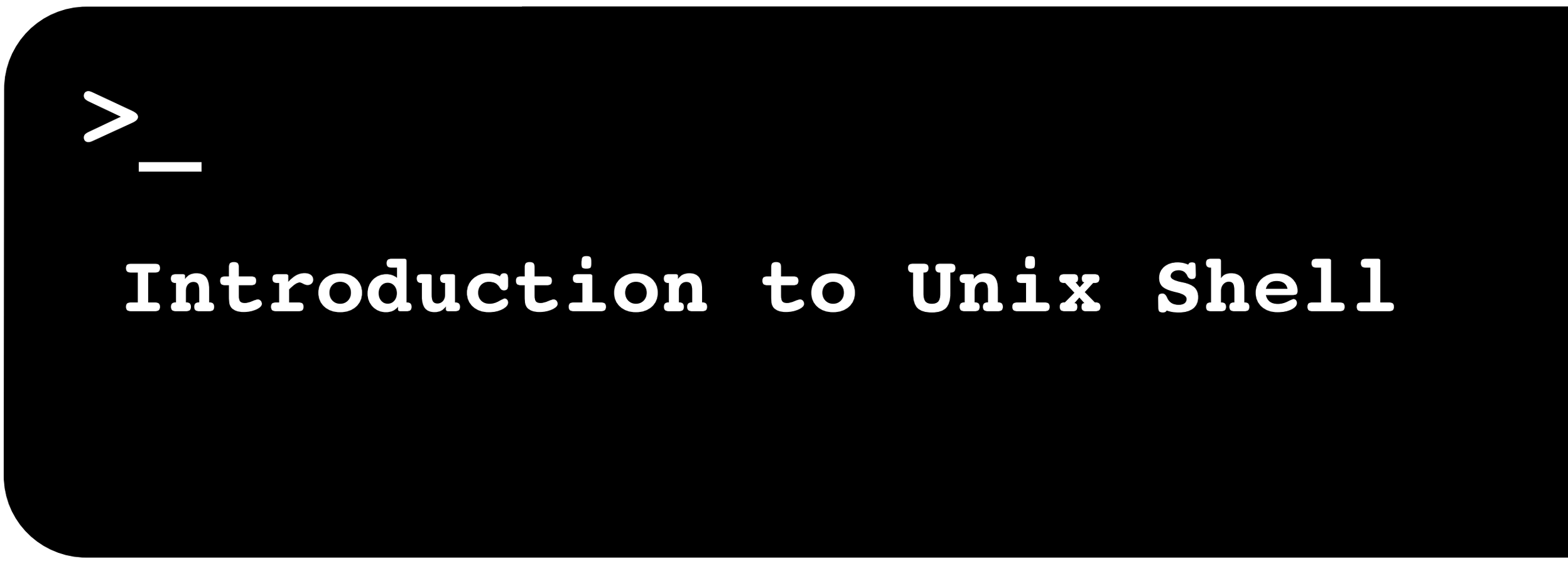
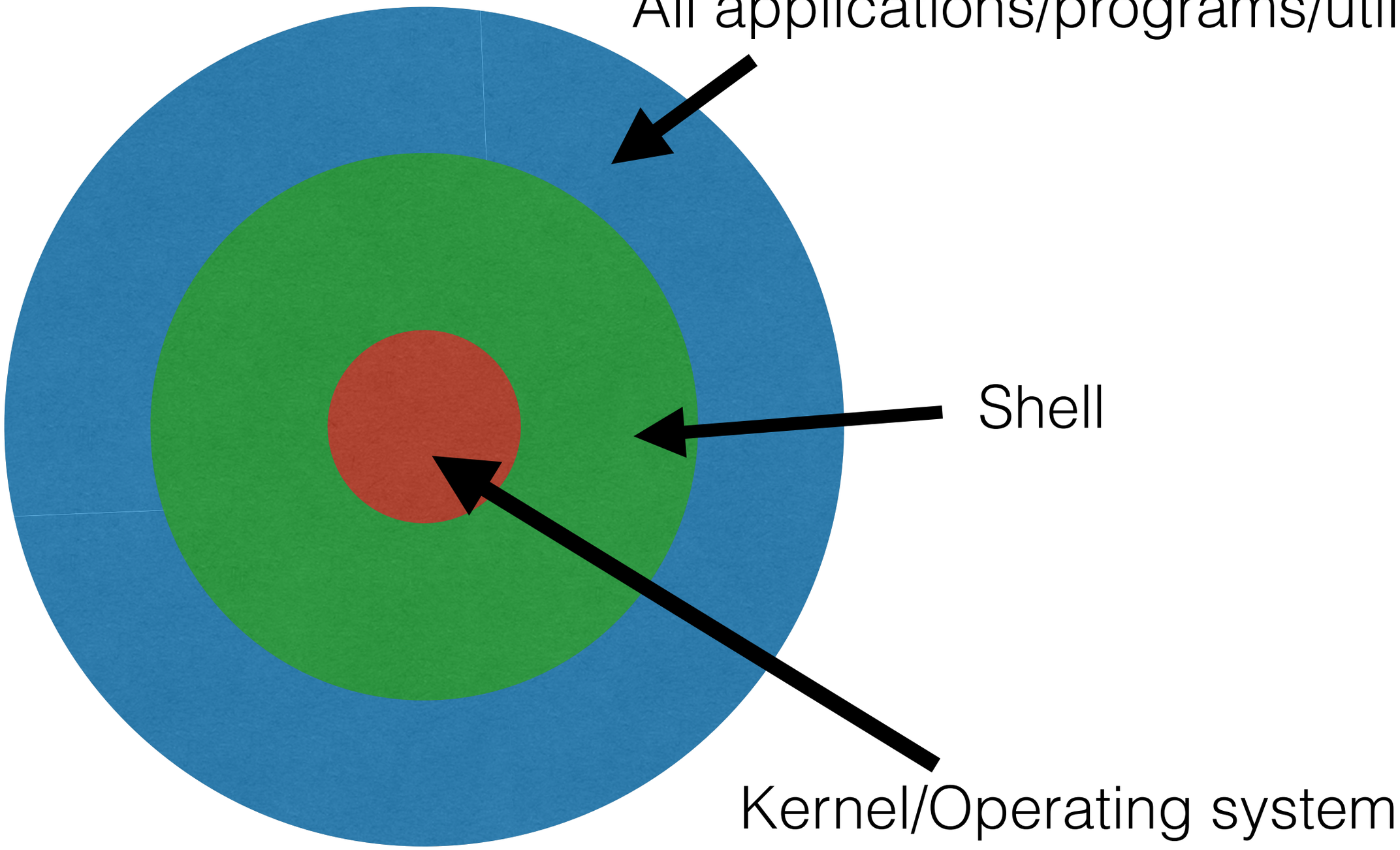
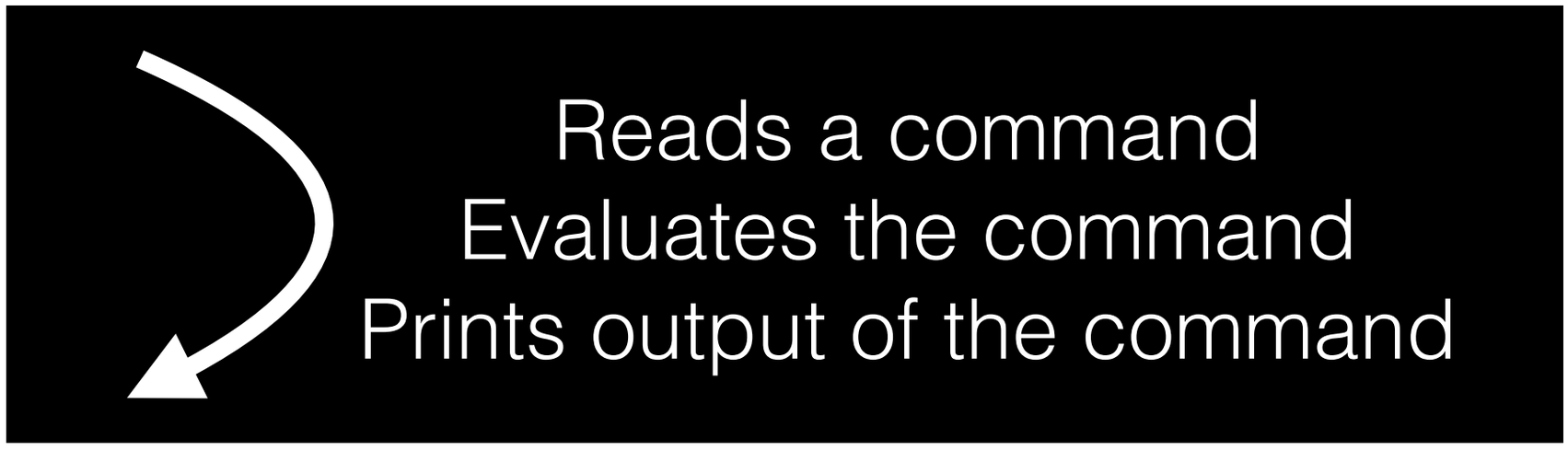


http://swcarpentry.github.io/shell-novice/

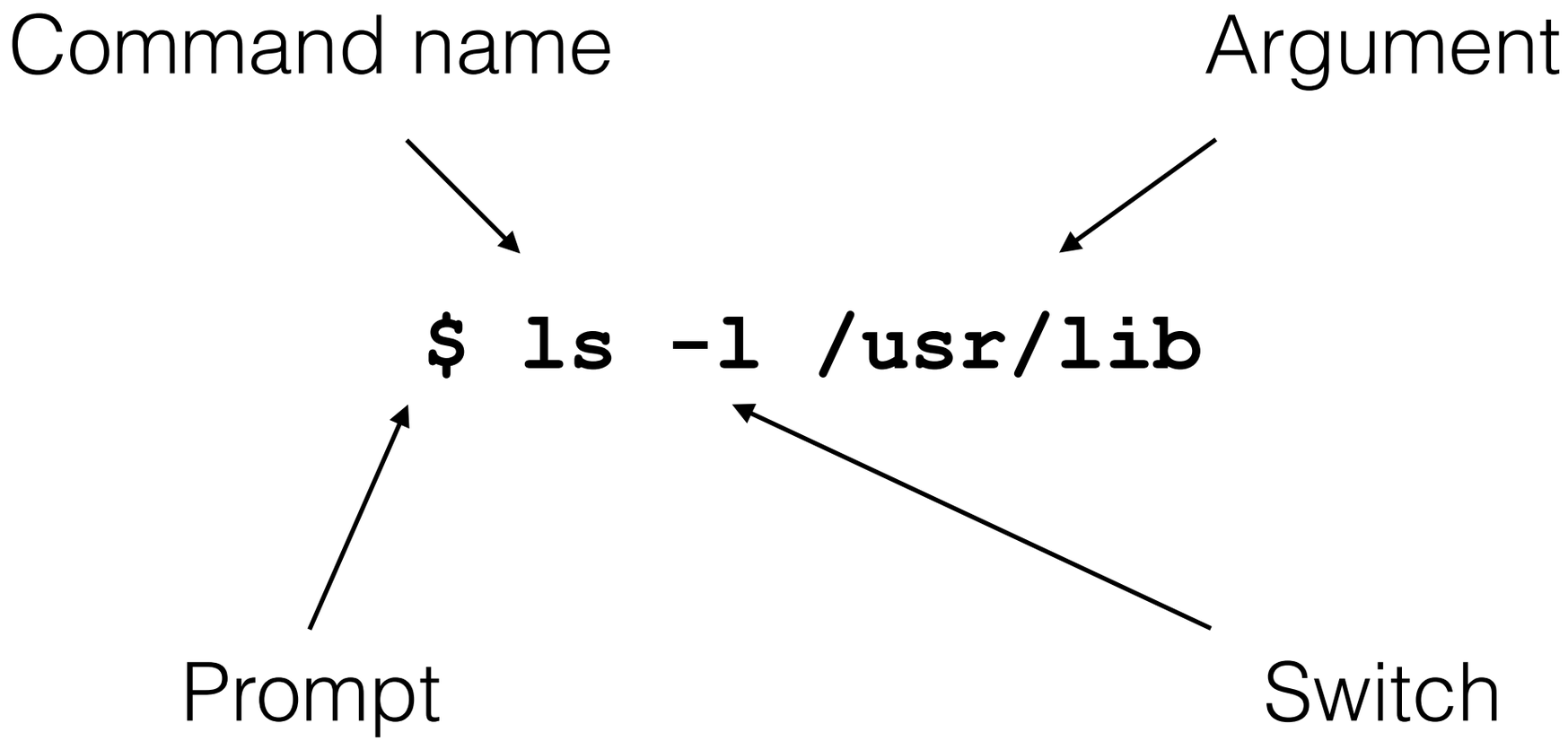

All applications/programs/utilities
Shell
Kernel/Operating system

Command line interface (CLI) vs. Graphical User Interface (GUI)



- CLI and GUI are complimentary to each other
- Some things are faster in CLI and some are impossible
- Remote access/programming tools/repetitive tasks are much easier in CLI
- Depending what do you need choose CLI or GUI

When you execute a command



Keyboard (input) > Shell (interpretation) >
Execution > Screen (output)

When you execute a command

In more details:

- **$** - prompt - signal that the shell is ready for a command
- every command ends with **<Enter>** or **<return>**
- Shell will look for the command (in the standard location defined by variable **$PATH** (shell settings)
- Command is executed and it's standard output is printed to the screen (more "output" may appear as files)
- **$** - prompt is displayed again

When you execute a command

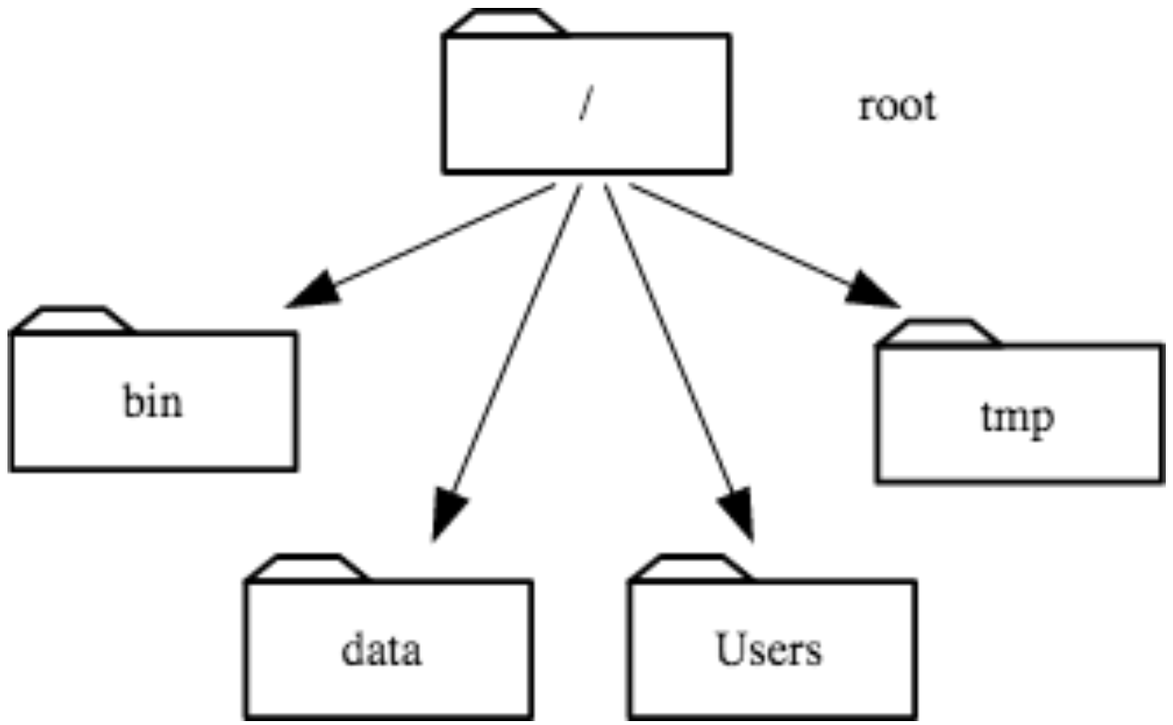
Commands have MANY switches are may take several arguments

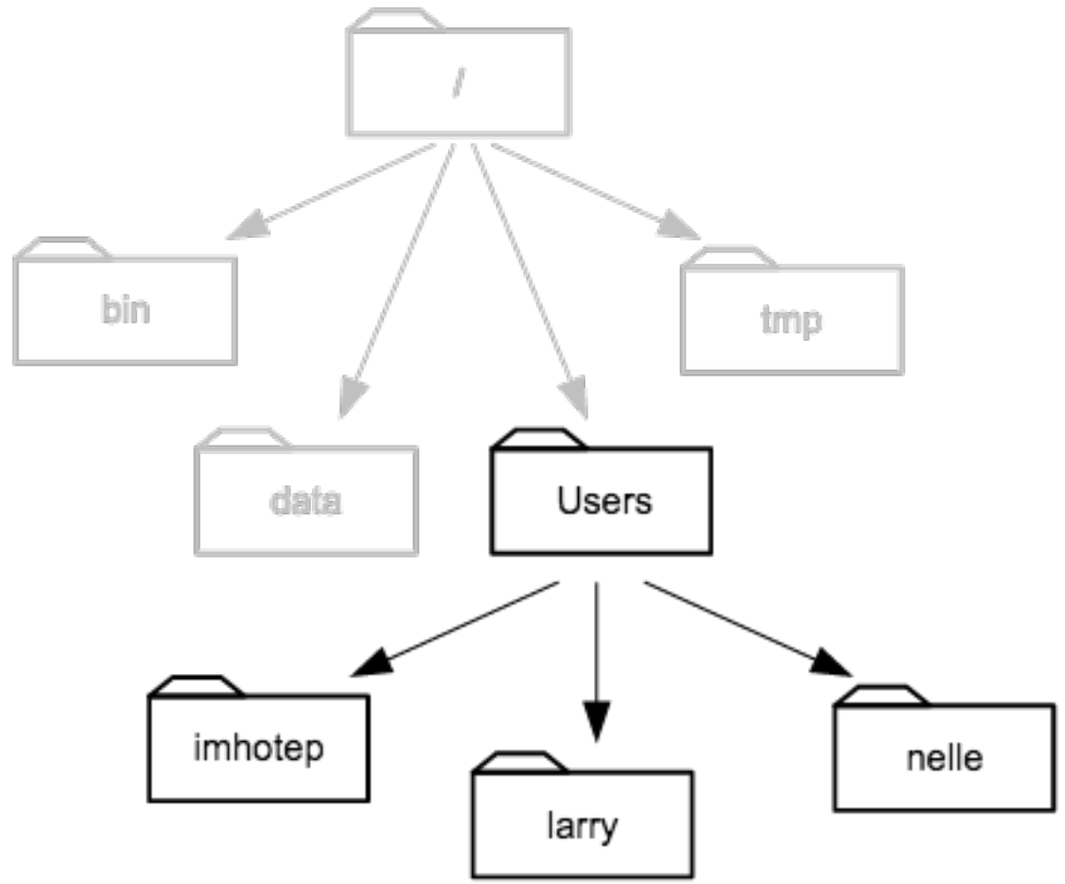
The same command in different distributions of operating systems may behave in different way e.g. **`ls`** (GNU) vs. **`ls`** (BSD)

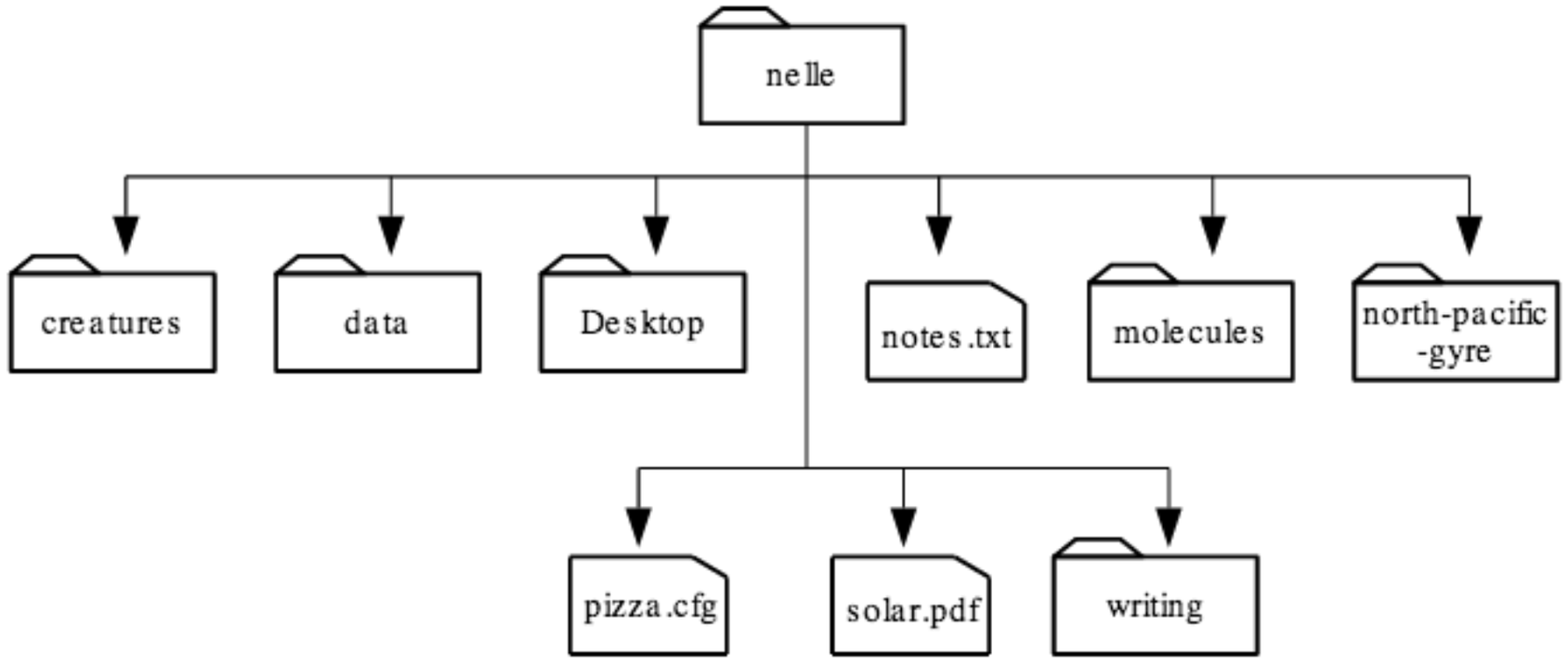
Command's switches are not universal e.g. **`-l`** means something else in **`ls`** and **`gcc`**

> Files and directories _

```
$
$ whoami
$ pwd
$ ls
```








```
$ ls -F
$ ls -a
$ ls -l
$ ls -l -h / or $ ls -lh
```

Path

Notice that there are two meanings for the / character. When it appears at the front of a file or directory name, it refers to the root directory. When it appears inside a name, it's just a separator.

Names

Names are "something dot something". This is just a convention: we can call a file **`mythesis`** or almost anything else we want. However, most people use two-part names most of the time to help them (and their programs) tell different kinds of files apart. The second part of such a name is called the filename extension, and indicates what type of data the file holds: **`.txt`** signals a plain text file, **`.pdf`** indicates a PDF document, .cfg is a configuration file full of parameters for some program or other, and so on.

Two special places are called **"."** and **".."**

"." - current directory
".." - one directory higher in the hierarchy of directories

Relative path vs. Absolute (full) path

relative - path to a file/directory starting from the "." (./job/cv.pdf) - "./" is often skipped

absolute - path to a file/directory starting from the top directory "/" (/home/joe/job/cv.pdf)

```
$ pwd
$ cd ..
$ cd ../..
$ cd -
$ cd
$ file example.dat
```

There are only several directories where a regular user (not an administrator) can write files. 1) home directory, 2) /tmp directory. There may be special places except them but those are not default locations.

You can use a **<tab>** key to autocomplete paths after typing several first characters.

Use arrow up and down keys to access commands that you have recently used.

> Creating things _

```
$ mkdir thesis
$ mkdir my_thesis
```
but not
```
$ mkdir my thesis
```

Unix shell is case sensitive (small/capital letters are different characters).

Unix shell does not accept “ “ (space) in names unless you use “\ “ as true space character. It is a good practice to use “_” or “-“ instead of a space in names.

```
$ mkdir -p thesis/drafts/first
```

-p allows for creating parent directories if they do not exist yet

```
$ cd thesis
$ nano draft.txt
```

When we say, “nano is a text editor,” we really do mean “text”: it can only work with plain character data, not tables, images, or any other human-friendly media.

Anyone can drive it anywhere without training, but please use something more powerful for real work.

On Unix systems (such as Linux and Mac OS X), many programmers use **Emacs** or **Vim** (both of which are completely unintuitive, even by Unix standards), or a graphical editor such as **Gedit**. On Windows, you may wish to use **Notepad++**.

GNU nano 2.0.6 File: draft.txt

^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where Is ^V Next Page ^U UnCut Text ^T To Spell

```
$ cd ..
$ ls
$ rm draft.txt
```

If you remove something there is no easy way of “undoing” it. There is no “Thrash” folder. Things are removed forever.

However, this is not the whole truth. If you remove something VERY important there are options to bring it back - VERY EXPENSIVE OPTIONS.

```
$ touch draft.txt
$ cd ..
$ rm thesis
$ rmdir thesis
$ rm thesis/draft.txt
$ rmdir thesis
```

```
$ mkdir thesis
$ cd thesis
$ nano draft.txt
$ cd ..
$ rm -r thesis
```

`-r` with `rm` means “recursively” (this and everything inside). Be careful with this as it will truly remove everything inside `thesis`

```
$ mv thesis/draft.txt thesis/quotes.txt
```

The general structure is `$ mv source destination`

where source is path to what you want to move and destination is its final path (path = relative or full path)

Just `mv` does not work for directories, use it with `-r` i.e `mv -r`

```
$ cp quotes.txt thesis/quotations.txt
$ ls quotes.txt thesis/quotations.txt
```

The **`cp`** command works very much like **`mv`**, except it copies a file instead of moving it. We can check that it did the right thing using ls with two paths as parameters — like most Unix commands.

To copy a directory again use **`-r`** options to copy the directory with its content, i.e. **`cp -r`**

The shell interprets the character ~ (tilde) at the start of a path to mean “the current user’s home directory”. For example, if Nelle’s home directory is **`/Users/nelle`**, then **`~/data`** is equivalent to **`/Users/nelle/data`**. This only works if it is the first character in the path.

Shell uses variables to store information. A variable starts with `$` character. There are many variables defined for you in all shell instances. Two are important

`$HOME`

`$PATH`

`$HOME` **–** defines the path to your home directory (similar to ~)

`$PATH` **–** defines the places where shell looks for programs

Command: **`echo`**, **`export`** and **`unset`** allow for manipulation

```
$ echo $PATH
$ export SC=“Software Carpentry”
$ echo $SC
$ export SC=$SC" is today”
$ echo $SC
$ unset SC
```

`echo` - print value
`export` - define variable
`export SC=$SC…` - append the value
`unset` - clear the variable

> Pipes and filters _

```
$ ls molecules
cubane.pdb    ethane.pdb    methane.pdb
octane.pdb    pentane.pdb   propane.pdb
```

```
$ cd molecules
$ wc *.pdb
  20  156 1158 cubane.pdb
  12   84  622 ethane.pdb
   9   57  422 methane.pdb
  30  246 1828 octane.pdb
  21  165 1226 pentane.pdb
  15  111  825 propane.pdb
 107  819 6081 total
```

`*` is a wildcard. It matches zero or more characters, so **`*.pdb`** matches **`ethane.pdb`, `propane.pdb`**, and every file that ends with **`'.pdb'`**. On the other hand, **`p*.pdb`** only matches **`pentane.pdb`** and **`propane.pdb`**, because the **`'p'`** at the front only matches filenames that begin with the letter **`'p'`**.

`?` is also a wildcard, but it only matches a single character. This means that **`p?.pdb`** matches **`pi.pdb`** or **`p5.pdb`**, but not **`propane.pdb`**.

```
$ wc -l *.pdb
  20  cubane.pdb
  12  ethane.pdb
   9  methane.pdb
  30  octane.pdb
  21  pentane.pdb
  15  propane.pdb
 107  total
```

```
$ wc -l *.pdb > lengths.txt
```

```
$ ls lengths.txt
$ cat lengths.txt
```

```
$ sort -n lengths.txt
  9  methane.pdb
 12  ethane.pdb
 15  propane.pdb
 20  cubane.pdb
 21  pentane.pdb
 30  octane.pdb
107  total
```

```
$ sort -n lengths.txt > sorted-lengths.txt
$ head -1 sorted-lengths.txt
  9  methane.pdb
```

Redirection to files

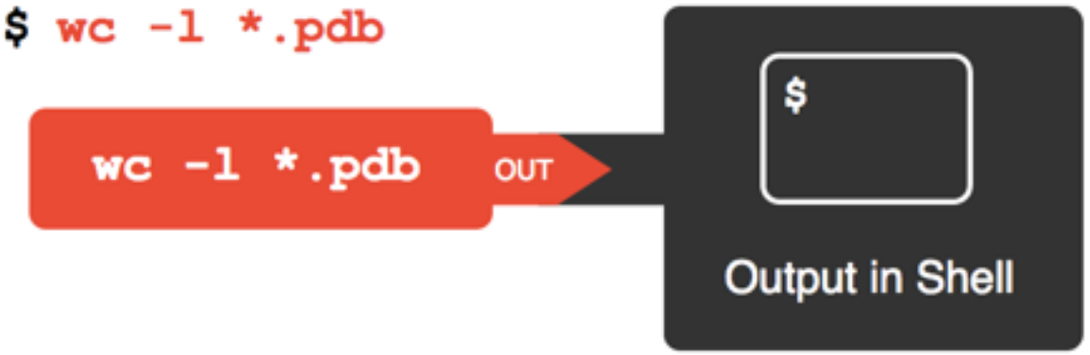
Redirection to another command

```
$ sort -n lengths.txt | head -1
  9  methane.pdb
```

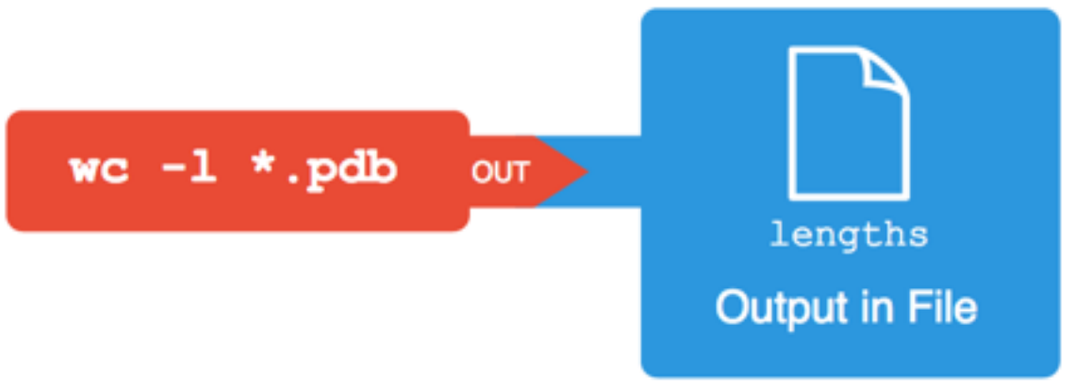
The vertical bar between the two commands is called a **pipe**. It tells the shell that we want to use the output of the command on the left as the input to the command on the right. The computer might create a temporary file if it needs to, or copy data from one program to the other in memory, or something else entirely; we don't have to know or care.

```
$ wc -l *.pdb | sort -n | head -1
  9  methane.pdb
```

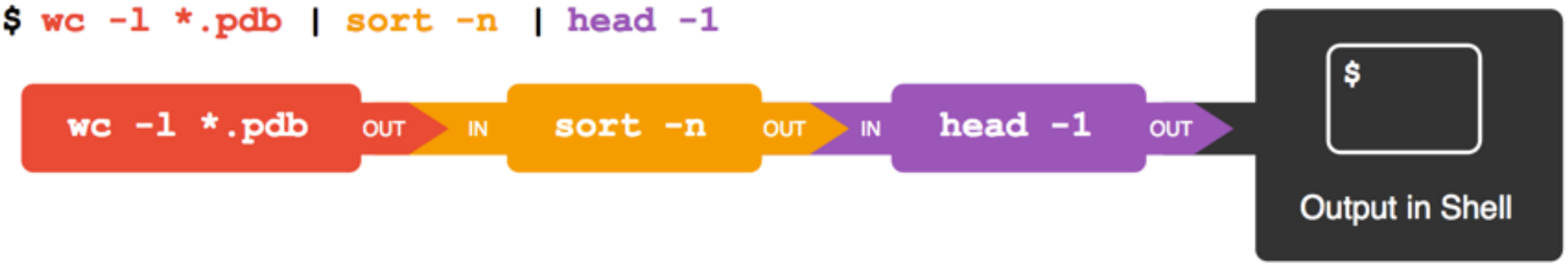
```
$ wc -l *.pdb
```




```
$ wc -l *.pdb > lengths
```




```
$ wc -l *.pdb | sort -n | head -1
```



Redirection the other way <

```
$ wc < methane.pdb
```

Appending operator >>

```
$ wc -l methane.pdb > results.txt
$ wc -l ethane.pdb >> results.txt
$ cat results.txt
```

```
$ cd north-pacific-gyre/2012-07-03
$ wc -l *.txt

300 NENE01729A.txt
300 NENE01729B.txt
300 NENE01736A.txt
300 NENE01751A.txt
300 NENE01751B.txt
300 NENE01812A.txt
... ...
```

```
$ wc -l *.txt | sort -n | head -5
     240 NENE02018B.txt
     300 NENE01729A.txt
     300 NENE01729B.txt
     300 NENE01736A.txt
     300 NENE01751A.txt
```

`uniq` command

coho
coho
steelhead
coho
steelhead
steelhead

then **`$ uniq salmon.txt`** produces:

coho
steelhead
coho
steelhead

> Loops _

```
$ cd creatures
$ ls
```

We have two files, i.e. **`basilisk.dat`** and **`unicorn.dat`** and we would like a backup copy with names **`original-basilisk.dat`** and **`original-unicorn.dat`**

How to proceed?

```
$ mv basilisk.dat unicorn.dat original-*.dat
```

This wouldn't back up our files, instead we get an error:

```
mv: target `original-*.dat' is not a directory
```

We can use loops for repetitive tasks

```
$ for filename in basilisk.dat unicorn.dat
> do
>     head -3 $filename
> done
```

When the shell sees the keyword for, it knows it is supposed to repeat a command (or group of commands) once for each thing in a list. In this case, the list is the two filenames.

Each time through the loop, the name of the thing currently being operated on is assigned to the variable called filename. Inside the loop, we get the variable's value by putting $ in front of it: $filename is basilisk.dat the first time through the loop, unicorn.dat the second, and so on.

When using variables it is also possible to put the names into curly braces to clearly delimit the variable name: **`$filename`** is equivalent to **`${filename}`**

The variable name does not matter

```
for x in basilisk.dat unicorn.dat
do
    head -3 $x
done
```

```
for temperature in basilisk.dat unicorn.dat
do
    head -3 $temperature
done
```

```
for filename in *.dat
do
    echo $filename
    head -100 $filename | tail -20
done
```

```
for filename in *.dat
do
    $filename
    head -100 $filename | tail -20
done
```

Going back to the original problem

```
for filename in *.dat
do
    mv $filename original-$filename
done
```

effectively we did this

```
mv basilisk.dat original-basilisk.dat
mv unicorn.dat original-unicorn.dat
```

Be on the safe side of things

A loop is a way to do many things at once — or to make many mistakes at once if it does the wrong thing. One way to check what a loop would do is to echo the commands it would run instead of actually running them.

```
for filename in *.dat
do
    echo mv $filename original-$filename
done
```

Instead of running `mv`, this loop runs echo, which prints out:

```
mv basilisk.dat original-basilisk.dat
mv unicorn.dat original-unicorn.dat
```

Processing many files

```
$ cd north-pacific-gyre/2012-07-03
$ for datafile in *[AB].txt
> do
>     echo $datafile
> done
```

```
$ for datafile in *[AB].txt
> do
>     echo $datafile stats-$datafile
> done
```

```
$ for datafile in *[AB].txt
> do
>     echo $datafile stats-$datafile
> done
```

is equivalent to

```
$ for datafile in *[AB].txt; do echo $datafile
    stats-$datafile; done
```

```
$ for datafile in *[AB].txt; do bash goostats
    $datafile stats-$datafile; done
```

```
  $ for datafile in *[AB].txt; do echo
$datafile; bash goostats $datafile stats-
    $datafile; done
```

> Shell scripts _

For historical reasons, a bunch of commands saved in a file is usually called a shell script, but make no mistake: these are actually small programs.

```
$ cd molecules
$ nano middle.sh
head -15 octane.pdb | tail -5
```

```
$ bash middle.sh
ATOM      9  H                1      -4.502    0.681    0.785  1.00  0.00
ATOM     10  H                1      -5.254   -0.243   -0.537  1.00  0.00
ATOM     11  H                1      -4.357    1.252   -0.895  1.00  0.00
ATOM     12  H                1      -3.009   -0.741   -1.467  1.00  0.00
ATOM     13  H                1      -3.172   -1.337    0.206  1.00  0.00
```

What if we want to select lines from an arbitrary file?

We could edit middle.sh each time to change the filename, but that would probably take longer than just retyping the command. Instead, let's edit middle.sh and replace **`octane.pdb`** with a special variable called **`$1`**:

```
$ cat middle.sh
head -15 "$1" | tail -5
```

```
$ bash middle.sh octane.pdb
ATOM      9  H                1      -4.502    0.681    0.785  1.00  0.00
ATOM     10  H                1      -5.254   -0.243   -0.537  1.00  0.00
ATOM     11  H                1      -4.357    1.252   -0.895  1.00  0.00
ATOM     12  H                1      -3.009   -0.741   -1.467  1.00  0.00
ATOM     13  H                1      -3.172   -1.337    0.206  1.00  0.00
```

```
$ bash middle.sh pentane.pdb
ATOM      9  H                1       1.324    0.350   -1.332  1.00  0.00
ATOM     10  H                1       1.271    1.378    0.122  1.00  0.00
ATOM     11  H                1      -0.074   -0.384    1.288  1.00  0.00
ATOM     12  H                1      -0.048   -1.362   -0.205  1.00  0.00
ATOM     13  H                1      -1.183    0.500   -1.412  1.00  0.00
```

Double-Quotes Around Arguments

We put the `$1` inside of double-quotes in case the filename happens to contain any spaces. The shell uses whitespace to separate arguments, so we have to be careful when using arguments that might have whitespace in them. If we left out these quotes, and `$1` expanded to a filename like **`methyl butane.pdb`**, the command in the script would effectively be:

```
head -15 methyl butane.pdb | tail -5
```

This would call head on two separate files, **`methyl`** and **`butane.pdb`**, which is probably not what we intended.

```
$ cat middle.sh
head "$2" "$1" | tail “$3"

$ bash middle.sh pentane.pdb -20 -5
ATOM     14  H           1      -1.259    1.420    0.112  1.00  0.00
ATOM     15  H           1      -2.608   -0.407    1.130  1.00  0.00
ATOM     16  H           1      -2.540   -1.303   -0.404  1.00  0.00
ATOM     17  H           1      -3.393    0.254   -0.321  1.00  0.00
TER      18              1
```

After some time we will forget how to use it, so lets document

```
$ cat middle.sh
# Select lines from the middle of a file.
# Usage: middle.sh filename -end_line -
num_lines
head "$2" "$1" | tail "$3"
```

What if we want to process many files in a single pipeline?

```
$ wc -l *.pdb | sort -n
```

We can't use `$1, $2`, and so on because we don't know how many files there are. Instead, we use the special variable `$@`, which means, "All of the command-line parameters to the shell script."

```
$ cat sorted.sh
wc -l "$@" | sort -n

$ bash sorted.sh *.pdb ../creatures/*.dat
9 methane.pdb
12 ethane.pdb
15 propane.pdb
20 cubane.pdb
21 pentane.pdb
30 octane.pdb
163 ../creatures/basilisk.dat
163 ../creatures/unicorn.dat
```

Why Isn't It Doing Anything?

What happens if a script is supposed to process a bunch of files, but we don't give it any filenames? For example, what if we type:

```
$ bash sorted.sh
```

but don't say `*.dat` (or anything else)? In this case, `$@` expands to nothing at all, so the pipeline inside the script is effectively:

```
wc -l | sort -n
```

Since it doesn't have any filenames, `wc` assumes it is supposed to process standard input, so it just sits there and waits for us to give it some data interactively. From the outside, though, all we see is it sitting there: the script doesn't appear to do anything.

Suppose we have just run a series of commands that did something useful — for example, that created a graph we'd like to use in a paper. We'd like to be able to re-create the graph later if we need to, so we want to save the commands in a file. Instead of typing them in again (and potentially getting them wrong) we can do this:

```
$ history | tail -4 > redo-figure-3.sh
```

The file **`redo-figure-3.sh`** now contains:

```
297 bash goostats -r NENE01729B.txt stats-NENE01729B.txt
298 bash goodiff stats-NENE01729B.txt /data/validated/01729.txt > 01729-differences.txt
299 cut -d ',' -f 2-3 01729-differences.txt > 01729-time-series.txt
300 ygraph --format scatter --color bw --borders none 01729-time-series.txt figure-3.png
```

Un-numbering

Use **`colrm`** (short for “column removal”) to remove the serial numbers on her previous commands. Its parameters are the range of characters to strip from its input:

```
$ history | tail -5
  173  cd /tmp
  174  ls
  175  mkdir bakup
  176  mv bakup backup
  177  history | tail -5
$ history | tail -5 | colrm 1 7
cd /tmp
ls
mkdir bakup
mv bakup backup
history | tail -5
history | tail -5 | colrm 1 7
```

> Finding things _

`grep` finds and prints lines in files that match a pattern. For our examples, we will use a file that contains three haikus taken from a 1998 competition in Salon magazine. For this set of examples we're going to be working in the writing subdirectory:

```
$ cd
$ cd writing
$ cat haiku.txt
The Tao that is seen
Is not the true Tao, until
You bring fresh toner.

With searching comes loss
and the presence of absence:
"My Thesis" not found.

Yesterday it worked
Today it is not working
Software is like that.
```

```
$ grep not haiku.txt
Is not the true Tao, until
"My Thesis" not found
Today it is not working
```

```
$ grep day haiku.txt
Yesterday it worked
Today it is not working
```

To restrict matches to lines containing the word “day” on its own, we can give grep with the **`-w`** flag. This will limit matches to word boundaries.

```
$ grep -w day haiku.txt
```

```
$ grep -w "is not" haiku.txt
Today it is not working
```

```
$ grep -n "it" haiku.txt
5:With searching comes loss
9:Yesterday it worked
10:Today it is not working
```

```
$ grep -n -w "the" haiku.txt
2:Is not the true Tao, until
6:and the presence of absence:
```

```
$ grep -n -w -i "the" haiku.txt
1:The Tao that is seen
2:Is not the true Tao, until
6:and the presence of absence:
```

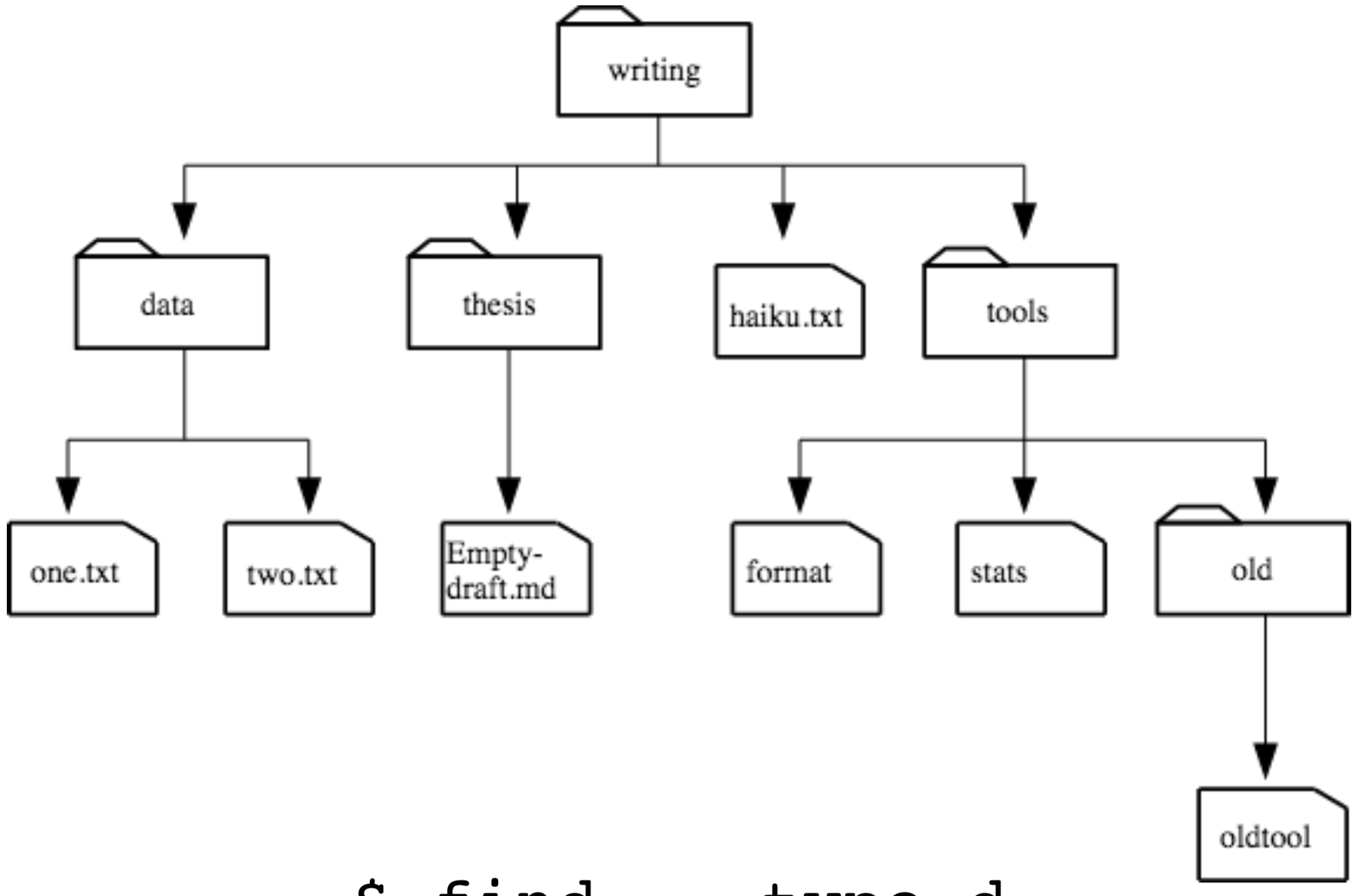
```
$ grep -n -w -v "the" haiku.txt
1:The Tao that is seen
3:You bring fresh toner.
4:
5:With searching comes loss
7:"My Thesis" not found.
8:
9:Yesterday it worked
10:Today it is not working
11:Software is like that.
```

Flavor of regular expressions

```
$ grep -E '^.o' haiku.txt
You bring fresh toner.
Today it is not working
Software is like that.
```

```
$ grep -E "ing$" haiku.txt
Today it is not working
```

`find` command finds files themselves. Again, it has a lot of options; to show how the simplest ones work, we'll use the directory tree shown below.




```
$ find . -type d
```

```
$ find . -type f
$ find . -maxdepth 1 -type f
$ find . -mindepth 2 -type f
$ find . -name *.txt
$ find . -name haiku.txt
$ find . -name '*.txt'
```

Listing vs. Finding

ls and find can be made to do similar things given the right options, but under normal circumstances, ls lists everything it can, while find searches for things with certain properties and shows them.

```
$ wc -l $(find . -name '*.txt')
      11 ./haiku.txt
     300 ./data/two.txt
      70 ./data/one.txt
     381 total
```

`$(...)` creates a variable with the output of a command which was put inside of ()

```
$ grep "FE" $(find .. -name '*.pdb')
../data/pdb/heme.pdb:ATOM     25 FE           1      -0.924   0.535  -0.518
```