

STATS 701

Data Analysis using Python

Lecture 26: Advanced Command Line

Why UNIX/Linux?

As a data scientist, you will spend **most** of your time dealing with data

Data sets never arrive “ready to analyze”

Cleaning data, fixing formatting, etc is 80% of the process

These “data wrangling” tasks are (often) best done on the command line



The Unix philosophy: do one thing well

1. Write programs that do one thing and do it well.
2. Write programs to work together.
3. Write programs to handle text streams, because that is a universal interface.

https://en.wikipedia.org/wiki/Unix_philosophy



Reminder: Basic concepts

Shell : the program through which you interact with the computer.

provides the command line and facilitates typing commands and reading outputs.

Popular shells: bash (Bourne Again Shell), csh (C Shell), ksh (Korn Shell)

Redirect : take the output of one program and make it the input of another.

we'll see some simple examples in a few slides



stdin, stdout, stderr : three special “file handles”

for reading inputs from the shell (stdin)

and writing output to the shell (stderr for error messages, stdout other information).

Special file handles: `stdin`, `stdout`, `stderr`

File handles are pointers to files

Familiar if you've programmed in C/C++

Similar: object returned by python `open()`

```
>>> f = open('workfile', 'w')
>>> print f
<open file 'workfile', mode 'w' at 80a0960>
```

By default, most command line programs

- take input from `stdin`
- Write output to `stdout`
- Write errors and status information to `stderr`

Special file handles: `stdin`, `stdout`, `stderr`

```
keith@Steinhaus:~$ echo "hello world."
hello world.
keith@Steinhaus:~$ echo "hello world." > myfile.txt
keith@Steinhaus:~$ cat myfile.txt
hello world.
keith@Steinhaus:~$ echo "!"
-bash: !: event not found
keith@Steinhaus:~$
```

`echo` sends its output to `stdout`, which is printed to the screen.

`echo` writes to `stdout`, which is redirected to the file `myfile.txt`.

`cat` writes the contents of `myfile.txt` to `stdout`, which is printed to the screen.

`echo` encounters an error, so it writes an error message to `stderr`. Both `stdout` and `stderr` are printed to the screen, but behave differently in other contexts.

Special file handles: `stdin`, `stdout`, `stderr`

```
keith@Steinhaus:~$ echo "hello world."
hello world.
keith@Steinhaus:~$ echo "hello world." > myfile.txt
keith@Steinhaus:~$ cat myfile.txt
hello world.
keith@Steinhaus:~$ echo "!"
-bash: !: event not found
keith@Steinhaus:~$
```

`echo` sends its output to `stdout`, which is printed to the screen.

`echo` writes to `stdout`, which is redirected to the file `myfile.txt`.

`cat` writes the contents of `myfile.txt` to `stdout`, which is printed to the screen.

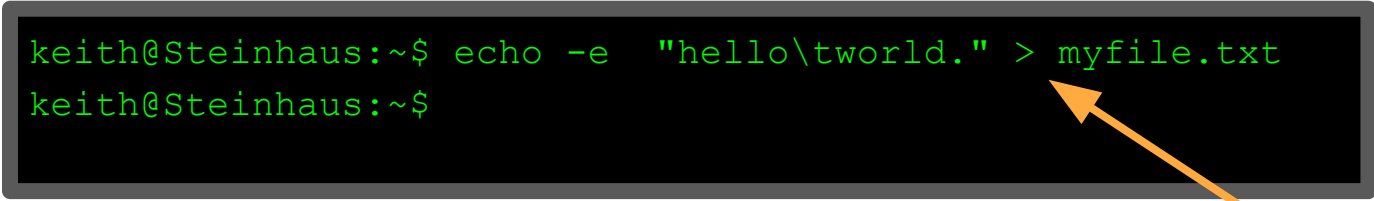
`echo` encounters an error, so it writes an error message to `stderr`. Both `stdout` and `stderr` are printed to the screen, but behave differently in other contexts.

We haven't learned any programs that use `stdin`, yet, but we will in a few slides.

Reminder: redirections using >

Redirect sends output to a file instead of stdout

```
keith@Steinhaus:~$ echo -e "hello\tworld." > myfile.txt
keith@Steinhaus:~$
```



Redirect tells the shell to send the output of the program on the “greater than” side to the file on the “lesser than” side. **This creates the file on the RHS, and overwrites the old file, if it already exists!**

Command line regexes: `grep`

Command line regex tool

```
keith@Steinhaus:~$ grep 'hello' myfile.txt
hello world.
keith@Steinhaus:~$ grep 'goat' myfile.txt
keith@Steinhaus:~$
keith@Steinhaus:~$ cat myfile.txt | grep 'hello'
hello world.
keith@Steinhaus:~$
```

Searches for the string `hello` in the file `myfile.txt`, prints all matching lines to `stdout`.

String `goat` does not occur in `myfile.txt`, so no lines to print.

`grep` can also be made to search for a pattern in its `stdin`. This is our first example of a **pipe**.

This writes the contents of `myfile.txt` to the `stdin` of `grep`, which searches its `stdin` for the string `hello`

Command line regexes: `grep`

Command line regex tool

```
keith@Steinhaus:~$ grep 'hello' myfile.txt
hello world.
keith@Steinhaus:~$ grep 'goat' myfile.txt
keith@Steinhaus:~$
keith@Steinhaus:~$ cat myfile.txt | grep 'hello'
hello world.
keith@Steinhaus:~$
```

Searches for the string `hello` in the file `myfile.txt`, prints all matching lines to `stdout`.

String `goat` does not occur in `myfile.txt`, so no lines to print.

`grep` can also be made to search for a pattern in its `stdin`. This is our first example of a **pipe**.

Note: the `grep` pattern can also be a regular expression. Use `grep -E` to tell `grep` to use “extended regular expressions”, which are (mostly) identical to those in Python `re`. See `man re_format` for more information.

Pipe (|) vs Redirect (>)

Pipe (|) reads the `stdout` from its left, and writes to `stdin` on its right.

Redirect (>) reads the `stdout` from its left and writes to a file on its right.

This is an important difference!

Warning: the example below is INCORRECT. It is an example of what NOT to do!

```
keith@Steinhaus:~$ cat myfile.txt > grep 'hello'
```

This writes the contents of `myfile.txt` to a file called `grep` and then `cats` the file `'hello'` to `stdout`, which is **not** what was intended.

Running example: Fisher's Iris data set

Widely-used data set in machine learning

Collected by E. Anderson, made famous by R. A. Fisher

Three different species: *Iris setosa*, *Iris virginica* and *Iris versicolor*

Each observation is a set of measurements of a flower:

Petal and sepal width and height (cm)

Along with species label

Common tasks:

clustering, classification








Available at UCI ML Repository: <https://archive.ics.uci.edu/ml/datasets/Iris>

Downloading the data

Following the download link on UCI ML repo leads to this index page

Index of /ml/machine-learning-databases/iris

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
 Parent Directory		-	
 Index	03-Dec-1996 04:01	105	
 bezdekIris.data	14-Dec-1999 12:12	4.4K	
 iris.data	08-Mar-1993 16:27	4.4K	
 iris.names	11-Jul-2000 21:30	2.9K	

What's the difference between these two files? The documentation actually doesn't say.

Apache/2.2.15 (CentOS) Server at archive.ics.uci.edu Port 443

Downloading the data

Create a project directory and `cd` into it.

Move the data files from downloads folder to project directory. Not mandatory, just convenient!

```
keith@Steinhaus:~$ mkdir demodir
keith@Steinhaus:~$ cd demodir
keith@Steinhaus:~/demodir$ mv ~/Downloads/iris.data .
keith@Steinhaus:~/demodir$ mv ~/Downloads/bezdekIris.data .
keith@Steinhaus:~/demodir$ ls
bezdekIris.data  iris.data      myfile.txt
keith@Steinhaus:~/demodir$ ls -l
total 40
-rw-r--r--@ 1 keith  staff  4551 Nov  15 13:47 bezdekIris.data
-rw-r--r--@ 1 keith  staff  4551 Nov  15 13:47 iris.data
-rw-r--r--@ 1 keith  staff   13 Nov   2 12:56 myfile.txt
keith@Steinhaus:~/demodir$
```

Files are there, now.

From `man ls`:

`-l` (The lowercase letter “ell”.) List in long format. (See below.) If the output is to a terminal, a total sum for all the file sizes is output on a line before the long listing.

Comparing files: `diff`

`diff` takes two files and compares them line by line

By default, prints only the lines that differ:

XcY means Xth
line in FILE1 was
replaced by Yth
line in FILE2

```
keith@Steinhaus:~/demodir$ diff iris.data bezdekIris.data
35c35
< 4.9,3.1,1.5,0.1,Iris-setosa
---
> 4.9,3.1,1.5,0.2,Iris-setosa
38c38
< 4.9,3.1,1.5,0.1,Iris-setosa
---
> 4.9,3.6,1.4,0.1,Iris-setosa
keith@Steinhaus:~/demodir$
```

< : lines from FILE1

> : lines from FILE2

Comparing files: `diff`

So, the two files differ in precisely two lines...

What's up with that?

```
keith@Steinhaus:~/demodir$ diff iris.data bezdekIris.data
35c35
< 4.9,3.1,1.5,0.1,Iris-setosa
---
> 4.9,3.1,1.5,0.2,Iris-setosa
38c38
< 4.9,3.1,1.5,0.1,Iris-setosa
---
> 4.9,3.6,1.4,0.1,Iris-setosa
keith@Steinhaus:~/demodir$
```

From UCI Documentation:

This data differs from the data presented in Fisher's article (identified by Steve Chadwick, [spchadwick '@' espeedaz.net](mailto:spchadwick@espeedaz.net)). The 35th sample should be: 4.9,3.1,1.5,0.2,"Iris-setosa" where the error is in the fourth feature. The 38th sample: 4.9,3.6,1.4,0.1,"Iris-setosa" where the errors are in the second and third features.

Comparing files: `diff`

So `bezdekIris.data` is a corrected version of `iris.data`. That's nice of them!

So, the two files differ in precisely two lines...

What's up with that?

```
keith@Steinhaus:~/demodir$ diff iris.data bezdekIris.data
35c35
< 4.9,3.1,1.5,0.1,Iris-setosa
---
> 4.9,3.1,1.5,0.2,Iris-setosa
38c38
< 4.9,3.1,1.5,0.1,Iris-setosa
---
> 4.9,3.6,1.4,0.1,Iris-setosa
keith@Steinhaus:~/demodir$
```

From UCI Documentation:

This data differs from the data presented in Fisher's article (identified by Steve Chadwick, [spchadwick '@' espeedaz.net](mailto:spchadwick@espeedaz.net)). The 35th sample should be: 4.9,3.1,1.5,0.2,"Iris-setosa" where the error is in the fourth feature. The 38th sample: 4.9,3.6,1.4,0.1,"Iris-setosa" where the errors are in the second and third features.

Comparing files: `diff`

Often useful: get the diff of two files and save it to another file

```
keith@Steinhaus:~/demodir$ diff iris.data bezdekIris.data > diff.txt
keith@Steinhaus:~/demodir$ cat diff.txt
35c35
< 4.9,3.1,1.5,0.1,Iris-setosa
---
> 4.9,3.1,1.5,0.2,Iris-setosa
38c38
< 4.9,3.1,1.5,0.1,Iris-setosa
---
> 4.9,3.6,1.4,0.1,Iris-setosa
keith@Steinhaus:~/demodir$
```

Before we go on...

It's a good habit to **always look at the data**. Go exploring!

```
keith@Steinhaus:~/demodir$ head bezdekIris.data
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
keith@Steinhaus:~/demodir$
```

Before we go on...

It's a good habit to **always look at the data**. Go exploring!

```
keith@Steinhaus:~/demodir$ head -n 70 bezdekIris.data | tail
5.0,2.0,3.5,1.0,Iris-versicolor
5.9,3.0,4.2,1.5,Iris-versicolor
6.0,2.2,4.0,1.0,Iris-versicolor
6.1,2.9,4.7,1.4,Iris-versicolor
5.6,2.9,3.6,1.3,Iris-versicolor
6.7,3.1,4.4,1.4,Iris-versicolor
5.6,3.0,4.5,1.5,Iris-versicolor
5.8,2.7,4.1,1.0,Iris-versicolor
6.2,2.2,4.5,1.5,Iris-versicolor
5.6,2.5,3.9,1.1,Iris-versicolor
keith@Steinhaus:~/demodir$
```

Before we go on...

It's a good habit to **always look at the data**. Go exploring!

```
keith@Steinhaus:~/demodir$ tail bezdekIris.data
6.9,3.1,5.1,2.3,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
6.8,3.2,5.9,2.3,Iris-virginica
6.7,3.3,5.7,2.5,Iris-virginica
6.7,3.0,5.2,2.3,Iris-virginica
6.3,2.5,5.0,1.9,Iris-virginica
6.5,3.0,5.2,2.0,Iris-virginica
6.2,3.4,5.4,2.3,Iris-virginica
5.9,3.0,5.1,1.8,Iris-virginica
keith@Steinhaus:~/demodir$
```

Species types are contiguous in the file. That means if we are going to, for example, make a train/dev/test split, we can't just take the first and second halves of the file!

File contains a trailing newline. We'll probably want to remove that!

Counting: `wc`

`wc` counts the number of lines, words, and bytes in a file or in `stdin`

Prints result to `stdout`

```
keith@Steinhaus:~/demodir$ wc bezdekIris.data
  151  150 4551 bezdekIris.data
keith@Steinhaus:~/demodir$ cat bezdekIris.data | wc
  151  150 4551
keith@Steinhaus:~/demodir$ cat bezdekIris.data | wc -l
  151
keith@Steinhaus:~/demodir$ cat bezdekIris.data | wc -w
  150
keith@Steinhaus:~/demodir$ cat bezdekIris.data | wc -c
 4551
keith@Steinhaus:~/demodir$
```

Note: a word is a group of one or more non-whitespace characters.

Counting: `wc`

`wc` counts the number of lines, words, and bytes.
Prints result to `stdout`

Test your understanding: we saw using `head` and `tail` that each line is a single word (group of non-whitespace characters), so number of words should be same as number of lines. Why isn't that the case?

```
keith@Steinhaus:~/demodir$ wc bezdekIris.data
 151 150 4551 bezdekIris.data
keith@Steinhaus:~/demodir$ cat bezdekIris.data | wc
 151 150 4551
keith@Steinhaus:~/demodir$ cat bezdekIris.data | wc -l
 151
keith@Steinhaus:~/demodir$ cat bezdekIris.data | wc -w
 150
keith@Steinhaus:~/demodir$ cat bezdekIris.data | wc -c
 4551
keith@Steinhaus:~/demodir$
```

Note: a word is a group of one or more non-whitespace characters.

Making small changes: `tr`

From the man page: The `tr` utility copies the standard input to the standard output with substitution or deletion of selected characters.

Right now, `bezdekIris.data` is comma-separated.

What if I want to make it tab-separated, instead?

`tr` is a good tool for the job

```
keith@Steinhaus:~/demodir$ cat bezdekIris.data | tr ',' '\t' > iris.tsv
keith@Steinhaus:~/demodir$ head -n 5 iris.tsv
5.1      3.5      1.4      0.2      Iris-setosa
4.9      3.0      1.4      0.2      Iris-setosa
4.7      3.2      1.3      0.2      Iris-setosa
4.6      3.1      1.5      0.2      Iris-setosa
5.0      3.6      1.4      0.2      Iris-setosa
keith@Steinhaus:~/demodir$
```

Replace commas with tabs. So we turn a comma-separated (.csv) file into a tab-separated (.tsv) file.

Making small changes: `tr`

From the man page: The `tr` utility copies the standard input to the standard output with substitution or deletion of selected characters.

```
keith@Steinhaus:~/demodir$ cat bezdekIris.data | tr '.,' ',\t' > iris_euro.tsv
keith@Steinhaus:~/demodir$ head iris_euro.tsv
5,1      3,5      1,4      0,2      Iris-setosa
4,9      3,0      1,4      0,2      Iris-setosa
4,7      3,2      1,3      0,2      Iris-setosa
4,6      3,1      1,5      0,2      Iris-setosa
5,0      3,6      1,4      0,2      Iris-setosa
5,4      3,9      1,7      0,4      Iris-setosa
4,6      3,4      1,4      0,3      Iris-setosa
5,0      3,4      1,5      0,2      Iris-setosa
4,4      2,9      1,4      0,2      Iris-setosa
4,9      3,1      1,5      0,1      Iris-setosa
keith@Steinhaus:~/demodir$
```

Turn decimal points into decimal commas, change from comma-separated to tab-separated.

Note: `tr 'abc' 'xyz'` turns all a into x, b into y, c into z. Importantly, `tr 'ab' 'bc'` turns a to b and b to c, but no a turns into c. `tr` doesn't “apply the transformation twice”

Picking out columns: `cut`

I want to make a new data set: **only** petal lengths and species

Could load everything into spreadsheet and edit there, or...

Attribute Information:

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm
5. class:
 - Iris Setosa
 - Iris Versicolour
 - Iris Virginica

```
keith:~/demodir$ cat bezdekIris.data | cut -d ',' -f 3,4,5 > petal.data
keith:~/demodir$ head -n 3 petal.data
1.4,0.2,Iris-setosa
1.4,0.2,Iris-setosa
1.3,0.2,Iris-setosa
keith:~/demodir$ head -n 3 bezdekIris.data
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
keith:~/demodir$
```

Columns delimited by `\,'`
Pick out fields 3,4 and 5.
Equivalent command:

```
cut -d '\,' -f 3-5
```

Picking out columns: `cut`

What if I want to split the attributes into their own files?

Attribute Information:

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm
5. class:
 - Iris Setosa
 - Iris Versicolour
 - Iris Virginica

```
keith:~/demodir$ cat bezdekIris.data | cut -d ',' -f 1 > sepal_len.data
keith:~/demodir$ cat bezdekIris.data | cut -d ',' -f 2 > sepal_wid.data
keith:~/demodir$ cat bezdekIris.data | cut -d ',' -f 3 > petal_len.data
keith:~/demodir$ cat bezdekIris.data | cut -d ',' -f 4 > petal_wid.data
keith:~/demodir$ cat bezdekIris.data | cut -d ',' -f 5 > species.data
keith:~/demodir$
```

Aggregation: paste and lam

Okay, I changed my mind. I want to put the five separate files back together!

```
keith:~/demodir$ paste sepal_len.data sepal_wid.data petal_len.data
petal_wid.data species.data > pasted.data
keith:~/demodir$ diff pasted.data iris.tsv
151c151
<
---
>
keith:~/demodir$
```

Recall that last line was blank, so we have some strange behavior here.

paste (from the man page):
concatenates the corresponding lines of the given input files, replacing all but the last file's newline characters with a single tab character, and writes the resulting lines to standard output.

Aggregation: paste and lam

lam (from the man page): copies the named files side by side onto the standard output.

Okay, I changed my mind. I want to put the five separate files back together!

```
keith:~/demodir$ lam sepal_len.data -s ',' sepal_wid.data -s ','  
petal_len.data -s ',' petal_wid.data -s ',' species.data | head -n 3  
5.1,3.5,1.4,0.2,Iris-setosa  
4.9,3.0,1.4,0.2,Iris-setosa  
4.7,3.2,1.3,0.2,Iris-setosa  
keith:~/demodir$ lam sepal_len.data -s ',' sepal_wid.data -s ','  
petal_len.data -s ',' petal_wid.data -s ',' species.data | tail -n 3  
6.2,3.4,5.4,2.3,Iris-virginica  
5.9,3.0,5.1,1.8,Iris-virginica  
,,,  
keith:~/demodir$
```

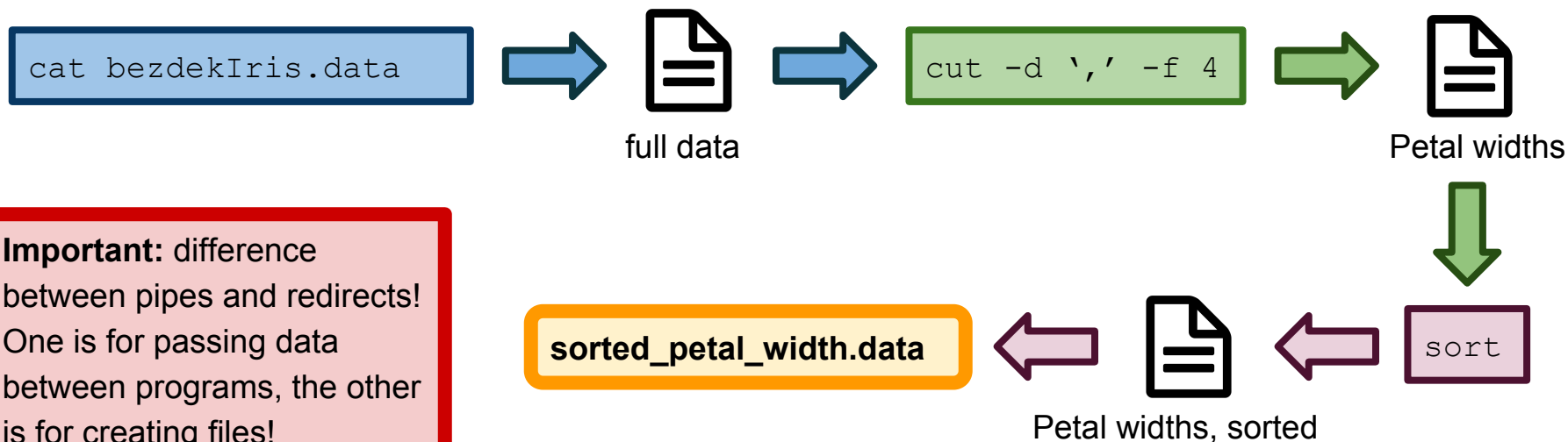
Have to specify a separator character with `-s` everywhere I want one.

Recall that the last line is blank, which `lam` handles as required, but here's a good reason to have removed that blank line sooner.

Sorting: `sort`

`sort` reads from `stdin`, sorts the lines, and sends the result to `stdout`.

```
keith:~$ cat bezdekIris.data | cut -d ',' -f 4 | sort > sorted_petal_width.data
keith:~$
```

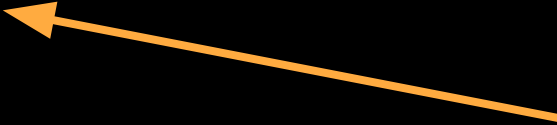


Important: difference between pipes and redirects! One is for passing data between programs, the other is for creating files!

Sorting: `sort`

```
keith:~$ cat bezdekIris.data | cut -d ',' -f 4 | sort > sorted_petal_width.data
keith:~$ head -n 8 sorted_petal_width.data
```

```
0.1
0.1
0.1
0.1
0.1
0.2
0.2
```



Blank line is still giving us trouble!

```
keith:~$ tail -n 2 sorted_petal_width.data
2.5
2.5
keith:~$
```

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, so we'll focus on one: **substitutions**

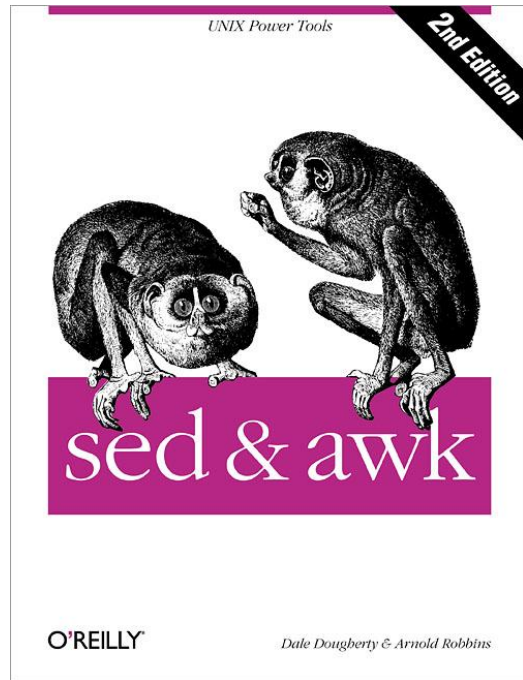
```
keith:~$ echo "hello world" | sed 's/hello/goodbye/g'
goodbye world
```

`s` for substitute

Replace this...

...with this.

`g` for globally, meaning everywhere in the input.

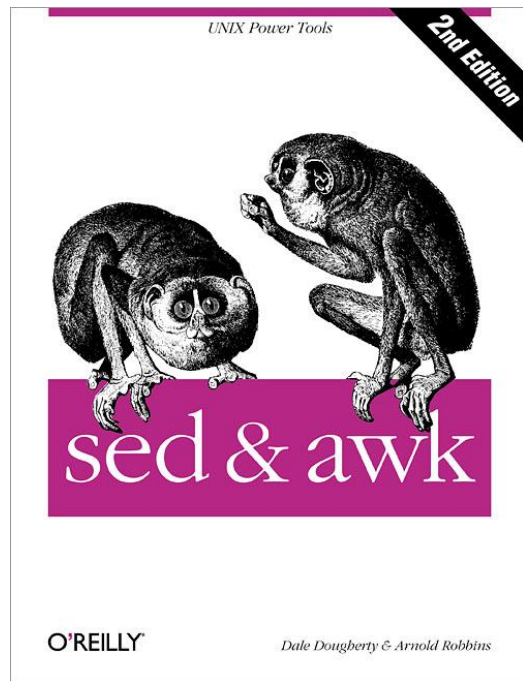


Editing text streams: sed

sed commands can include regular expressions

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

`*` Works like in Python `re`



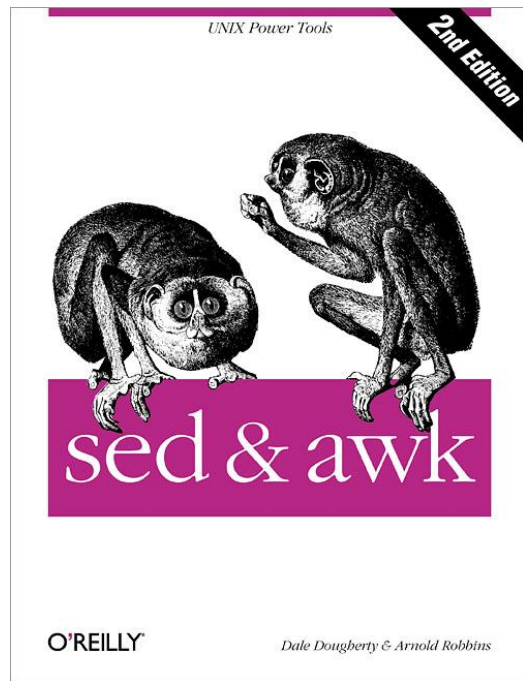
Editing text streams: `sed`

`sed` commands can include regular expressions

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

`*` Works like in Python `re`

Test your understanding: is the `sed` regex matcher greedy?



Editing text streams: sed

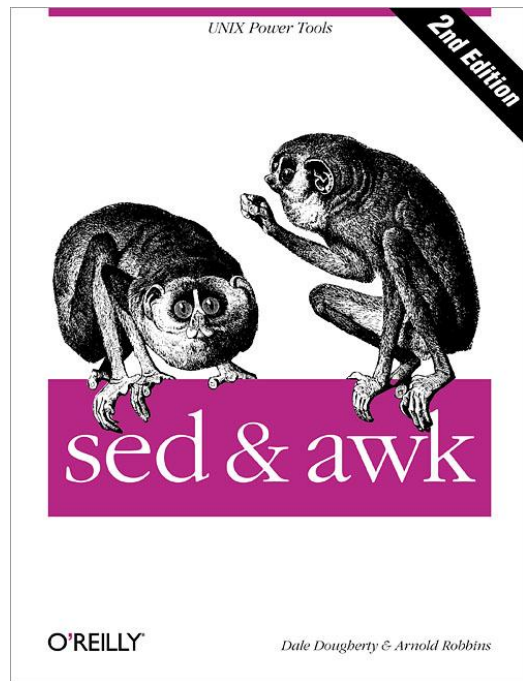
sed commands can include regular expressions

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

'*' Works like in Python re

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

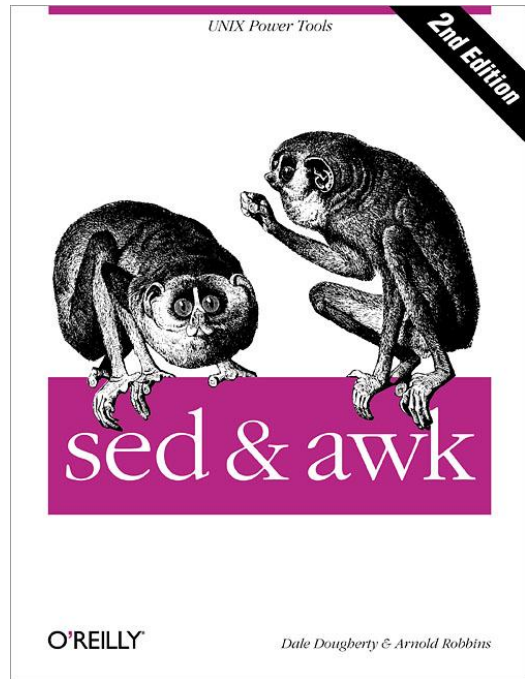
```
keith:~$ echo "a aa aaa" | sed 's/a*/b/g'  
b b b
```

`*` Works like in Python `re`

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 Python `re` package will transfer directly to `sed`, as well as other tools (e.g., `grep` and `perl`)



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 Python re

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

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

To use "extended" regexes,
need to give -E flag.



Editing text streams: sed

Basic syntax of sed s commands:

```
sed 's/regexp/replacement/flags'
```

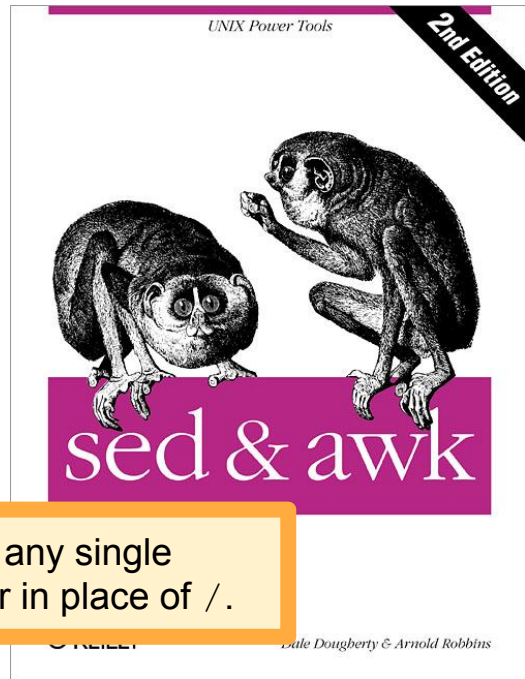
```
keith:~$ echo "a aa aaa" | sed -E 's/a+/b/g'
b b b
keith:~$ echo "a aa aaa" | sed -E 's|a+|b|g'
b b b
keith:~$ echo "a| aa| aaa| aaaa" | sed -E 's/a+\\|/b/g'
b b b aaaa
keith:~$
```

Can use any single character in place of /.

Special characters have to be escaped.

All the power of Python regexes, but with the convenience of the command line! And we're only barely scratching the surface:

https://www.gnu.org/software/sed/manual/html_node/index.html#Top



Basic Shell Scripting

Bash (and other shells) support scripting

Useful for automating repetitive tasks:

E.g., Renaming files; processing files in batches

The Bash command line supports its own programming language

Has variables, conditionals, for-loops, etc.

We'll only scratch the surface of this, here. See, for example, the Linux Documentation Project (TLDP, www.tldp.org) or *Learning the Bash Shell* by C. Newham for more.

Basic Shell Scripting

Variable assignment in bash is of the form

```
VARIABLE=[value]
```

Note that there should be NO spaces between the variable name and the assignment operator and between the assignment operator and the value.

```
keith:~$ MYVAR='cat dog bird goat'
keith:~$ echo $MYVAR
cat dog bird goat
keith:~$ FILENAME="myfile.txt"
keith:~$ echo "here is some text" > $FILENAME
keith:~$ cat $FILENAME
here is some text
keith:~$ echo FILENAME
FILENAME
keith:~$
```

To retrieve the value of a variable, prepend it with a dollar sign \$.

Once `FILENAME` has a value, we can treat it just as though we were writing the actual name of a file in its place.

Common error: forgetting to prepend with a dollar sign \$.

Basic Shell Scripting

For loops take the form
`for vname in <set>; do <expr>; done`

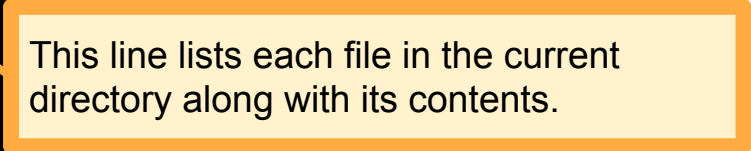
```
keith:~$ MYVAR='cat dog bird goat'
keith:~$ for s in $MYVAR; do echo $s; done
cat
dog
bird
goat
keith:~$ for x in `echo "1 2 3 4 5"`; do echo "$x" > ${x}.txt; done
keith:~$ ls
1.txt    2.txt    3.txt    4.txt    5.txt    myfile.txt
keith:~$
```

Enclosing in backticks (`) turns the output of the expression to a string-like expression that can be assigned to a variable or iterated over.

Enclosing a variable in curly braces is a good habit when putting a variable in a longer string. Prevents ambiguity of `$x.txt` or `$xfile.txt` vs `${x}file.txt`.

Basic Shell Scripting

```
keith:~$ for x in `echo "1 2 3 4 5"`; do echo "$x" > ${x}.txt; done
keith:~$ ls
1.txt    2.txt    3.txt    4.txt    5.txt    myfile.txt
keith:~$ for f in `ls .`; do echo -n "${f} : "; cat $f; done
1.txt : 1
2.txt : 2
3.txt : 3
4.txt : 4
5.txt : 5
myfile.txt : here is some text
keith:~$
```



This line lists each file in the current directory along with its contents.

Basic Shell Scripting

```
keith:~$ for x in `echo "1 2 3 4 5"`; do echo "$x" > ${x}.txt; done
keith:~$ ls
1.txt    2.txt    3.txt    4.txt    5.txt    myfile.txt
keith:~$ for f in `ls .`; do echo -n "${f} : "; cat $f; done
1.txt : 1
2.txt : 2
3.txt : 3
4.txt : 4
5.txt : 5
myfile.txt : here is some text
keith:~$
```

This line lists each file in the current directory along with its contents.

Lots more tools available (not in this lecture):
More syntax: conditionals, while-loops, etc.
Scripts: put a sequence of commands into a file and run it from the command line.

Readings (this lecture)

Required:

None

Recommended:

J. Janssens *Data Science at the Command Line*

<http://shop.oreilly.com/product/0636920032823.do>

S. Das (2005,2012). *Your UNIX: the Ultimate Guide*. McGraw-Hill.

Sed manual:

https://www.gnu.org/software/sed/manual/html_node/index.html#Top

GNU `awk` user's guide:

<https://www.gnu.org/software/gawk/manual/gawk.html>