STAT 605
Data Science Computing
Introduction to `sed` and `awk`
Editing text streams: **sed**

**sed** is short for **stream editor**

One of the most powerful and versatile UNIX tools

Commonly paired with **awk**
small command line language for string processing

Has lots of features, but we’ll focus on one: **substitutions**

```
keith:~$ echo "hello world" | sed 's/hello/goodbye/g'
goodbye world
```

**s** for substitute

Replace this...

...with this.

**g** for globally, meaning everywhere in the input.
Editing text streams: **sed**

**sed** commands can include regular expressions.

```
keith:~$ echo "a aa aaa" | sed 's/a*/b/g'
b b b
```

'*' works like in **egrep**
Editing text streams: **sed**

**sed** commands can include regular expressions

```
keith:~$ echo "a aa aaa" | sed 's/a*/b/g'
b b b
```

`'*' Works like in egrep`

**Test your understanding:** is the **sed** regex matcher greedy?
Editing text streams: **sed**

**sed** commands can include regular expressions

```bash
keith:~$ echo "a aa aaa" | sed 's/a*/b/g'
b b b
```

`*` Works like in `egrep`

**Test your understanding:** is the **sed** `*` operator greedy?

**Answer:** yes! If it were lazy, above would output just a mess of `'b`'s
Editing text streams: **sed**

**sed** commands can include regular expressions.

```bash
keith:~$ echo "a aa aaa" | sed 's/a*/b/g'
b b b
```

`'*' Works like in *egrep*

**Test your understanding:** is the **sed** * operator greedy?

**Answer:** yes! If it were lazy, above would output just a mess of 'b's

As promised, most of your knowledge of regexes in *egrep* will transfer directly to **sed**, as well as other tools (e.g., *vim*, *emacs*, *Python* and *perl*)
Editing text streams: `sed`

`sed` commands can include regular expressions.

```bash
keith:~$ echo "a aa aaa" | sed 's/a*/b/g'
b b b
```

`'*'` Works like in `egrep`

Basic syntax of `sed` commands:
```
```sed 's/regexp/replacement/flags'
```

```bash
keith:~$ echo "a aa aaa" | sed -E 's/a+/b/g'
b b b
```

To use “extended” regexes, need to give `-E` flag (there is no `esed`, unfortunately).
Editing text streams: **sed**

Basic syntax of sed `s` commands:
```
```sed 's/regexp/replacement/flags'
```

Can use any single character in place of `/`. Special characters have to be escaped.

Of course, we’re only barely scratching the surface:
Quick and dirty text processing: awk

`awk` is a command-line program that runs its own programming language, `AWK`.

Like `grep` and `sed`, `awk` operates on a data stream, read from its `stdin`.
Primarily designed for text processing.

`awk` is a **data driven** programming language.

“Describe what pattern to look for, and what to do when you find it.”

In contrast to **procedural** programming languages (e.g., R and Python).

Basic *awk*: patterns and actions

Basic *awk* program: series of (pattern, action) pairs.

`awk` reads its input one line at a time
When input matches a pattern, perform its associated action

```
pattern { action }
pattern { action }
...
```

Written on separate lines, by convention, though this isn’t required

*Succinctly summarized by A. V. Aho (the A in AWK):*
AWK reads the input a line at a time. A line is scanned for each pattern in the program, and for each pattern that matches, the associated action is executed.
Running awk on the command line

Write a short program, run it with input(s) read from files given on command line.

```
keith:~$ awk 'program' input-file1 input-file2 ...
keith:~$
keith:~$ awk -f program-file input-file1 input-file2 ...
keith:~$
keith:~$ cat input-file | awk -f program-file
keith:~$
```

When running longer programs, it's easier to write our program in a file and read it into `awk`.

We can also have `awk` operate on its stdin, instead. This is, in my experience, the most common way of invoking `awk`. 
Our first `awk` programs

The `BEGIN` pattern tells `awk` to run this command before doing anything with its input (of which there is none).

We’ve written the same program, but now it is stored in `print.awk`.

`awk` applies its (condition,action) pairs to every line of input. In this case, we are just printing every line of input that `awk` sees.

A line with no condition will *always* be executed.

```bash
keith:~$ awk 'BEGIN { print "Hello, world." }'
Hello, world.
keith:~$
keith:~$ echo "This is a string." | awk '{ print }'
This is a string.
keith:~$
```
Comments in awk

# is the comment character in awk (just like bash, R and Python).

keith:~$ cat commented_print.awk
# This program just prints its stdin.
# Not particularly interesting, I'd say.
{ print }

keith@:~/$ echo "dog cat goat bird" | awk -f commented_print.awk
dog cat goat bird

keith:~$ echo "words words words words" | awk '{print} # This is a comment.'
words words words words

keith:~$
**awk built-in variables**

**awk** breaks each line up into fields (i.e., columns), split on whitespace by default.

**awk** has some built-in variables to refer to these fields, similar to bash scripts...

- `$0` : the entire current line
- `$1`, `$2`, `$3`, … : the **field variables**

...and also has some other useful variables (these **do not** require dollar signs):

- `NF` : the number of fields in the current line
- `NR` : the number of records read so far

See documentation for a full list of built-in variables

or see [https://www.gnu.org/software/gawk/manual/gawk.html](https://www.gnu.org/software/gawk/manual/gawk.html)
<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
<th>Email</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amelia</td>
<td>555-5553</td>
<td><a href="mailto:amelia.zodiacusque@gmail.com">amelia.zodiacusque@gmail.com</a></td>
<td>F</td>
</tr>
<tr>
<td>Anthony</td>
<td>555-3412</td>
<td><a href="mailto:anthony.asserturo@hotmail.com">anthony.asserturo@hotmail.com</a></td>
<td>A</td>
</tr>
<tr>
<td>Becky</td>
<td>555-7685</td>
<td><a href="mailto:becky.algebrarum@gmail.com">becky.algebrarum@gmail.com</a></td>
<td>A</td>
</tr>
<tr>
<td>Bill</td>
<td>555-1675</td>
<td><a href="mailto:bill.drowning@hotmail.com">bill.drowning@hotmail.com</a></td>
<td>A</td>
</tr>
<tr>
<td>Broderick</td>
<td>555-0542</td>
<td><a href="mailto:broderick.aliquotiens@yahoo.com">broderick.aliquotiens@yahoo.com</a></td>
<td>R</td>
</tr>
<tr>
<td>Camilla</td>
<td>555-2912</td>
<td><a href="mailto:camilla.infusarum@skynet.be">camilla.infusarum@skynet.be</a></td>
<td>R</td>
</tr>
<tr>
<td>Fabius</td>
<td>555-1234</td>
<td><a href="mailto:fabius.undevicesimus@ucb.edu">fabius.undevicesimus@ucb.edu</a></td>
<td>F</td>
</tr>
<tr>
<td>Julie</td>
<td>555-6699</td>
<td><a href="mailto:julie.perscrutabor@skeeve.com">julie.perscrutabor@skeeve.com</a></td>
<td>F</td>
</tr>
<tr>
<td>Martin</td>
<td>555-6480</td>
<td><a href="mailto:martin.codicibus@hotmail.com">martin.codicibus@hotmail.com</a></td>
<td>A</td>
</tr>
<tr>
<td>Samuel</td>
<td>555-3430</td>
<td><a href="mailto:samuel.lanceolis@shu.edu">samuel.lanceolis@shu.edu</a></td>
<td>A</td>
</tr>
<tr>
<td>Jean-Paul</td>
<td>555-2127</td>
<td><a href="mailto:jeanpaul.campanorum@nyu.edu">jeanpaul.campanorum@nyu.edu</a></td>
<td>R</td>
</tr>
</tbody>
</table>

A: acquaintance  
F: friend  
R: relative
Rules using regexes

We can create rules that apply only to lines matching a regex

```
keith:~$ awk '/\.edu/ { print $0 }' mail-list.txt
Fabius 555-1234 fabius.undevicesimus@ucb.edu F
Samuel 555-3430 samuel.lanceolis@shu.edu A
Jean-Paul 555-2127 jeanpaul.campanorum@nyu.edu R
```

```
keith:~$ awk '/[:space:]]F$/ { print $1, $3 }' mail-list.txt
 Amelia amelia.zodiacusque@gmail.com
 Fabius fabius.undevicesimus@ucb.edu
 Julie julie.perscrutabor@skeeve.com
```

If a line contains the string '.edu', print the whole line.

Print the name and email (fields 1 and 3) of friends. “friend” entries end with a capital F, so that’s what our regex looks for. The comma in the print statement is necessary to put a space between fields 1 and 3.
Comparison patterns

This pattern matches lines whose first field is longer than 6 characters

```
keith:~$ cat mail-list.txt | awk 'length($1) > 6'
Anthony 555-3412 anthony.asserturo@hotmail.com A
Broderick 555-0542 broderick.aliquotiens@yahoo.com R
Camilla 555-2912 camilla.infusarum@skynet.be R
Jean-Paul 555-2127 jeanpaul.campanorum@nyu.edu R
```

We didn’t specify an action. The default is to print the whole line, like `print $0`.

This pattern finds the length of the longest name. Note that we did not have to declare the variable `max`.

```
keith:~$ awk '{ if (length($1) > max) max = length($1) }; END { print max }' mail-list.txt
9
```

The `END` pattern runs once we have reached the end of the input.
Our awk program can include multiple rules. A line can match multiple rules, in which case it gets processed multiple times.

```
keith:~$ awk '/12/ { print $2 }; /21/ { print $2 }' mail-list.txt
555-3412
555-2912
555-1234
555-2127
555-2127
keith:~$
```

```
keith:~$ awk '/12/ && /21/ { print $2 }' mail-list.txt
2127 matches both /12/ and /21/
```

```
keith:~$ awk '/12/ && /21/ { print $2 }' mail-list.txt
555-2127
keith:~$
```

`&&` is the AND operator. A line must match both of these regexes to match the pattern.

What else?

`awk` is a kind of command-line swiss army knife

A non-exhaustive list of things we haven’t discussed:

For- and while-loops
Importing variables from the shell into `awk`
Defining functions in `awk`

The best place to learn more is

*The GNU Awk User’s Guide*


Also recommended:

*sed & awk*, 2nd Edition by D. Dougherty and A. Robbins. O’Reilly Media