Navigating Files and Directories

Exploring More `ls` Flags

You can also use two options at the same time. What does the command `ls` do when used with the `-l` option? What about if you use both the `-l` and the `-h` option?

Some of its output is about properties that we do not cover in this lesson (such as file permissions and ownership), but the rest should be useful nevertheless.

Listing in Reverse Chronological Order

By default, `ls` lists the contents of a directory in alphabetical order by name. The command `ls -t` lists items by time of last change instead of alphabetically. The command `ls -r` lists the contents of a directory in reverse order. Which file is displayed last when you combine the `-t` and `-r` flags? Hint: You may need to use the `-l` flag to see the last changed dates.

Absolute vs Relative Paths

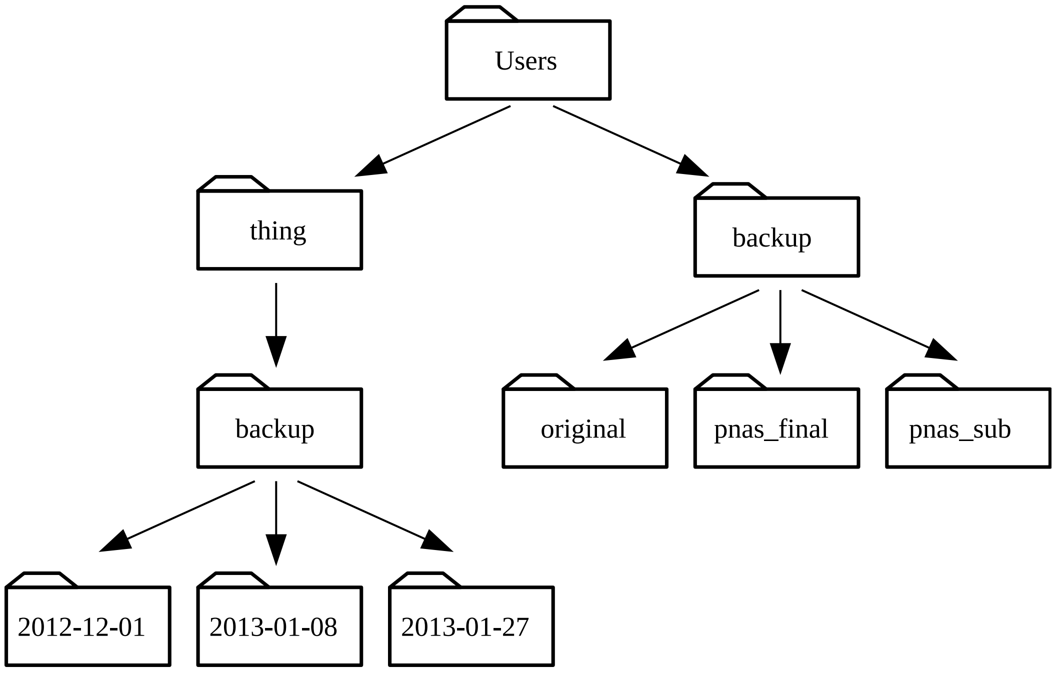
Starting from `/Users/amanda/data`, which of the following commands could Amanda use to navigate to her home directory, which is `/Users/amanda`?

1. `cd .`
2. `cd /`
3. `cd /home/amanda`
4. `cd ../..`
5. `cd ~`
6. `cd home`
7. `cd ~/data/..`
8. `cd`
9. `cd ..`

Relative Path Resolution

Using the filesystem diagram below, if `pwd` displays `/Users/thing`, what will `ls -F ../backup` display?

1. `../backup: No such file or directory`
2. `2012-12-01 2013-01-08 2013-01-27`
3. `2012-12-01/ 2013-01-08/ 2013-01-27/`
4. `original/ pnas_final/ pnas_sub/`

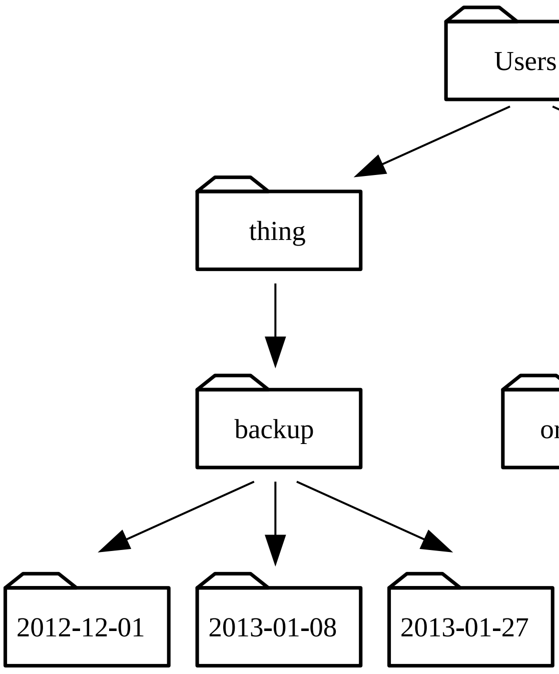


`ls` Reading Comprehension

Using the filesystem diagram below, if `pwd` displays `/Users/backup`, and `-r` tells `ls` to display things in reverse order, what command(s) will result in the following output:

Output

```
pnas_sub/ pnas_final/ original/
```



1. `ls pwd`
2. `ls -r -F`
3. `ls -r -F /Users/backup`

Working With Files and Directories

Creating Files a Different Way

We have seen how to create text files using the `nano` editor. Now, try the following command:

Bash

```
$ touch my_file.txt
```

1. What did the `touch` command do? When you look at your current directory using the GUI file explorer, does the file show up?
2. Use `ls -l` to inspect the files. How large is `my_file.txt` ?
3. When might you want to create a file this way?

Moving Files to a new folder

After running the following commands, Jamie realizes that she put the files `sucrose.dat` and `maltose.dat` into the wrong folder. The files should have been placed in the `raw` folder.

Bash

```
$ ls -F
 analyzed/ raw/
$ ls -F analyzed
fructose.dat glucose.dat maltose.dat sucrose.dat
$ cd analyzed
```

Fill in the blanks to move these files to the `raw/` folder (i.e. the one she forgot to put them in)

Bash

```
$ mv sucrose.dat maltose.dat ____/____
```

Renaming Files

Suppose that you created a plain-text file in your current directory to contain a list of the statistical tests you will need to do to analyze your data, and named it: `statstics.txt`

After creating and saving this file you realize you misspelled the filename! You want to correct the mistake, which of the following commands could you use to do so?

1. `cp statstics.txt statistics.txt`
2. `mv statstics.txt statistics.txt`
3. `mv statstics.txt .`
4. `cp statstics.txt .`

Moving and Copying

What is the output of the closing `ls` command in the sequence shown below?

Bash

```bash
$ pwd
```

Output

```
/Users/jamie/data
```

Bash

```bash
$ ls
```

Output

```
proteins.dat
```

Bash

```bash
$ mkdir recombined
$ mv proteins.dat recombined/
$ cp recombined/proteins.dat ../proteins-saved.dat
$ ls
```

1. `proteins-saved.dat recombined`
2. `recombined`
3. `proteins.dat recombined`
4. `proteins-saved.dat`

More on Wildcards

Sam has a directory containing calibration data, datasets, and descriptions of the datasets:

Bash

```
.
├── 2015-10-23-calibration.txt
├── 2015-10-23-dataset1.txt
├── 2015-10-23-dataset2.txt
├── 2015-10-23-dataset_overview.txt
├── 2015-10-26-calibration.txt
├── 2015-10-26-dataset1.txt
├── 2015-10-26-dataset2.txt
├── 2015-10-26-dataset_overview.txt
├── 2015-11-23-calibration.txt
├── 2015-11-23-dataset1.txt
├── 2015-11-23-dataset2.txt
├── 2015-11-23-dataset_overview.txt
├── backup
│   ├── calibration
│   └── datasets
└── send_to_bob
    ├── all_datasets_created_on_a_23rd
    └── all_november_files
```

Before heading off to another field trip, she wants to back up her data and send some datasets to her colleague Bob. Sam uses the following commands to get the job done:

Bash

```
$ cp *dataset* backup/datasets
$ cp ____calibration____ backup/calibration
$ cp 2015-____-____ send_to_bob/all_november_files/
$ cp ____ send_to_bob/all_datasets_created_on_a_23rd/
```

Help Sam by filling in the blanks.

The resulting directory structure should look like this

Bash

```
.
├── 2015-10-23-calibration.txt
├── 2015-10-23-dataset1.txt
├── 2015-10-23-dataset2.txt
├── 2015-10-23-dataset_overview.txt
├── 2015-10-26-calibration.txt
├── 2015-10-26-dataset1.txt
├── 2015-10-26-dataset2.txt
├── 2015-10-26-dataset_overview.txt
├── 2015-11-23-calibration.txt
├── 2015-11-23-dataset1.txt
├── 2015-11-23-dataset2.txt
├── 2015-11-23-dataset_overview.txt
├── backup
│   ├── calibration
│   │   ├── 2015-10-23-calibration.txt
│   │   ├── 2015-10-26-calibration.txt
│   │   └── 2015-11-23-calibration.txt
│   └── datasets
│       ├── 2015-10-23-dataset1.txt
│       ├── 2015-10-23-dataset2.txt
│       ├── 2015-10-23-dataset_overview.txt
│       ├── 2015-10-26-dataset1.txt
│       ├── 2015-10-26-dataset2.txt
│       ├── 2015-10-26-dataset_overview.txt
│       ├── 2015-11-23-dataset1.txt
│       ├── 2015-11-23-dataset2.txt
│       └── 2015-11-23-dataset_overview.txt
└── send_to_bob
    ├── all_datasets_created_on_a_23rd
    │   ├── 2015-10-23-dataset1.txt
    │   ├── 2015-10-23-dataset2.txt
    │   ├── 2015-10-23-dataset_overview.txt
    │   ├── 2015-11-23-dataset1.txt
    │   ├── 2015-11-23-dataset2.txt
    │   └── 2015-11-23-dataset_overview.txt
    └── all_november_files
        ├── 2015-11-23-calibration.txt
        ├── 2015-11-23-dataset1.txt
        ├── 2015-11-23-dataset2.txt
        └── 2015-11-23-dataset_overview.txt
```

Organizing Directories and Files

Jamie is working on a project and she sees that her files aren’t very well organized:

Bash

```
$ ls -F
```

Output

```
analyzed/  fructose.dat    raw/   sucrose.dat
```

The `fructose.dat` and `sucrose.dat` files contain output from her data analysis. What command(s) covered in this lesson does she need to run so that the commands below will produce the output shown?

Bash

```
$ ls -F
```

Output

```
analyzed/   raw/
```

Bash

```
$ ls analyzed
```

Output

```
fructose.dat    sucrose.dat
```

Reproduce a folder structure

You're starting a new experiment and would like to duplicate the directory structure from your previous experiment so you can add new data.

Assume that the previous experiment is in a folder called '2016-05-18', which contains a `data` folder that in turn contains folders named `raw` and `processed` that contain data files. The goal is to copy the folder structure of the `2016-05-18-data` folder into a folder called `2016-05-20` so that your final directory structure looks like this:

```
2016-05-20/
└── data
    ├── processed
    └── raw
```

Which of the following set of commands would achieve this objective? What would the other commands do?

Bash

```bash
$ mkdir 2016-05-20
$ mkdir 2016-05-20/data
$ mkdir 2016-05-20/data/processed
$ mkdir 2016-05-20/data/raw
```

Bash

```bash
$ mkdir 2016-05-20
$ cd 2016-05-20
$ mkdir data
$ cd data
$ mkdir raw processed
```

Bash

```bash
$ mkdir 2016-05-20/data/raw
$ mkdir 2016-05-20/data/processed
```

Bash

```bash
$ mkdir -p 2016-05-20/data/raw
$ mkdir -p 2016-05-20/data/processed
```

Bash

```bash
$ mkdir 2016-05-20
$ cd 2016-05-20
$ mkdir data
$ mkdir raw processed
```

Pipes and Filters

What Does `sort -n` Do?

The file `shell-lesson-data/numbers.txt` (../shell-lesson-data/numbers.txt) contains the following lines:

Code

```
10
2
19
22
6
```

If we run `sort` on this file, the output is:

Output

```
10
19
2
22
6
```

If we run `sort -n` on the same file, we get this instead:

Output

```
2
6
10
19
22
```

Explain why `-n` has this effect.

What Does `>>` Mean?

We have seen the use of `>`, but there is a similar operator `>>` which works slightly differently. We'll learn about the differences between these two operators by printing some strings. We can use the `echo` command to print strings e.g.

Bash

```
$ echo The echo command prints text
```

Output

```
The echo command prints text
```

Now test the commands below to reveal the difference between the two operators:

Bash

```
$ echo hello > testfile01.txt
```

and:

Bash

```
$ echo hello >> testfile02.txt
```

Hint: Try executing each command twice in a row and then examining the output files.

Appending Data

We have already met the `head` command, which prints lines from the start of a file. `tail` is similar, but prints lines from the end of a file instead.

Consider the file `shell-lesson-data/data/animals.txt`. After these commands, select the answer that corresponds to the file `animals-subset.txt`:

Bash

```
$ head -n 3 animals.txt > animals-subset.txt
$ tail -n 2 animals.txt >> animals-subset.txt
```

1. The first three lines of `animals.txt`
2. The last two lines of `animals.txt`
3. The first three lines and the last two lines of `animals.txt`
4. The second and third lines of `animals.txt`

Piping Commands Together

In our current directory, we want to find the 3 files which have the least number of lines. Which command listed below would work?

1. `wc -l * > sort -n > head -n 3`
2. `wc -l * | sort -n | head -n 1-3`
3. `wc -l * | head -n 3 | sort -n`
4. `wc -l * | sort -n | head -n 3`

Pipe Reading Comprehension

A file called `animals.txt` contains the following data:

Code

```
2012-11-05,deer
2012-11-05,rabbit
2012-11-05,raccoon
2012-11-06,rabbit
2012-11-06,deer
2012-11-06,fox
2012-11-07,rabbit
2012-11-07,bear
```

What text passes through each of the pipes and the final redirect in the pipeline below?

Bash

```bash
$ cat animals.txt | head -n 5 | tail -n 3 | sort -r > final.txt
```

Pipe Construction

For the file `animals.txt` from the previous exercise, consider the following command:

Bash

```bash
$ cut -d , -f 2 animals.txt
```

The `cut` command is used to remove or 'cut out' certain sections of each line in the file, and `cut` expects the lines to be separated into columns by a Tab character. A character used in this way is a called a **delimiter**. In the example above we use the `-d` option to specify the comma as our delimiter character. We have also used the `-f` option to specify that we want to extract the second field (column). This gives the following output:

Output

```
deer
rabbit
raccoon
rabbit
deer
fox
rabbit
bear
```

The `uniq` command filters out adjacent matching lines in a file. How could you extend this pipeline (using `uniq` and another command) to find out what animals the file contains (without any duplicates in their names)?

Which Pipe?

The file `animals.txt` contains 8 lines of data formatted as follows:

Output

```
2012-11-05,deer
2012-11-05,rabbit
2012-11-05,raccoon
2012-11-06,rabbit
...
```

The `uniq` command has a `-c` option which gives a count of the number of times a line occurs in its input. Assuming your current directory is `shell-lesson-data/data/`, what command would you use to produce a table that shows the total count of each type of animal in the file?

1. `sort animals.txt | uniq -c`
2. `sort -t, -k2,2 animals.txt | uniq -c`
3. `cut -d, -f 2 animals.txt | uniq -c`
4. `cut -d, -f 2 animals.txt | sort | uniq -c`
5. `cut -d, -f 2 animals.txt | sort | uniq -c | wc -l`