title: The Unix Shell
subtitle: Files & Directories
minutes: 20

The Unix Shell: Files and Directories

Learning 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:

```
$
```

The dollar sign is a **prompt**, which shows us that the shell is waiting for input; your shell may show something more elaborate.

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
```

```
oski
```

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.

Bash Syntax

Note that when we use the shell, we're giving the computer instructions in code, like with other programming languages. Since the purpose here is to interact with the computer's file system, the **things** are usually files or directories, and the **actions** are usually commands we want to execute on those files/directories, or commands to launch other programs stored in the computer's file system.

The syntax supports that intent. The basic form of an instruction we enter is a line containing the following parts:

1. a command (the action)
2. optional flags to tailor the command to how we wish to use it

- indicated by a "-" followed by a letter or option name

3. files or directories (the things) we want to operate on

- a command may require two, e.g. a source and a destination if we're moving or copying files
- a command may allow an arbitrary list of files, if it can be run on many files at once
- a command may not require any, or may work on a default if none is specified, which we'll learn about next

Home Directories

To continue, 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 `/home/oski`, which is the **home directory**:

```
$ pwd
```

```
/home/oski
```

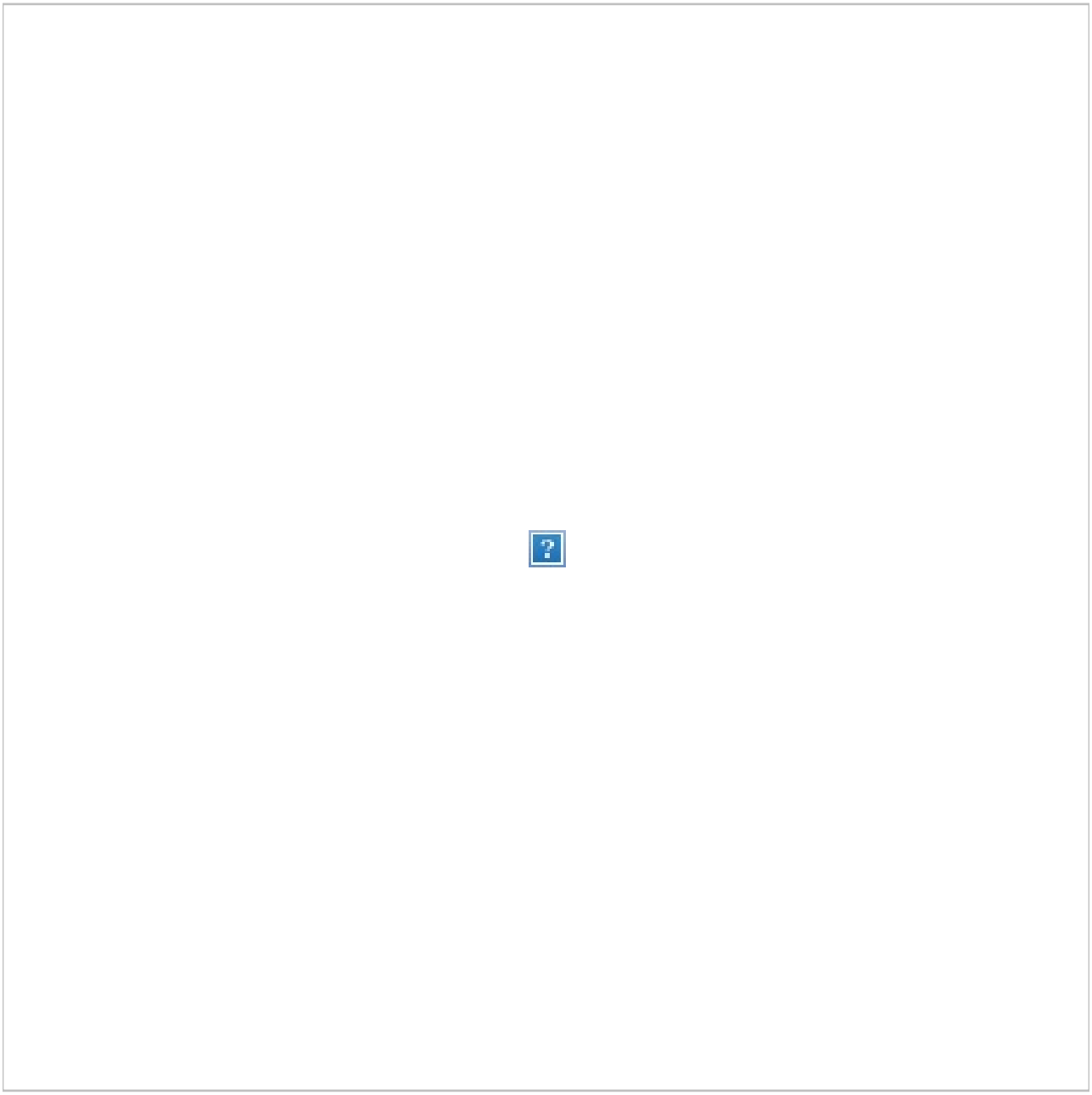
> **Alphabet Soup**
>
> If the command to find out who we are is `whoami`, the command to find out where we are ought to be called `whereami`, so why is it `pwd` instead? The usual answer is that in the early 1970s, when Unix was first being developed, every keystroke counted: the devices of the day were slow, and backspacing on a teletype was so painful that cutting the number of keystrokes in order to cut the number of typing mistakes was actually a win for usability. The reality is that commands were added to Unix one by one, without any master plan, by people who were immersed in its jargon. The result is as inconsistent as the roolz uv Inglish speling, but we're stuck with it now.
>
> The good news is: because these basic commands were so integral to the development of early Unix, they have stuck around, and appear (in some form) in almost all programming languages.

To understand what a "home directory" is, let's have a look at how the file system as a whole is organized. 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 /home/oski.

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

> If you're working on a Mac, the file structure will look similar, but not identical. The following image shows a file system graph for the typical Mac.



(url)

Notice that there is more than one folder called "Library", at different places in the tree. You can think of this like any hierarchical system with items located within larger ones. For instance, if you had data on U.S. cities, you might have more than one city called "Albany". So you'd probably

identify each city with its state (e.g. "Albany, NY" and "Albany, CA"), and maybe also the country ("U.S."), which might be the "root" of all your data. On the other hand, if you were just telling a friend that you're going up to Albany for lunch, you wouldn't say you're going to "Albany, CA, U.S.", because your friend knows you mean the Albany in this local area. We can specify locations in our file system in similar ways: starting from the root, or just from our current location.

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

> 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.

Listing

Let's see what's in Rochelle's home directory by running `ls`, which stands for "listing":

```
$ ls
```

```
Applications    Documents     Library    textfile.txt
bin             Downloads     Movies
Box Sync        Dropbox       Pictures
Desktop         Google Drive  Public
```

`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/    textfile.txt
bin/            Downloads/     Movies/
Box Sync/       Dropbox/       Pictures/
Desktop/        Google Drive/  Public/
```

Here,
we can see that home contains 12 **sub-directories**.
The names that don't have trailing slashes, such as textfile.txt 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. If you want to better understand a command and its flags, for example `ls`, you can type:

```
$ man ls
```

To quite, type *q*. If you are on Windows you'll need to type the Mac succinct version:

```
$ ls --help
```

What's In A Name?

You may have noticed that all of our's files' names are "something dot something". This is just a convention: we can call a file `file` 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.

This is just a convention, albeit an important one. Files contain bytes: it's up to us and our programs to interpret those bytes according to the rules for PDF documents, images, and so on.

Naming a PNG image of a whale as `whale.mp3` doesn't somehow magically turn it into a recording of whalesong, though it *might* cause the operating system to try to open it with a music player when someone double-clicks it.

Now let's take a look at what's in Rochelle's `Movies` directory by running `ls -F Movies`, i.e., the command `ls` with the **arguments** `-F` and `bin`. 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 Movies
```

```
Comedy/     Thriller/       Action/     movie_file.mov
```

The output shows us that there is one file and three sub-sub folders. 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.

Notice, by the way that we spelled the directory name `Movies`. It doesn't have a trailing slash: that's added to directory names by `ls` when we use the `-F` flag to help us tell things apart. And it doesn't begin with a slash because it's a **relative path**, i.e., it tells `ls` how to find something from where we are, rather than from the root of the file system.

Parameters vs. Arguments

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

To understand this try running `ls -F bin` and `ls -F /bin` (*with* a leading slash). You see we get different answers,
because `/bin` is an **absolute path**. You also conveniently see all the available bash commands:

```
$ ls -F /bin
```

```
[*          df*             launchctl*  pwd*        tcsh*
bash*       domainname*     link*       rcp*        test*
cat*        echo*           ln*         rm*         unlink*
chmod*      ed*             ls*         rmdir*      wait4path*
cp*         expr*           mkdir*      sh*         zsh*
csh*        hostname*       mv*         sleep*
date*       kill*           pax*        stty*
dd*         ksh*            ps*         sync*
```

Remember that explanation of how these commands call little programs to work with another? The programs are right here!

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.

What if we want to change our current working directory? Before we do this, pwd shows us that we're in /home/oski, and `ls` without any arguments shows us that directory's contents:

```
$ pwd
```

```
/home/oski
```

```
$ ls
```

```
Applications    Documents       Library     textfile.txt
bin             Downloads       Movies
Box Sync        Dropbox         Pictures
Desktop         Google Drive    Public
```

Downloading the Github Repository

We are now going to download the materials we'll use for the rest of the class. You can cd into your Desktop. We can use 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. To cd into Desktop we simply type:

```
$ cd Desktop
```

If you are using Windows and have downloaded Git Bash, or you know you have Git installed, you can get the materials by typing:

```
$ git clone https://github.com/dlab-berkeley/programming-fundamentals.git
```

Alternatively, go to the [repository (https://github.com/dlab-berkeley/programming-fundamentals)](https://github.com/dlab-berkeley/programming-fundamentals) in your browser and download the zip file to your desktop.

Once you cd into your Desktop and type ls you should now see the programming-fundamentals-master folder appear in your listing.

Moving Around

Let's go inside that directory:

```
$ cd programming-fundamentals-master
```

cd doesn't print anything, but if we run pwd after it, we can see that we are now in /home/oski/Desktop/programming-fundamentals-master.

If we run ls without arguments now, it lists the contents of /home/oski/Desktop/programming-fundamentals-master, because that's where we now are:

```
$ pwd
```

```
/home/oski/Desktop/programming-fundamentals-master
```

```
$ ls -F
```

```
0-0_Introduction.md  1-1_fildir.md  1-4_loop.md     2-0_help.md  madlib.py
0-1_BCE.md           1-2_create.md  1-5_scripts.md  data/        README.md
1-0_shell.md         1-3_pipe.md    1-6_python.md   LICENSE      resource.md
```

We now know how to go down the directory tree: how do we go up? We could use an absolute path:

```
$ cd /home/oski/Desktop
```

but it's almost always simpler to use cd .. to go up one level:

```
$ pwd
```

```
/home/oski/Desktop/programming-fundamentals-master
```

```
$ 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 /home/oski/Desktop:

```
$ pwd
```

```
/home/oski/Desktop
```

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 -a
```

This will then list all the files on our Desktop. -a stands for "show all"; it forces ls to show us file and directory names that begin with ., such as ...

> **Hidden Files: For Your Own Protection**
>
> As you can see, a bunch of other items just appeared when we enter `ls -a`. These files and directories begin with `.` followed by a name. These are usually files and directories that hold important programmatic information, not usually edited by the casual computer user. They are kept hidden so that users don't accidentally delete or edit them without knowing what they're doing.

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.

> **Phone Home**
>
> If you ever want to get to the home directory immediately, you can use the shortcut `~`. For example, type `cd ~` and you'll get back home in a jiffy. `~` will also stand in for your home directory in paths, so for instance `~/Desktop` is the same as `/home/oski/Desktop`. This only works if it is the first character in the path: `here/there/~/elsewhere` is not `/home/oski/elsewhere`.

Rochelle's Pipeline: Getting Ready

In order to start her text analysis project, Rochelle first has to figure out where her data is stored.

Everything Rochelle needs for her text project is in the `data` directory of the git repository (i.e. the directory) `programming-fundamentals`. So Rochelle will migrate there.

```
$ cd ~/Desktop/programming-fundamentals-master/data
$ ls
```

```
articles  downloads
```

Each of Rochelle's text files is labeled according to the parameters leading to her LexisNexis Search. Since she searched and downloaded articles containing the phrase 'human rights' for each year, she will call her files `human-rights-2001.txt`, `human-rights-2002.txt`, and so on. All files are in currently in the `downloads` directory.

```
$ cd downloads
$ ls
```

```
human-rights-2000.TXT  human-rights-2004.TXT  human-rights-2008.TXT
human-rights-2001.TXT  human-rights-2005.TXT  human-rights-2009.TXT
human-rights-2002.TXT  human-rights-2006.TXT
human-rights-2003.TXT  human-rights-2007.TXT
```

If she is in her home directory, Rochelle can see what files she has using the command:

```
$ cd ~
$ ls programming-fundamentals-master/data/downloads
```

This is a lot to type, but she can let the shell do most of the work. If she types:

```
$ ls prog
```

and then presses tab, the shell automatically completes the directory name for her:

```
$ ls programming-fundamentals-master/
```

Pressing tab again does nothing, since there are multiple possibilities. Pressing tab twice brings up a list of all the files and directories, and so on.

This is called **tab completion**, and we will see it in many other tools as we go on.

> **Quick File Paths**
>
> If you quickly need the path of a file or directory, you can also copy the file/directory in the GUI and paste it into your shell. The full path of the file or directory will appear.

Exercises

Write down, or think through, the answers to the following questions, then run the appropriate commands to check your answer.

Challenge 1

If pwd displays `/home/oski/Desktop/programming-fundamentals-master/data/articles`, what will `ls ../downloads` display?

1. no output
2. human-rights-2000.TXT human-rights-2004.TXT human-rights-2008.TXT human-rights-2001.TXT human-rights-2005.TXT human-rights-2009.TXT human-rights-2002.TXT human-rights-2006.TXT split_ln.py human-rights-2003.TXT human-rights-2007.TXT
3. animals.txt articles downloads
4. error

Challenge 2

If pwd displays `/home/oski/Desktop/programming-fundamentals-master`, and `-r` tells `ls` to display things in reverse order, what command will display:

```
test/                1-5_scripts.md       1-0_shell.md
resource.md          1-4_loop.md          0-2_help.md
data/                1-3_pipe.md          0-1_BCE.md
README.md            1-2_create.md        0-0_Introduction.md
LICENSE              1-1_fildir.md
```

1. `ls pwd`
2. `ls -r -F`
3. `ls -rF`
4. Either #2 or #3 above, but not #1.

Challenge 3

What does the command cd without a directory name do?

1. It has no effect.
2. It changes the working directory to /.
3. It changes the working directory to the user's home directory.
4. It produces an error message.

Challenge 4

What does the command `ls` do when used with the `-s` arguments? What about `-l`?

Adapted from: [Software Carpentry (http://software-carpentry.org/v5/novice/shell/01-filedir.html)]