‹ (http://swcarpentry.github.io/shell-novice/01-intro/)

Navigating Files and Directories

› (http://swcarpentry.github.io/shell-novice/03-create/)

Overview

Teaching: 15 min
Exercises: 0 min
Questions

- How can I move around on my computer?
- How can I see what files and directories I have?
- How can I specify the location of a file or directory on my computer?

Objectives

- Explain the similarities and differences between a file and a directory.
- Translate an absolute path into a relative path and vice versa.
- Construct absolute and relative paths that identify specific files and directories.
- Explain the steps in the shell's read-run-print cycle.
- Identify the actual command, flags, and filenames in a command-line call.
- Demonstrate the use of tab completion, and explain its advantages.

The part of the operating system responsible for managing files and directories is called the **file system**. It organizes our data into files, which hold information, and directories (also called "folders"), which hold files or other directories.

Several commands are frequently used to create, inspect, rename, and delete files and directories. To start exploring them, let's open a shell window:

Preparation Magic

If you type the command: `PS1='$ '` into your shell, followed by pressing the 'enter' key, your window should look like our example in this lesson.
This isn't necessary to follow along (in fact, your prompt may have other helpful information you want to know about). This is up to you!

```
$
```

The dollar sign is a **prompt**, which shows us that the shell is waiting for input; your shell may use a different character as a prompt and may add information before the prompt. When typing commands, either from these lessons or from other sources, do not type the prompt, only the commands that follow it.

Type the command `whoami`, then press the Enter key (sometimes marked Return) to send the command to the shell. The command's output is the ID of the current user, i.e., it shows us who the shell thinks we are:

```
$ whoami
```

```
nelle
```

More specifically, when we type `whoami` the shell:

1. finds a program called `whoami`,
2. runs that program,
3. displays that program's output, then
4. displays a new prompt to tell us that it's ready for more commands.

📌 Username Variation

In this lesson, we have used the username `nelle` (associated with our hypothetical scientist Nelle) in example input and output throughout.
However, when you type this lesson's commands on your computer, you should see and use something different, namely, the username associated with the user account on your computer. This username will be the output from `whoami`. In what follows, `nelle` should always be replaced by that username.

Next, let's find out where we are by running a command called `pwd` (which stands for "print working directory"). At any moment, our **current working directory** is our current default directory, i.e., the directory that the computer assumes we want to run commands in unless we explicitly specify something else. Here, the computer's response is `/Users/nelle`, which is Nelle's **home directory**:

```
$ pwd
```

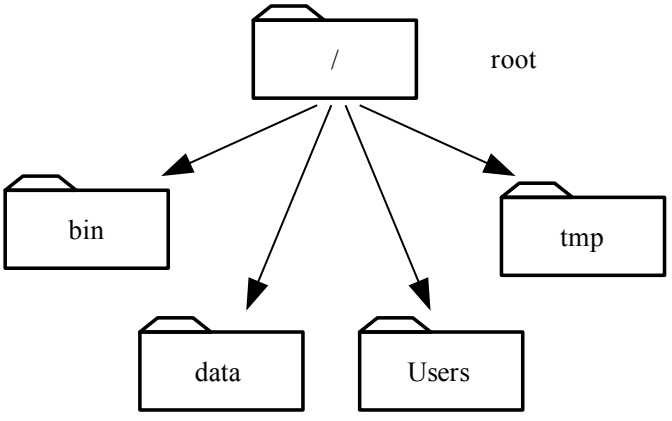
```
/Users/nelle
```

📌 Home Directory Variation

The home directory path will look different on different operating systems. On Linux it may look like `/home/nelle`, and on Windows it will be similar to `C:\Documents and Settings\nelle` or `C:\Users\nelle`. (Note that it may look slightly different for different versions of Windows.) In future examples, we've used Mac output as the default - Linux and Windows output may differ slightly, but should be generally similar.

To understand what a "home directory" is, let's have a look at how the file system as a whole is organized. For the sake of example, we'll be illustrating the filesystem on our scientist Nelle's computer. After this illustration, you'll be learning commands to explore your own filesystem, which will be constructed in a similar way, but not be exactly identical.

On Nelle's computer, the filesystem looks like this:



At the top is the **root directory** that holds everything else. We refer to it using a slash character `/` on its own; this is the leading slash in `/Users/nelle`.

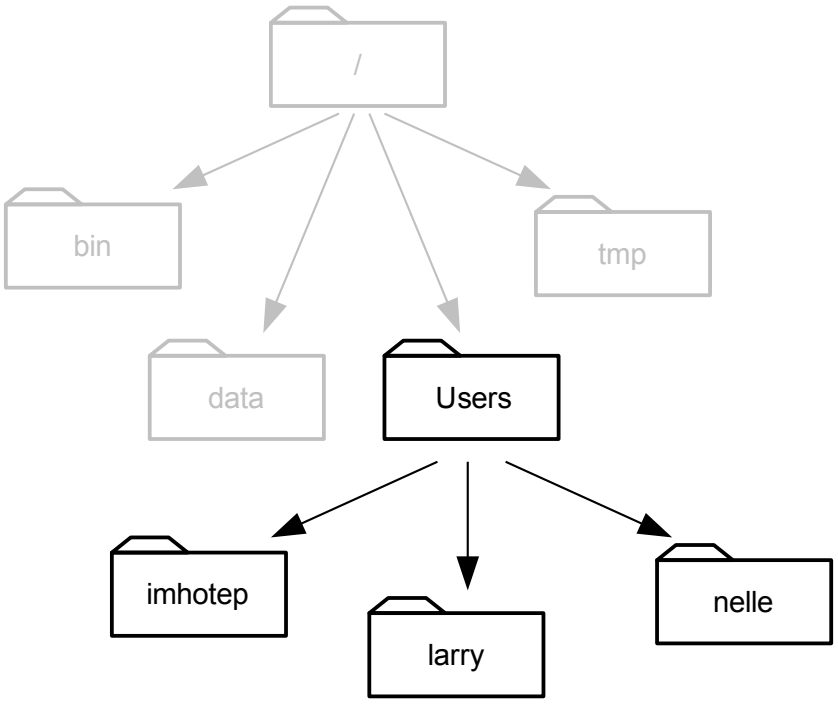
Inside that directory are several other directories: `bin` (which is where some built-in programs are stored), `data` (for miscellaneous data files), `Users` (where users' personal directories are located), `tmp` (for temporary files that don't need to be stored long-term), and so on.

We know that our current working directory `/Users/nelle` is stored inside `/Users` because `/Users` is the first part of its name. Similarly, we know that `/Users` is stored inside the root directory `/` because its name begins with `/`.

📌 Slashes

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.

Underneath `/Users`, we find one directory for each user with an account on Nelle's machine, her colleagues the Mummy and Wolfman.



The Mummy's files are stored in `/Users/imhotep`, Wolfman's in `/Users/larry`, and Nelle's in `/Users/nelle`. Because Nelle is the user in our examples here, this is why we get `/Users/nelle` as our home directory. Typically, when you open a new command prompt you will be in your home directory to start.

Now let's learn the command that will let us see the contents of our own filesystem. We can see what's in our home directory by running `ls`, which stands for "listing":

```
$ ls
```

```
Applications Documents    Library      Music        Public
Desktop      Downloads    Movies       Pictures
```

(Again, your results may be slightly different depending on your operating system and how you have customized your filesystem.)

`ls` prints the names of the files and directories in the current directory in alphabetical order, arranged neatly into columns. We can make its output more comprehensible by using the **flag** `-F`, which tells `ls` to add a trailing `/` to the names of directories:

```
$ ls -F
```

```
Applications/ Documents/    Library/      Music/        Public/
Desktop/      Downloads/    Movies/       Pictures/
```

`ls` has lots of other options. To find out what they are, we can type:

```
$ ls --help
```

```
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

Mandatory arguments to long options are mandatory for short options too.
  -a, --all                  do not ignore entries starting with .
  -A, --almost-all           do not list implied . and ..
      --author               with -l, print the author of each file
  -b, --escape               print C-style escapes for nongraphic characters
      --block-size=SIZE      scale sizes by SIZE before printing them; e.g.,
                               '--block-size=M' prints sizes in units of
                               1,048,576 bytes; see SIZE format below
  -B, --ignore-backups       do not list implied entries ending with ~
  -c                         with -lt: sort by, and show, ctime (time of last
                               modification of file status information);
                               with -l: show ctime and sort by name;
                               otherwise: sort by ctime, newest first
  -C                         list entries by columns
      --color[=WHEN]         colorize the output; WHEN can be 'always' (default
                               if omitted), 'auto', or 'never'; more info below
  -d, --directory            list directories themselves, not their contents
  -D, --dired                generate output designed for Emacs' dired mode
  -f                         do not sort, enable -aU, disable -ls --color
  -F, --classify             append indicator (one of */=>@|) to entries
      --file-type            likewise, except do not append '*'
      --format=WORD          across -x, commas -m, horizontal -x, long -l,
                               single-column -1, verbose -l, vertical -C
      --full-time            like -l --time-style=full-iso
  -g                         like -l, but do not list owner
      --group-directories-first
                             group directories before files;
                               can be augmented with a --sort option, but any
                               use of --sort=none (-U) disables grouping
  -G, --no-group             in a long listing, don't print group names
  -h, --human-readable       with -l and/or -s, print human readable sizes
                               (e.g., 1K 234M 2G)
      --si                   likewise, but use powers of 1000 not 1024
  -H, --dereference-command-line
                             follow symbolic links listed on the command line
      --dereference-command-line-symlink-to-dir
                             follow each command line symbolic link
                               that points to a directory
      --hide=PATTERN         do not list implied entries matching shell PATTERN
                               (overridden by -a or -A)
      --indicator-style=WORD  append indicator with style WORD to entry names:
                               none (default), slash (-p),
                               file-type (--file-type), classify (-F)
  -i, --inode                print the index number of each file
  -I, --ignore=PATTERN       do not list implied entries matching shell PATTERN
  -k, --kibibytes            default to 1024-byte blocks for disk usage
  -l                         use a long listing format
  -L, --dereference          when showing file information for a symbolic
                               link, show information for the file the link
                               references rather than for the link itself
  -m                         fill width with a comma separated list of entries
  -n, --numeric-uid-gid      like -l, but list numeric user and group IDs
  -N, --literal              print raw entry names (don't treat e.g. control
```

```
                             characters specially)
  -o                         like -l, but do not list group information
  -p, --indicator-style=slash
                             append / indicator to directories
  -q, --hide-control-chars   print ? instead of nongraphic characters
      --show-control-chars   show nongraphic characters as-is (the default,
                             unless program is 'ls' and output is a terminal)
  -Q, --quote-name           enclose entry names in double quotes
      --quoting-style=WORD   use quoting style WORD for entry names:
                             literal, locale, shell, shell-always,
                             shell-escape, shell-escape-always, c, escape
  -r, --reverse              reverse order while sorting
  -R, --recursive            list subdirectories recursively
  -s, --size                 print the allocated size of each file, in blocks
  -S                         sort by file size, largest first
      --sort=WORD            sort by WORD instead of name: none (-U), size (-S),
                             time (-t), version (-v), extension (-X)
      --time=WORD            with -l, show time as WORD instead of default
                             modification time: atime or access or use (-u);
                             ctime or status (-c); also use specified time
                             as sort key if --sort=time (newest first)
      --time-style=STYLE     with -l, show times using style STYLE:
                             full-iso, long-iso, iso, locale, or +FORMAT;
                             FORMAT is interpreted like in 'date'; if FORMAT
                             is FORMAT1<newline>FORMAT2, then FORMAT1 applies
                             to non-recent files and FORMAT2 to recent files;
                             if STYLE is prefixed with 'posix-', STYLE
                             takes effect only outside the POSIX locale
  -t                         sort by modification time, newest first
  -T, --tabsize=COLS         assume tab stops at each COLS instead of 8
  -u                         with -lt: sort by, and show, access time;
                             with -l: show access time and sort by name;
                             otherwise: sort by access time, newest first
  -U                         do not sort; list entries in directory order
  -v                         natural sort of (version) numbers within text
  -w, --width=COLS           set output width to COLS.  0 means no limit
  -x                         list entries by lines instead of by columns
  -X                         sort alphabetically by entry extension
  -Z, --context              print any security context of each file
  -1                         list one file per line.  Avoid '\n' with -q or -b
      --help     display this help and exit
      --version  output version information and exit

The SIZE argument is an integer and optional unit (example: 10K is 10*1024).
Units are K,M,G,T,P,E,Z,Y (powers of 1024) or KB,MB,... (powers of 1000).

Using color to distinguish file types is disabled both by default and
with --color=never.  With --color=auto, ls emits color codes only when
standard output is connected to a terminal.  The LS_COLORS environment
variable can change the settings.  Use the dircolors command to set it.

Exit status:
 0  if OK,
 1  if minor problems (e.g., cannot access subdirectory),
 2  if serious trouble (e.g., cannot access command-line argument).

GNU coreutils online help: <http://www.gnu.org/software/coreutils/>
```

```
Full documentation at: <http://www.gnu.org/software/coreutils/ls>
or available locally via: info '(coreutils) ls invocation'
```

Many bash commands, and programs that people have written that can be run from within bash, support a `--help` flag to display more information on how to use the commands or programs.

For more information on how to use `ls` we can type `man ls`. `man` is the Unix "manual" command: it prints a description of a command and its options, and (if you're lucky) provides a few examples of how to use it.

📌 `man` and Git for Windows

The bash shell provided by Git for Windows does not support the `man` command. Doing a web search for `unix man page COMMAND` (e.g. `unix man page grep`) provides links to numerous copies of the Unix manual pages online. For example, GNU provides links to its manuals (http://www.gnu.org/manual/manual.html): these include grep (http://www.gnu.org/software/grep/manual/), and the core GNU utilities (http://www.gnu.org/software/coreutils/manual/coreutils.html), which covers many commands introduced within this lesson.

To navigate through the `man` pages, you may use the up and down arrow keys to move line-by-line, or try the "b" and spacebar keys to skip up and down by full page. Quit the `man` pages by typing "q".

Here, we can see that our home directory contains mostly **sub-directories**. Any names in your output that don't have trailing slashes, are plain old **files**. And note that there is a space between `ls` and `-F`: without it, the shell thinks we're trying to run a command called `ls-F`, which doesn't exist.

📌 Parameters vs. Arguments

According to Wikipedia (https://en.wikipedia.org/wiki/Parameter_(computer_programming)#Parameters_and_arguments), the terms **argument** and **parameter** mean slightly different things. In practice, however, most people use them interchangeably or inconsistently, so we will too.

We can also use `ls` to see the contents of a different directory. Let's take a look at our `Desktop` directory by running `ls -F Desktop`, i.e., the command `ls` with the **arguments** `-F` and `Desktop`. The second argument — the one *without* a leading dash — tells `ls` that we want a listing of something other than our current working directory:

```
$ ls -F Desktop
```

```
data-shell/
```

Your output should be a list of all the files and sub-directories on your Desktop, including the `data-shell` directory you downloaded at the start of the lesson. Take a look at your Desktop to confirm that your output is accurate.

As you may now see, using a bash shell is strongly dependent on the idea that your files are organized in an hierarchical file system.
Organizing things hierarchically in this way helps us keep track of our work: it's possible to put hundreds of files in our home directory, just as it's possible to pile hundreds of printed papers on our desk, but it's a self-defeating strategy.

Now that we know the `data-shell` directory is located on our Desktop, we can do two things.

First, we can look at its contents, using the same strategy as before, passing a directory name to `ls`:

```
$ ls -F Desktop/data-shell
```

```
creatures/          molecules/          notes.txt           solar.pdf
data/               north-pacific-gyre/ pizza.cfg           writing/
```

Second, we can actually change our location to a different directory, so we are no longer located in our home directory.

The command to change locations is `cd` followed by a directory name to change our working directory. `cd` stands for “change directory”, which is a bit misleading: the command doesn’t change the directory, it changes the shell’s idea of what directory we are in.

Let’s say we want to move to the `data` directory we saw above. We can use the following series of commands to get there:

```
$ cd Desktop
$ cd data-shell
$ cd data
```

These commands will move us from our home directory onto our Desktop, then into the `data-shell` directory, then into the `data` directory. `cd` doesn’t print anything, but if we run `pwd` after it, we can see that we are now in `/Users/nelle/Desktop/data-shell/data`. If we run `ls` without arguments now, it lists the contents of `/Users/nelle/Desktop/data-shell/data`, because that’s where we now are:

```
$ pwd
```

```
/Users/nelle/Desktop/data-shell/data
```

```
$ ls -F
```

```
amino-acids.txt   elements/     pdb/            salmon.txt
animals.txt       morse.txt     planets.txt     sunspot.txt
```

We now know how to go down the directory tree: how do we go up? We might try the following:

```
cd data-shell
```

```
-bash: cd: data-shell: No such file or directory
```

But we get an error! Why is this?

With our methods so far, `cd` can only see sub-directories inside your current directory. There are different ways to see directories above your current location; we’ll start with the simplest.

There is a shortcut in the shell to move up one directory level that looks like this:

```
$ cd ..
```

`..` is a special directory name meaning “the directory containing this one”, or more succinctly, the **parent** of the current directory. Sure enough, if we run `pwd` after running `cd ..`, we’re back in `/Users/nelle/Desktop/data-shell`:

```
$ pwd
```

```
/Users/nelle/Desktop/data-shell
```

The special directory `..` doesn't usually show up when we run `ls`. If we want to display it, we can give `ls` the `-a` flag:

```
$ ls -F -a
```

```
./                  creatures/          notes.txt
../                 data/               pizza.cfg
.bash_profile       molecules/          solar.pdf
Desktop/            north-pacific-gyre/ writing/
```

`-a` stands for "show all"; it forces `ls` to show us file and directory names that begin with `.`, such as `..` (which, if we're in `/Users/nelle`, refers to the `/Users` directory) As you can see, it also displays another special directory that's just called `.`, which means "the current working directory". It may seem redundant to have a name for it, but we'll see some uses for it soon.

📌 Other Hidden Files

In addition to the hidden directories `..` and `.`, you may also see a file called `.bash_profile`. This file usually contains shell configuration settings. You may also see other files and directories beginning with `.`. These are usually files and directories that are used to configure different programs on your computer. The prefix `.` is used to prevent these configuration files from cluttering the terminal when a standard `ls` command is used.

📌 Orthogonality

The special names `.` and `..` don't belong to `ls`; they are interpreted the same way by every program. For example, if we are in `/Users/nelle/data`, the command `ls ..` will give us a listing of `/Users/nelle`. When the meanings of the parts are the same no matter how they're combined, programmers say they are **orthogonal**: Orthogonal systems tend to be easier for people to learn because there are fewer special cases and exceptions to keep track of.

These then, are the basic commands for navigating the filesystem on your computer: `pwd`, `ls` and `cd`. Let's explore some variations on those commands. What happens if you type `cd` on its own, without giving a directory?

```
$ cd
```

How can you check what happened? `pwd` gives us the answer!

```
$ pwd
```

```
/Users/nelle
```

It turns out that `cd` without an argument will return you to your home directory, which is great if you've gotten lost in your own filesystem.

Let's try returning to the `data` directory from before. Last time, we used three commands, but we can actually string together the list of directories to move to `data` in one step:

```
$ cd Desktop/data-shell/data
```

Check that we've moved to the right place by running `pwd` and `ls -F`.

If we want to move up one level from the shell directory, we could use `cd ..`. But there is another way to move to any directory, regardless of your current location.

So far, when specifying directory names, or even a directory path (as above), we have been using **relative paths**. When you use a relative path with a command like `ls` or `cd`, it tries to find that location from where we are, rather than from the root of the file system.

However, it is possible to specify the **absolute path** to a directory by including its entire path from the root directory, which is indicated by a leading slash. The leading `/` tells the computer to follow the path from the root of the file system, so it always refers to exactly one directory, no matter where we are when we run the command.

This allows us to move to our `data-shell` directory from anywhere on the filesystem (including from inside `data`). To find the absolute path we're looking for, we can use `pwd` and then extract the piece we need to move to `data-shell`.

```
$ pwd
```

```
/Users/nelle/Desktop/data-shell/data
```

```
$ cd /Users/nelle/Desktop/data-shell
```

Run `pwd` and `ls -F` to ensure that we're in the directory we expect.

📌 Two More Shortcuts

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: `here/there/~/elsewhere` is *not* `here/there/Users/nelle/elsewhere`.

Another shortcut is the `-` (dash) character. `cd` will translate `-` into *the previous directory I was in*, which is faster than having to remember, then type, the full path. This is a *very* efficient way of moving back and forth between directories. The difference between `cd ..` and `cd -` is that the former brings you *up*, while the latter brings you *back*.

Nelle's Pipeline: Organizing Files

Knowing just this much about files and directories, Nelle is ready to organize the files that the protein assay machine will create. First, she creates a directory called `north-pacific-gyre` (to remind herself where the data came from). Inside that, she creates a directory called `2012-07-03`, which is the date she started processing the samples. She used to use names like `conference-paper` and `revised-results`, but she found them hard to understand after a couple of years. (The final straw was when she found herself creating a directory called `revised-revised-results-3`.)

📌 Sorting Output

Nelle names her directories “year-month-day”, with leading zeroes for months and days, because the shell displays file and directory names in alphabetical order. If she used month names, December would come before July; if she didn’t use leading zeroes, November (‘11’) would come before July (‘7’). Similarly, putting the year first means that June 2012 will come before June 2013.

Each of her physical samples is labelled according to her lab’s convention with a unique ten-character ID, such as “NENE01729A”. This is what she used in her collection log to record the location, time, depth, and other characteristics of the sample, so she decides to use it as part of each data file’s name. Since the assay machine’s output is plain text, she will call her files `NENE01729A.txt`, `NENE01812A.txt`, and so on. All 1520 files will go into the same directory.

Now in her current directory `data-shell`, Nelle can see what files she has using the command:

```
$ ls north-pacific-gyre/2012-07-03/
```

This is a lot to type, but she can let the shell do most of the work through what is called **tab completion**. If she types:

```
$ ls nor
```

and then presses tab (the tab key on her keyboard), the shell automatically completes the directory name for her:

```
$ ls north-pacific-gyre/
```

If she presses tab again, Bash will add `2012-07-03/` to the command, since it’s the only possible completion. Pressing tab again does nothing, since there are 19 possibilities; pressing tab twice brings up a list of all the files, and so on. This is called **tab completion**, and we will see it in many other tools as we go on.

🖉 Absolute vs Relative Paths

🖉 Relative Path Resolution

🖉 `ls` Reading Comprehension

🖉 Exploring More `ls` Arguments

🖉 Listing Recursively and By Time

Key Points

- The file system is responsible for managing information on the disk.
- Information is stored in files, which are stored in directories (folders).
- Directories can also store other directories, which forms a directory tree.
- `cd path` changes the current working directory.
- `ls path` prints a listing of a specific file or directory; `ls` on its own lists the current working directory.
- `pwd` prints the user's current working directory.
- `whoami` shows the user's current identity.
- `/` on its own is the root directory of the whole file system.
- A relative path specifies a location starting from the current location.
- An absolute path specifies a location from the root of the file system.
- Directory names in a path are separated with '/' on Unix, but '\' on Windows.
- '..' means 'the directory above the current one'; '.' on its own means 'the current directory'.
- Most files' names are `something.extension`. The extension isn't required, and doesn't guarantee anything, but is normally used to indicate the type of data in the file.
- Most commands take options (flags) which begin with a '-'.
