STAT 605
Data Science Computing

Introduction to Shell Scripting

Basic concepts

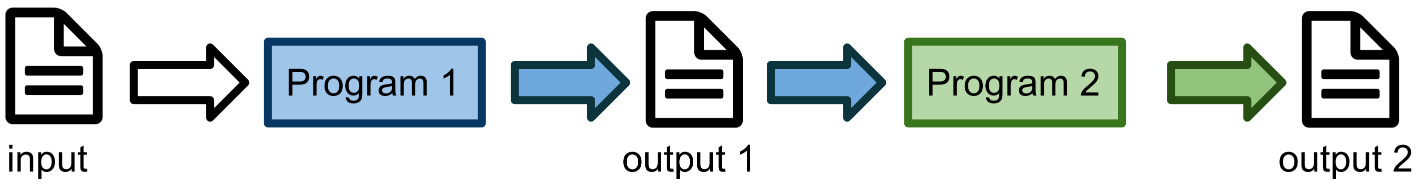
Shell : the program through which you interact with the computer.

Reads, parses and executes the commands typed into the terminal

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

Redirect : take the output of one program and send it somewhere else

we’ll see some simple examples soon



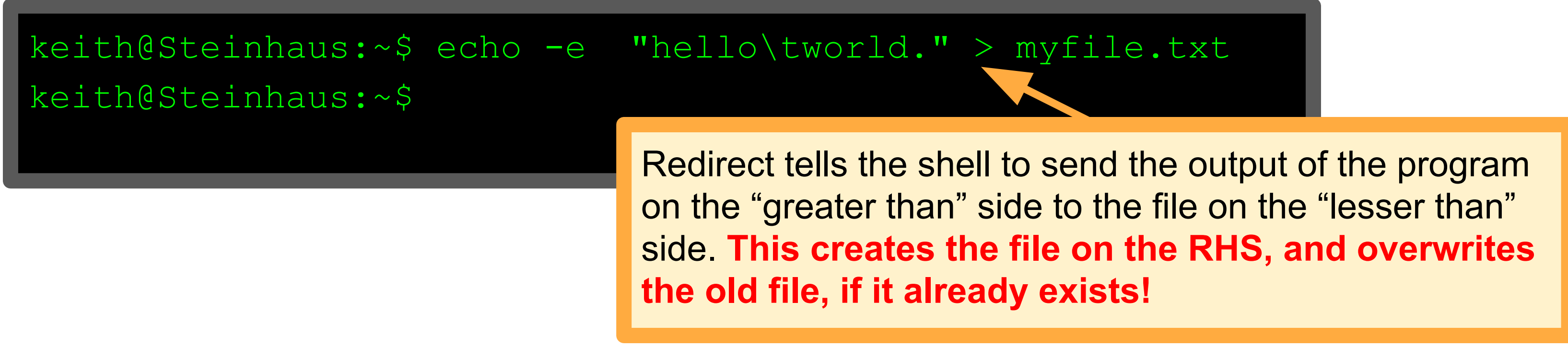
stdin, stdout, stderr : three special “file handles”

for reading inputs from the shell (stdin)

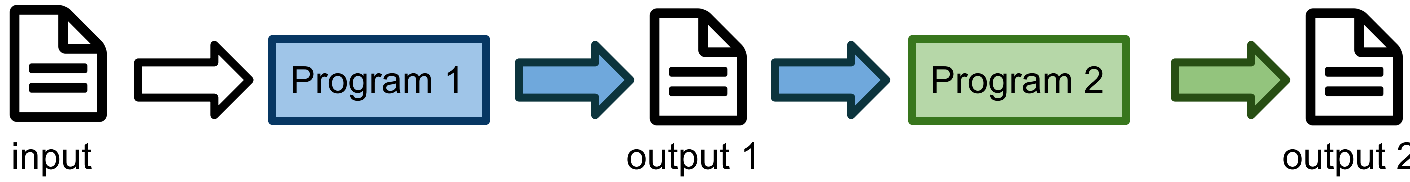
and writing output to the shell (stderr for error messages, stdout other information).

Reminder: redirections using >

Redirect sends output to a file instead of stdout



But what if I want to pass the output on the left to another program, instead?



Command line regexes: grep

grep is a command line search 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, which we'll learn about soon

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 `cat`s 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

Name	Last modified	Size	Description
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	

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

What's the difference between these two files?

Downloading the data

```
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$
```

Create a project directory and `cd` into it.

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

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:

```
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$
```

`XcY` means `X`th line in `FILE1` was replaced by `Y`th line in `FILE2`

< : 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). 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). 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, wo

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 data 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 **d**elimited by `‘,’`
Pick out **f**ields 3,4 and 5.
Equivalent command:
`cut -d ‘,’ -f 3-5`

Picking out columns: `cut`

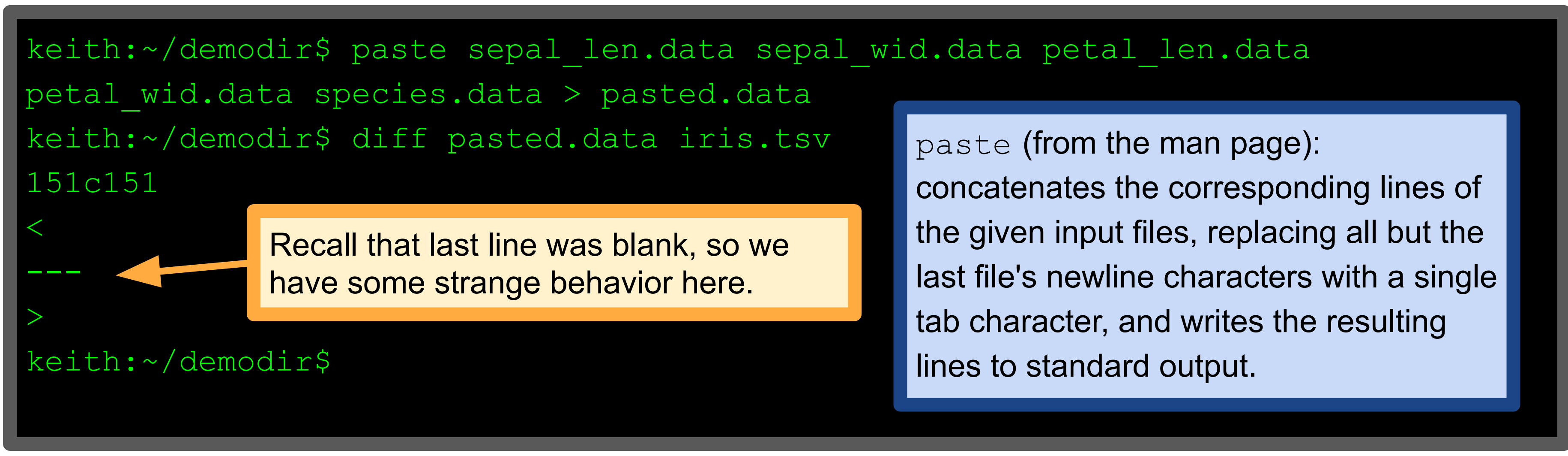
What if I want to split the attributes into their own files?

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!



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.

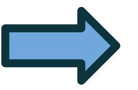
Recall that the last line is blank, which `lam` handles as required, but here's a good reason to have removed that blank line sooner.

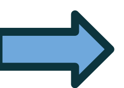
Sorting: `sort`

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

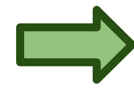
```
keith:~$ cat bezdekIris.data | cut -d ',' -f 4 | sort > sorted_petal_width.data
keith:~$
```

`cat bezdekIris.data`

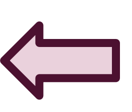


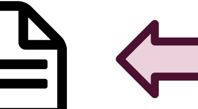


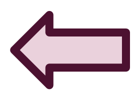
`cut -d ',' -f 4`



`sort`







sorted_petal_width.data

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

Blank line is still giving us trouble!

`find`: searching for files

Basic usage: `find <where> <comparison> <pattern>`

Example: `find ./my_dir -name "my_file.txt"`

Looks in directory `my_dir` for a file matching the name `my_file.txt`

```
keith:~$ ls Lec_shellscript/
foo.txt  myfile.txt
keith:~$ find Lec_shellscript -name myfile.txt
Lec_shellscript/myfile.txt
keith:~$
```

There are a mess of other options for controlling search. For example, pattern matching, directory depth, date of last access, etc. See **`man find`** for more.

UNIX Groups

On UNIX-like systems, files are owned by users

On UNIX/Linux/MacOS:

```
[klevin@cavium-thunderx-login01 pyspark_demo]$ ls -l
total 241
-rw-r--r-- 1 klevin statistics 1170 Mar 12 11:09 gen_demo_data.py
-rw-r--r-- 1 klevin statistics   39 Mar 12 11:12 poly.py
-rw-r--r-- 1 klevin statistics  239 Mar 12 11:09 prime.py
-rw-r--r-- 1 klevin statistics 1269 Mar 12 11:09 ps_demo.py
-rw-r--r-- 1 klevin statistics  746 Mar 12 11:09 ps_wordcount.py
drwxr-xr-x 2 klevin statistics   75 Mar 12 11:18 __pycache__
-rw-r--r-- 1 klevin statistics  251 Mar 12 11:09 scientists.txt
```

UNIX Groups

On UNIX-like systems, files are owned by users

On UNIX/Linux/MacOS:

This column lists which user owns the file

```
[klevin@cavium-thunderx-login01 pyspark_demo]$ ls -l
total 241
-rw-r--r-- 1 klevin statistics 1170 Mar 12 11:09 gen_demo_data.py
-rw-r--r-- 1 klevin statistics   39 Mar 12 11:12 poly.py
-rw-r--r-- 1 klevin statistics  239 Mar 12 11:09 prime.py
-rw-r--r-- 1 klevin statistics 1269 Mar 12 11:09 ps_demo.py
-rw-r--r-- 1 klevin statistics  746 Mar 12 11:09 ps_wordcount.py
drwxr-xr-x 2 klevin statistics   75 Mar 12 11:18 __pycache__
-rw-r--r-- 1 klevin statistics  251 Mar 12 11:09 scientists.txt
```

UNIX Groups

Legend
- d : directory
- r : read access
- w : write access
- x : execute access

On UNIX-like systems, files are owned by users

On UNIX/Linux/MacOS:

These lines are permission information.

```
[klevin@cavium-thunderx-login01 pyspark_demo]$ ls -l
total 241
-rw-r--r-- 1 klevin statistics 1170 Mar 12 11:09 gen_demo_data.py
-rw-r--r-- 1 klevin statistics   39 Mar 12 11:12 poly.py
-rw-r--r-- 1 klevin statistics  239 Mar 12 11:09 prime.py
-rw-r--r-- 1 klevin statistics 1269 Mar 12 11:09 ps_demo.py
-rw-r--r-- 1 klevin statistics  746 Mar 12 11:09 ps_wordcount.py
drwxr-xr-x 2 klevin statistics   75 Mar 12 11:18 __pycache__
-rw-r--r-- 1 klevin statistics  251 Mar 12 11:09 scientists.txt
```

UNIX Groups

On UNIX-like systems, files are owned by users

Legend
d : directory
r : read access
w : write access
x : execute access

On UNIX/Linux/MacOS:

These specific columns specify owner permissions.
The owner has these permissions on these files.

```
[klevin@cavium-thunderx-login01 pyspark_demo]$ ls -l
total 241
-rw-r--r-- 1 klevin statistics 1170 Mar 12 11:09 gen_demo_data.py
-rw-r--r-- 1 klevin statistics   39 Mar 12 11:12 poly.py
-rw-r--r-- 1 klevin statistics  239 Mar 12 11:09 prime.py
-rw-r--r-- 1 klevin statistics 1269 Mar 12 11:09 ps_demo.py
-rw-r--r-- 1 klevin statistics  746 Mar 12 11:09 ps_wordcount.py
drwxr-xr-x 2 klevin statistics   75 Mar 12 11:18 __pycache__
-rw-r--r-- 1 klevin statistics  251 Mar 12 11:09 scientists.txt
```

UNIX Groups

Legend
`d` : directory
`r` : read access
`w` : write access
`x` : execute access

On UNIX-like systems, files are owned by users

Sets of users, called **groups**, can be granted special permissions

On UNIX/Linux/MacOS:

This column lists what group owns the file

```
[klevin@cavium-thunderx-login01 pyspark_demo]$ ls -l
total 241
-rw-r--r-- 1 klevin statistics 1170 Mar 12 11:09 gen_demo_data.py
-rw-r--r-- 1 klevin statistics   39 Mar 12 11:12 poly.py
-rw-r--r-- 1 klevin statistics  239 Mar 12 11:09 prime.py
-rw-r--r-- 1 klevin statistics 1269 Mar 12 11:09 ps_demo.py
-rw-r--r-- 1 klevin statistics  746 Mar 12 11:09 ps_wordcount.py
drwxr-xr-x 2 klevin statistics   75 Mar 12 11:18 __pycache__
-rw-r--r-- 1 klevin statistics  251 Mar 12 11:09 scientists.txt
```

UNIX Groups

Legend
d : directory
r : read access
w : write access
x : execute access

On UNIX-like systems, files are owned by users

Sets of users, called **groups**, can be granted special permissions

On UNIX/Linux/MacOS:

These specific columns specify group permissions. Anyone in the statistics group has these permissions on these files.

```
[klevin@cavium-thunderx-login01 pyspark_demo]$ ls -l
total 241
-rw-r--r-- 1 klevin statistics 1170 Mar 12 11:09 gen_demo_data.py
-rw-r--r-- 1 klevin statistics   39 Mar 12 11:12 poly.py
-rw-r--r-- 1 klevin statistics  239 Mar 12 11:09 prime.py
-rw-r--r-- 1 klevin statistics 1269 Mar 12 11:09 ps_demo.py
-rw-r--r-- 1 klevin statistics  746 Mar 12 11:09 ps_wordcount.py
drwxr-xr-x 2 klevin statistics   75 Mar 12 11:18 __pycache__
-rw-r--r-- 1 klevin statistics  251 Mar 12 11:09 scientists.txt
```

UNIX Groups

Legend
`d` : directory
`r` : read access
`w` : write access
`x` : execute access

On UNIX-like systems, files are owned by users

Sets of users, called **groups**, can be granted special permissions

On UNIX/Linux/MacOS:

These specific columns specify the permissions for everyone else on the system (i.e., anyone who is not klevin and not in the `statistics` group.

```
[klevin@cavium-thunderx-lo
total
-rw-r--r-- 1 klevin statistics 1170 Mar 12 11:09 gen_demo_data.py
-rw-r--r-- 1 klevin statistics   39 Mar 12 11:12 poly.py
-rw-r--r-- 1 klevin statistics  239 Mar 12 11:09 prime.py
-rw-r--r-- 1 klevin statistics 1269 Mar 12 11:09 ps_demo.py
-rw-r--r-- 1 klevin statistics  746 Mar 12 11:09 ps_wordcount.py
drwxr-xr-x 2 klevin statistics   75 Mar 12 11:18 __pycache__
-rw-r--r-- 1 klevin statistics  251 Mar 12 11:09 scientists.txt
```

Changing permissions: chmod

We can change the permissions on a file with the chmod command

Usage: chmod <who><+-=><permissions> [file]

Who: u for owner, g for group, o for others, a for all

Add/set/remove: + to add, – to remove, = to set to these permissions

Permissions: r for read, w for write, x for execute

```
keith:~$ ls -l
total 8
-rw-r--r--@ 1 keith  staff  13 Sep 16 18:39 myfile.txt
keith:~$ chmod go+w myfile.txt
keith:~$ ls -l
total 8
-rw-rw-rw-@ 1 keith  staff  13 Sep 16 18:39 myfile.txt
```

Group and others have only read permissions

Group and others have gained the write permission.

Scripting in `bash`

`bash` (short for "Bourne again shell") is the Ubuntu command line program

- Bash is a programming language unto itself
- We can write for-loops, if-then statements, etc., just like in other languages

Example: cat the contents of every file in a directory

Example: look at each file in a directory and change its ownership permissions

Example: rename every file in a directory to change its name to all lower-case

Bash scripting lets us combine the "single-purpose" command line tools into powerful, complex, reusable programs. It is worth your time to learn this well!

When *shouldn't* I use `bash`?

`bash` is a scripting language

That means it's useful for prototyping and "quick-and-dirty" tasks…

...but it's less well-suited to other problems

`bash` is **not** the best tool for the job if you need

Complicated mathematical operations (e.g., floating point arithmetic)

High-throughput tasks (bash is *very* slow)

Data structures (e.g., R vectors)

Anatomy of a `bash` script

By convention, we write bash scripts with a `.sh` extension: `my_script.sh`

And every script starts with a “crunch-bang”, `#!` (see `man magic` for more):

```
#!/bin/sh
```

(`echo $SHELL` will tell you which shell you’re using)

This tells the system what shell to use when running this script

We run our script (provided we have execute permissions!) like any other command line program:

```
keith:~$ ./my_script.sh
```

See the lecture video for a demonstration of writing a simple bash script.

Exercises: Part 1

1) Write a bash script called "my_first_script.sh" that echos your username (have a look at the command `whoami`), the date (see the command `date`) and what shell you're using (recall the $SHELL internal variable)
2) Use chmod to make it so that the owner can read, write and execute the file, while the rest of the group and all other users are only able to read (hint: `chmod X=Y` makes it so that `X` has exactly the permissions `Y`, where `Y` can be, for example, `rw` for read and write access but *not* execute).

Variables in bash

We declare a variable (and assign it a value) with

```
variable_name=VALUE
```

We retrieve the value of a variable with

```
$variable_name
```

We can also give a variable the output of a program in two different ways:

```
username=$(whoami)
contents=`echo my_file.txt`
```

Note: variables can have any capitalization we want, but by convention, variables that we create are lower-case, with upper-case variables, like `$SHELL`, reserved for global or system-level variables (called “internal variables”).

Accessing command line arguments

Command line arguments are accessible as variables `$1`, `$2`, `$3`,...

- variable `$#` stores the number of command line args
- `$*` stores list of all command line arguments
- `$0` is the name of the running file

Example: Inside the running `./my_script.sh file.txt bar stat605` ,
`cat $1` would cat the contents of file.txt
`echo $3` would print “stat605”
`echo $0` would print “my_script.sh”

Related: we can define functions in bash

```
function my_new_function() {
    Arguments are accessible as $1, $2, …
}
```

Once we have defined a function, we can reuse it elsewhere in our script

See the lecture video for a demonstration of creating and using variables in bash.

Exercises: Part 2

1) Write a script called `my_grep.sh` that takes two command line arguments, a file and a string. Make your script use grep to search for the given string in the given file. **Note:** this is yet another rather silly example, since we are writing a script to do what `grep` already does for us. It's the kind of thing that we wouldn't do in practice, but it's a good way to get you familiar with `bash`.
2) Write a script called `count_args.sh` that takes any number of command line arguments, and prints the number of arguments that it got. **Hint:** use the built-in `$*` variable.

Conditional statements

```
if [ CONDITION ]; then
    CODE TO EXECUTE
fi
```

Unlike many programming languages, `bash` doesn't have Boolean types (i.e., values TRUE and FALSE). Instead, 0 is "true" in bash, and 1 is "false". Yes, I agree it's counter-intuitive!

```
keith:~$ a="dog"; b="cat";
keith:~$ if [ $a = $b ]; then echo "a and b are equal (as strings)"; fi
keith:~$ a="cat"; b="cat";
keith:~$ if [ $a = $b ]; then echo "a and b are equal (as strings)"; fi

a and b are equal (as strings)

keith:~$
```

Note: bash has different symbols for string comparison and numerical comparison. Refer to **`man test`** to read more.

Conditional statements

```
if [ CONDITION ]; then
    CODE TO EXECUTE
fi
```

Unlike many programming languages, `bash` doesn't have Boolean types (i.e., values TRUE and FALSE). Instead, 0 is "true" in bash, and 1 is "false". Yes, I agree it's counter-intuitive!

```
keith:~$ a="dog"; b="cat";
keith:~$ if [ $a = $b ]; then echo "a and b are equal (as strings)"; fi
keith:~$ a="cat"; b="cat";
keith:~$ if [ $a = $b ]; then echo "a and b are equal (as strings)"; fi

a and b are equal (as strings)

keith:~$ [ 9 -eq 6 ]
keith:~$ echo $?
1
keith:~$
```

Note: bash has different symbols for string comparison and numerical comparison. Refer to **`man test`** to read more.

`$?` always holds the result of the previous command.

Conditional statements

```
if [ CONDITION ]; then
   CODE TO EXECUTE
elif [ CONDITION2 ]; then
   CODE FOR COND2
else
    DIFFERENT CODE
fi
```

```
keith:~$ a="dog"; b="cat";
keith:~$ if [ $a = $b ]; then echo "a and b are equal (as strings)"; else
echo "a and b are NOT equal"; fi
a and b are NOT equal
keith:~$
```

See the lecture video for a demonstration of using conditionals in bash.

Exercises: Part 3

1) Create a bash script called `compare.sh` that takes two numbers as its arguments, and prints “greater” if the first argument is greater than the second, “less” if the first argument is less than the second, and “equal” otherwise. You may assume the inputs are both numbers.

2) Add error checking to `compare.sh` by using a conditional to check that two arguments were supplied. If the number of arguments is not as expected, print a message saying so, and exit with the exit status 1, using `exit 1` (see https://tldp.org/LDP/abs/html/exit-status.html for more on `exit` and exit status). Use `echo “MESSAGE” 1>&2`, to print your message to `stderr`. This redirects `echo`’s `stdout` to the `stderr` of our script. See https://tldp.org/LDP/abs/html/io-redirection.html for more information.

Loops: `for` and `while`

```
for i in LIST; do
    body of for-loop
done;
```

`LIST` is usually just a collection of strings separated by spaces. It can be the output of another program, a pattern we'll see in the VM.

```
while [ CONDITION ]; do
    body of while-loop
done;
```

Execute this code repeatedly so long as `CONDITION` evaluates to true.

Bash also features the more exotic until-loop, which is like the opposite of a while loop. It runs until the condition evaluates to true. Just write `until` instead of `while` in the code above.

See the lecture video for a demonstration of using for-loops and while-loops in bash.

Exercises: Part 4

1) Create a bash script `first_lines.sh` that prints the first line of every file in the current directory.
2) Write a script `list_dirs.sh` that prints the names of all the directories in the current directory, one per line.
3) Create a bash script called `my_head.sh`, that mimics the behavior of `head` (except for the flags), by taking two command line arguments: a file and a number, say, N. Print the first N lines of the file (if N is bigger than the number of lines in the file, then your script should just print the whole file). **Hint:** the syntax `while read line; do [code]; done` will read one line at a time from `stdin` into the variable `$line`, accessible inside the while-loop. See here for an example: https://linuxhint.com/read_file_line_by_line_bash/