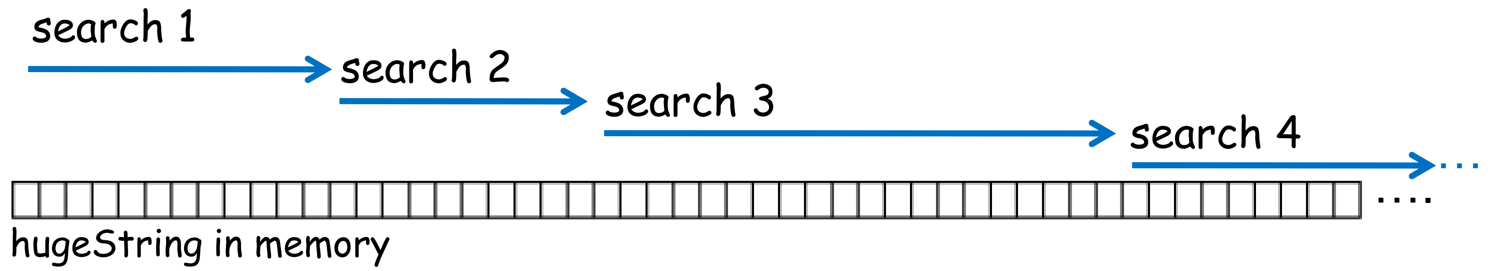
be wary if you are splitting strings a lot

```
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 - <u>very very slow</u>!

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(query, 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 - very fast.

- This sort of issue arises in many contexts, not just string slicing
- Always be aware of what your code is doing with memory

N.B. review later

Same find method, avoiding `while True`

```
index = hugeString.find(query)
while index != -1:
    print index
    index += 1  # start just after last match
    index = hugeString.find(query, index)
```

Sequential splitting of file contents

Many problems in text parsing can employ this strategy:

- First, read file content in chunks (often lines)

- Second, split each chunk to extract the needed data

- This can be repeated - split each chunk into subchunks, extract needed data from subchunks.

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

shorthand way to read each line in a file

How many levels of splitting does this do? 2

General tips for improving your code

- Write <u>compact code</u> as long as it is clear to read.

- Consider whether you do things that are <u>unnecessary</u>.

- If you write code that seems clunky, think carefully about how to make it nice and clean (fewer lines, clearer flow, etc.).

- <u>Don't waste memory</u> by keeping information you don't need.

- When reading from files, read (and store in memory) <u>only what you need</u>.

- Persistent storage (e.g. hard drive) is large but slow to read and write. Volatile storage (RAM or just "memory") is smaller but fast.

NB solid-state drives are faster but still pretty slow

Sample problem #1

Write a program `read-lines.py` that prints the Ith through Jth lines from a file, where I, J, and filename are command-line arguments. Be sure it handles very short and very long files correctly and efficiently.

```
> python read-N-lines.py 1 7 file.txt
this
file
has
five
lines
> python read-N-lines.py 2 3 file.txt
file
has
>
```

Solution #1

```
import sys
start = int(sys.argv[1]) - 1  # change to zero-based
end = int(sys.argv[2])
infile = open(sys.argv[3], "r")
counter = 0
while counter <= start:
    counter += 1
    line = infile.readline()
while counter <= end and line != "":
    print line.strip()
    line = infile.readline()
    counter += 1
infile.close()
```

It is tempting to read all the lines and use a for loop because it is simpler, but <u>not</u> a good design if file might be large.

Solution #1 alt

```
import sys
start = int(sys.argv[1]) - 1  # change to zero-based
end = int(sys.argv[2])
infile = open(sys.argv[3], "r")
counter = 0                  # count lines read
for line in infile:   # read every line but only print those wanted
   if counter >= end:        # terminate when past end
      break
   if counter >= start:      # only print when past start
      print line.strip()
   counter += 1              # increment counter every line
infile.close()
```

Notice that this also reads lines one at a time starting at the beginning of the file. It "throws away" lines until `counter` is in range and breaks the loop when `counter` is beyond range. This is functionally identical to the other solution.

Sample problem #2

Write a program `find-match.py` that prints all the lines from the file `cfam_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. (11th is one-based counting)

[The file is an annotation of all the repeat sequences known in the dog genome. It is 4,533,479 lines long. Each line has 17 tab-delimited text fields.]

You will know you got it right if the "CfERV1" match count is 1,168. (If you use the smaller file cfam_repmask2.txt, the count should be 184.) With the large file, your program should run in about 10-20 seconds (500K – 1M lines per second!).

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
infile = open(sys.argv[3])           # open the file
for line in infile:                  # for each line in file
    lineCount += 1
    fields = line.split('\t')
    if fields[fnum] == query:        # test for match
        print line.strip()
        hitCount += 1
infile.close()
print hitCount, "matches,", lineCount, "lines"
```

Remark - in Problem #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 <u>minimal memory</u> because it never <u>stores</u> 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 repeat 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 11th field 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:       # get the optional min length
    minSpan = int(sys.argv[4])
hitCount = 0
lineCount = 0
of = open(sys.argv[3])
for line in of:
    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
of.close()
print hitCount, "matches,", lineCount, "lines"
```

Challenge problem 2 etc

Modify sample problem 2 so that the number of matches and number of lines prints BEFORE the specific matches (often useful for the user, because the output has a summary first, followed by specifics). Solution on next slide.

Modify sample problem 2 so that you won't get an error if the number of fields on a line is too small (e.g. suppose you are testing field 9 but there is a blank line somewhere in the file).

Modify again so that you report lines in the file that don't have enough fields to match your request. Report the line number (e.g. line 39 etc), which will help a user edit a file that has mistakes in it.

Write a program that tests that every line in a file has some expected number of fields and reports those that don't. Modify the program so you remove those that don't rather than reporting them.

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
of = open(sys.argv[3])
for line in of:
    lineCount += 1
    fields = line.split('\t')
    if fields[fnum] == query:
        matchLines.append(line.strip())  # put line in list
of.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.