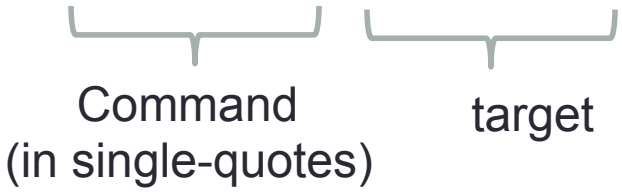
Today's Class

- Answers to AWK problems
- Shell-Programming
  - Using loops to automate tasks
- Future:
  - Download and Install:
    - Python (Windows only.)
    - R

Awk basics

- From the command line:

```
$ awk '$1>20' filename
```



- Awk commands have the following syntax:

```
condition { action }
```

  - If no action is specified, prints the entire line
  - $1,$2, $3… refer to column1, column2, column3, etc.
  - $0 refers to the entire line

```
$ awk '$1>20 {print $5} ' filename
```

(for all lines where column 1 > 20, print column 5)

Awk basics

- The ‘condition’ can be a pattern-match *(aka regular expression)*:

```
awk '$2~/act/ && $4 > 30' dd_dev_transcriptome.txt
```

(prints all lines where column 2 matches “act” and column 4 is greater than 30)

- You can perform substitutions:

```
awk '{gsub(/DDB_G0/, "ddb0");print}' dd_dev_transcriptome.txt
```

(replaces DDB_G0 with ddb0)

- You can do calculations:

```
awk '{sum+=$3} END {print sum/NR}' dd_dev_transcriptome.txt
```

(calculates mean of column 3)

NF = Number of Fields (=columns)
NR = Number of Records (=rows)

BEGIN = Do prior to reading file
END = After reading file

Awk basics

- Combine with UNIX commands

```
awk '$3>30'filename | sort -n -k4 | tail -10 > outfile
```

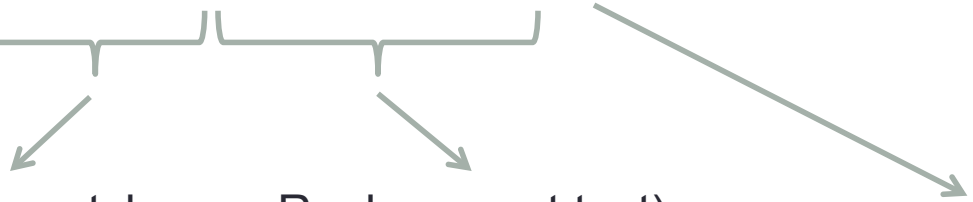
(Extract all rows where column 3 is greater than 30, sort them by column 4, take the 10 highest values, and save the results to outfile)

Sed (stream editor)

- UNIX utility that parses and transforms text
- Developed at Bell Labs
  - One of the first tools for regular expressions (aka pattern matching)
  - grep, sed and AWK developed from “ed” (early editor) and were the inspiration for Perl
    - All of these are notable for their editing of strings
      - Influenced the syntax

Example sed command - substitute

```
$ sed 's/regexp/replacement/g' inFileName > outFileName
```



Text to match (exact or regular expression)

Replacement text)

If nothing: defaults to first Instance

g: global (all matches)
1: first match
2: second match
Etc

Try this example:

```
sed 's/DDB_G/ddb_g/g' dd_dev_transcriptome.txt > dd_dev_transcriptome.txt2
```

References on Sed

http://www.panix.com/~elflord/unix/sed.html

http://www.catonmat.net/blog/worlds-best-introduction-to-sed/

https://www.digitalocean.com/community/tutorials/the-basics-of-using-the-sed-stream-editor-to-manipulate-text-in-linux

Filename “globbing” (wildcards)

“refers to pattern matching based on wildcard characters”
- Wikipedia

See Wikipedia “glob”

Two most common are * and ?

```
$ ls *.dat
```

…Prints all files that end in “.dat”

```
$ ls tgr_?.dat
```

…Matches tgr_1.dat but not tgr_10.dat

```
$ ls file[12].txt
```

…Matches file1.txt and file2.txt

Shell Programming

- Shell is the interface between the user and the computer
- Shell is a program that runs other programs (e.g., ls, cp, grep)
- Shell is also a programming language!

    - Actually, many programming languages…
      sh (Original “Bourne” shell; UNIX)
      csh (C-shell, implemented syntax of C programming language)
      bash (Bourne Again Shell - GNU)
      tcsh (Variant of csh)
      … and many more

- A shell program is automatically started when you login (“login shell”)
  --default is usually bash

- You can enter another shell by typing its name at the prompt
- Different shells have different syntax

Two Ways to do Shell-Programming

1. Interactively, entering commands on the command-line

2. Use a text editor to write a shell-script and run it at the command line

   Example: `$ bash my_script.sh`

   If you don't want to write "bash", you can do the following:

   (1) Add this line to the top of the script: `#! /bin/bash`
   (2) Change permissions to make the file executable (owner)

   Now try it:
   `$ my_script.sh` (no longer need to use **bash** command)

   (Note: This works on other languages, too – e.g., #! /usr/bin/ruby)

Using a **for loop** to do a repetitive task

Example 1: Concatenate output together

(In directory: /home/b/bio6297eo13/files_for_students/compute_in_batches)

Note the change in the prompt!

```
$ touch compute_ALL
$ for i in {1..14}
> do
> less compute_all_$i >> compute_ALL
> done
```

As a one-liner:

```
$ touch compute_ALL
$ for i in {1..14}; do less compute_all_$i >> compute_ALL; done
```

Q: How can you ensure the header of each file is only printed in the final file once (at the top)?

Using a **for loop** to do a repetitive task

```
for each_item in list_of_items; do
    this_action
done
```

Semi-colons can usually be used be interchangeably with returns – e.g.:

```
for each_item in list_of_items
do this_action
done
```

…is equivalent to:

```
for each_item in list_of_items; do this_action; done
```

Control Flow Statements

- The “for” loop is an example of a control flow statement
- Control flow statements change the execution of commands, usually making them conditional in some way
- Three most common loops: for, if, while

Bash “if” loop:

```
if [ condition ]
then
    statement1
else
    statement 2
fi
```

Note:
Kewords to start the loop
Keywords (“end”) to end the loop

Indentation is used to indicate the level

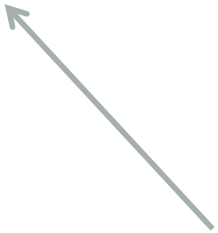
Makes it easy to see the flow and to ensure your loops have been closed.

Bash "if" loop:

```
if [ condition ]
then
    statement1
else
    statement 2
fi
```

Example conditions:

```
if [ $x -le 6 ]
```



le = "less than or equal to"

Bash "while" loop:

```
while [ condition ]
do
   command1
   command2
   command3
done
```

Must have spaces within [ ]:

```
[ $x -le 6 ]
```

not

```
[$x -le 6]
```

Using a **for loop** to do a repetitive task

Example 2: Copy all files that match a pattern (="tgr") to a new directory:

(In directory: /home/h/hpc13f52/example_files/alignments)

From the command line:

```
$ mkdir tgr_aligns
> for i in ./*tgr*; do
> cp $i tgr_aligns
> done
```

Note the change in the prompt!

As a script:

```
mkdir tgr_aligns
for i in ./*tgr*
do
    cp $i tgr_aligns
done
```

A one-liner to find which files contain error messages:

(from directory: /home/b/bio6297/files_for_students/):

```
$ for i in tgr_?.dat; do echo $i; less $i | grep Error; done
```

(prints out all tgr results files with Error messages)

*Note: Commands can be separated with a ⏎ **(return)** or with a **; (semi-colon)***

Loops can be nested within other loops

```
for statement; do
if condition ; then
statement2
else
while condition do;
do_this_other_thing
done
fi
done
```

Deeply nested loops can be difficult to follow…

Indentation & syntax highlighting make it easier to see the flow of execution

```
for statement; do
      if condition ; then
            statement2
      else
            while condition do;
                  do_this_other_thing
            done
     fi
done
```

Same as before, but with indentation and highlighting

All programming languages have the same basic loops (for, while, and if)

E.g., All the programs below produce the same result:

Ruby

```
for i in 1..5
    puts “Value is #{i}”
end
```

bash

```
for i in {1..5}
do
    echo Value is $i
done
```

csh:

```
foreach i (1 2 3 4 5)
    echo Value is $i
end
```

Python

```
for i in range(1,6):
    print "Value is %d" % (i)
```

R

```
for (i in 1:5){
    cat("Value is", i, "\n")
}
```

C++ **(*note this a snippet of a larger program)**

```
 for(int i=1;i<6;i++) {
    cout << "Value is " << i << endl;
 }
```

Q: Can you write a **bash** script that accomplishes the same thing with while loop instead?

Standard in, standard out, standard error

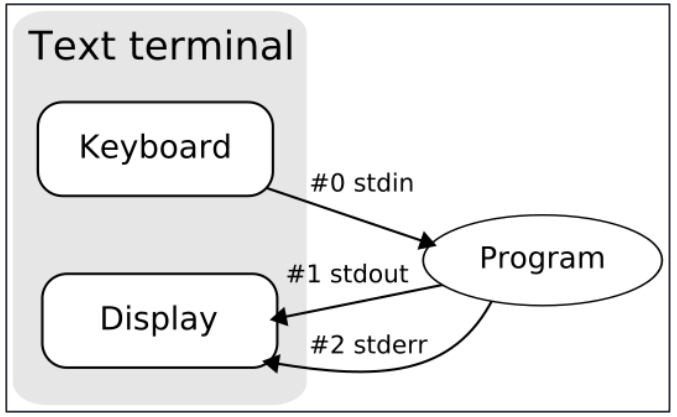
All programs have input/output (I/O) streams:

- Take input → “standard in”, “stdin”
- Results from the program printed out -> “standard out”, “stdout”
- Error / Warning Messages → “standard error”, or “stderr”

By default:

- stdin is the keyboard
- stdout and stderr is the terminal (prints to screen)

- stdout and stderr can be redirected to files



Source: Wikipedia

Re-directing stdout or stderr to file:

```
python test_stdout_stderr.py 1>so 2>se
```

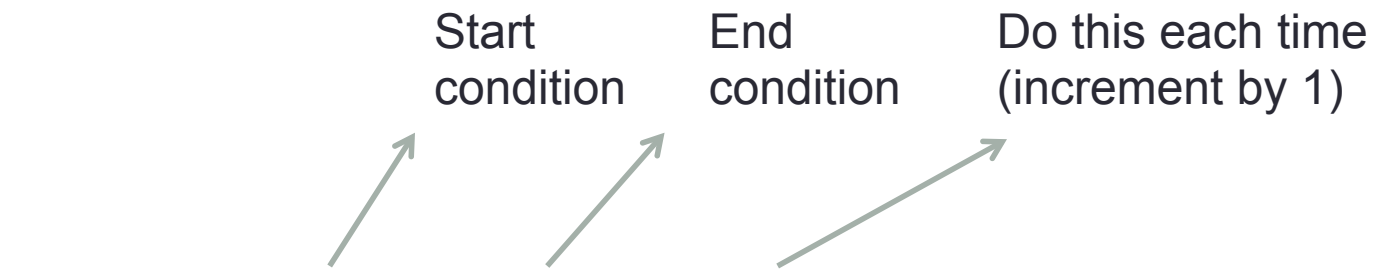
‘`&> filename`’ (redirects stderr and stdout to same file)

‘`2>&1`’ redirects stderr to whatever stdout was being directed to
– e.g., python test_so_se.py >out 2>&1

(1 refers to stdout, 2 refers to stderr, “>” is the redirect symbol, and “&” indicates file descriptor)

Alternative “for” loop syntax (C-style)

Similar to syntax of C programming language:




```
for (( i=0; i<10; i++ )); do
      echo $i
done
```

Command-line arguments

- Set variables on the command-line
- Allow user to change how the program (e.g., alter parameters) without having to edit the code itself.

- `$ bash my_script.sh [arg1] [arg2] .. [arg n]`

Inside your script:

```
echo $1
touch $2
(etc…)
```

Will be replaced with whatever you specified on the command-line

It is usually best to rename these variables inside your script to something more intuitive – e.g.:

```
indir = $1
outdir = $2
```

Monday's Quiz

- Using AWK to filter output

- Bash Programming
  - Command-line entry versus a script
  - How to make a script executable (in any language – bash, Python, etc.)
  - Loops (simple!)

Some Useful References on BASH:

http://www.panix.com/~elflord/unix/bash-tute.html
http://tldp.org/LDP/Bash-Beginners-Guide/html/