Computational Methods for the Physical Sciences

Course material for PHYS2200 class

Storrs, October 20, 2016

Contents

Part I

Programming tools

Chapter 1

Version Control with Git

Wolfman and Dracula have been hired by Universal Missions (a space services spinoff from Euphoric State University) to investigate if it is possible to send their next planetary lander to Mars. They want to be able to work on the plans at the same time, but they have run into problems doing this in the past. If they take turns, each one will spend a lot of time waiting for the other to finish, but if they work on their own copies and email changes back and forth things will be lost, overwritten, or duplicated.

A colleague suggests using version control to manage their work. Version control is better than mailing files back and forth:

- Nothing that is committed to version control is ever lost. Since all old versions of files are saved, it's always possible to go back in time to see exactly who wrote what on a particular day, or what version of a program was used to generate a particular set of results.
- As we have this record of who made what changes when, we know who to ask if we have questions later on, and, if needed it, revert to a previous version, much like the "undo" feature in an editor.
- When several people collaborate in the same project, it's possible to accidentally overlook or overwrite someone's changes: the version control system automatically notifies users whenever there's a conflict between one person's work and another's.

Teams are not the only ones to benefit from version control: lone researchers can benefit immensely. Keeping a record of what was changed, when, and why is extremely useful for all researchers if they ever need to come back to the project later on (e.g., a year later, when memory has faded).

Version control is the lab notebook of the digital world: it's what professionals use to keep track of what they've done and to collaborate with other people. Every large software development project relies on it, and most programmers use it for their small jobs as well. And it isn't just for software: books, papers, small data sets, and anything that changes over time or needs to be shared can and should be stored in a version control system.

In this notes we use Git from the Unix Shell. Some previous experience with the shell is expected.

1.1 Basics

In this section we learn (a) the benefits of an automated version control system; (b) the basics of how Git works.

We'll start by exploring how version control can be used to keep track of what one person did and when. Even if you aren't collaborating with other people, automated version control is much better than having multiple nearly-identical versions of the same document.

Version control systems start with a base version of the document and then save just the changes you made at each step of the way. You can think of it as a tape: if you rewind the tape and start at the base document, then you can play back each change and end up with your latest version.

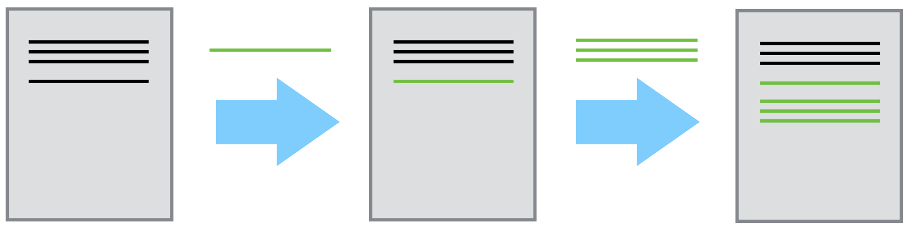


Figure 1.1: Changes are saved sequentially

Once you think of changes as separate from the document itself, you can then think about "playing back" different sets of changes onto the base document and getting different versions of the document. For example, two users can make independent sets of changes based on the same document (see Fig. **??**).

If there aren't conflicts, you can even try to play two sets of changes onto the same base document (see Fig. **??**).

A version control system is a tool that keeps track of these changes for us and helps us version and merge our files. It allows you to decide which changes make up the next version, called a *commit*, and keeps useful metadata about them. The complete history of commits for a particular project and their metadata make up a *repository*. Repositories can be kept in sync across different computers facilitating collaboration among different people.

Automated version control systems are nothing new. Early systems have been around since the 1980s. However modern systems, such as Git are *distributed*, meaning that they do not need a centralized server to host the repository. The modern systems also include powerful

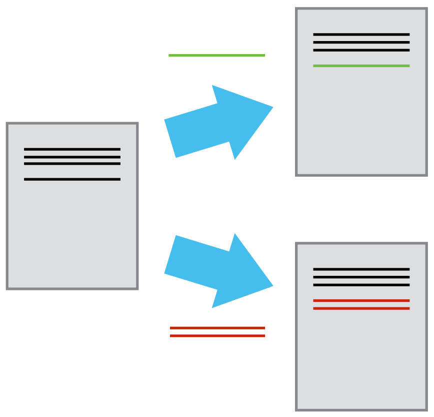


Figure 1.2: Different versions can be saved.

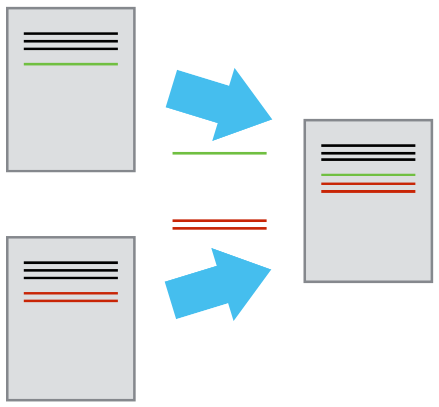


Figure 1.3: Multiple versions can be merged.

merging tools that make it possible for multiple authors to work within the same files concurrently.

1.2 Setting Up Git

In this section we (a) configure git the first time it is used on a computer, and (b) understand the meaning of the `--global` configuration flag.

When we use Git on a new computer for the first time, we need to configure a few things. Below are a few examples of configurations we will set as we get started with Git:

- our name and email address,
- to colorize our output,
- what our preferred text editor is,
- and that we want to use these settings globally (i.e. for every project)

On a command line, Git commands are written as `git verb`, where `verb` is what we actually want to do. So here is how sets up his new laptop:

```
$ git config --global user.name   "Jonathan the Husky"
$ git config --global user.email  "jonathan.husky@uconn.edu"
$ git config --global color.ui    "auto"
$ git config --global core.editor "gedit -s -w"
```

(Please use your own name and email address instead of J's.)

In the last command he has set his favorite text editor, following this table:

Editor	Configuration command
nano	`git config --global core.editor "nano -w"`
gedit	`git config --global core.editor "gedit -s -w"`
emacs	`git config --global core.editor "emacs"`
vim	`git config --global core.editor "vim"`

The four commands we just ran above only need to be run once: the flag `--global` tells Git to use the settings for every project, in your user account, on this computer. If the optional flag `--global` is omitted, the settings are applied the the current project only.

You can check your settings at any time:

```
$ git config --list
```

You can change your configuration as many times as you want: just use the same commands to choose another editor or update your email address.

1.3 Creating a Repository

Once Git is configured, we can start using it. Let's create a directory for our work and then move into that directory:

```
$ mkdir planets
$ cd planets
```

Then we tell Git to make `planets` a repository — a place where Git can store versions of our files:

```
$ git init
```

If we use `ls` to show the directory's contents, it appears that nothing has changed:

```
$ ls
```

But if we add the `-a` flag to show everything, we can see that Git has created a hidden directory within `planets` called `.git`:

```
$ ls -a
```

```
.    ..   .git
```

Git stores information about the project in this special sub-directory. If we ever delete it, we will lose the project's history.

We can check that everything is set up correctly by asking Git to tell us the status of our project:

```
$ git status
```

```
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

1.4 Tracking Changes

In this section we (a) go through the modify-add-commit cycle for a single file; (b) explain where information is stored at each stage of Git commit workflow.

Let's create a file called `mars.txt` that contains some notes about the Red Planet's suitability as a base. (We'll use nano to edit the file; you can use whatever editor you like. In particular, this does not have to be the `core.editor` you set globally earlier.)

```
$ nano mars.txt
```

Type the text below into the `mars.txt` file:

```
Cold and dry, but everything is my favorite color
```

`mars.txt` now contains a single line, which we can see by running:

```
$ ls
mars.txt
```

```
$ cat mars.txt
Cold and dry, but everything is my favorite color
```

If we check the status of our project again, Git tells us that it's noticed the new file:

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        mars.txt

nothing added to commit but untracked files present
```

The "untracked files" message means that there's a file in the directory that Git isn't keeping track of. We can tell Git to track a file using `git add`:

```
$ git add mars.txt
```

and then check that the right thing happened:

```
$ git status
```

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   mars.txt
```

Git now knows that it's supposed to keep track of `mars.txt`, but it hasn't recorded these changes as a commit yet. To get it to do that, we need to run one more command:

```
$ git commit -m "Start notes on Mars as a base"
```

```
[master (root-commit) 61f01fe] Start notes on Mars as a base
 1 file changed, 1 insertion(+)
 create mode 100644 mars.txt
```

When we run `git commit`, Git takes everything we have told it to save by using `git add` and stores a copy permanently inside the special `.git` directory. This permanent copy is called a *commit* (or *revision*) and its short identifier is `61f01fe` (Your commit will have another identifier.)

We use the `-m` flag (for "message") to record a short, descriptive, and specific comment that will help us remember later on what we did and why. If we just run `git commit` without the `-m` option, Git will launch the editor we configured as `core.editor` so that we can write a longer message.

Good commit messages start with a brief (<50 characters) summary of changes made in the commit. If you want to go into more detail, add a blank line between the summary line and your additional notes.

If we run `git status` now:

```
$ git status
```

```
On branch master
nothing to commit, working directory clean
```

it tells us everything is up to date. If we want to know what we've done recently, we can ask Git to show us the project's history using `git log`:

```
$ git log
```

```
commit f22b25e3233b4645dabd0d81e651fe074bd8e73b
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 09:51:46 2013 -0400

    Start notes on Mars as a base
```

`git log` lists all commits made to a repository in reverse chronological order. The listing for each commit includes the commit's full identifier (which starts with the same characters as the short identifier printed by the `git commit` command earlier), the commit's author, when it was created, and the log message Git was given when the commit was created.

1.4.1 Where Are My Changes?

If we run `ls` at this point, we will still see just one file called `mars.txt`. That's because Git saves information about files' history in the special `.git` directory mentioned earlier so that our filesystem doesn't become cluttered (and so that we can't accidentally edit or delete an old version).

Now suppose Dracula adds more information to the file. (Again, we'll edit with nano and then `cat` the file to show its contents; you may use a different editor, and don't need to `cat`.)

```
$ nano mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
```

When we run `git status` now, it tells us that a file it already knows about has been modified:

```
$ git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
     directory)

  modified:   mars.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

The last line is the key phrase: "no changes added to commit". We have changed this file, but we haven't told Git we will want to save those changes (which we do with `git add`) nor have we saved them (which we do with `git commit`). So let's do that now. It is good practice to always review our changes before saving them. We do this using `git diff`. This shows us the differences between the current state of the file and the most recently saved version:

```
$ git diff
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..315bf3a 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,2 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
```

The output is cryptic because it is actually a series of commands for tools like editors and `patch` telling them how to reconstruct one file given the other. If we break it down into pieces:

1. The first line tells us that Git is producing output similar to the Unix `diff` command comparing the old and new versions of the file.
2. The second line tells exactly which versions of the file Git is comparing; `df0654a` and `315bf3a` are unique computer-generated labels for those versions.
3. The third and fourth lines once again show the name of the file being changed.
4. The remaining lines are the most interesting, they show us the actual differences and the lines on which they occur. In particular, the + markers in the first column show

where we have added lines.

After reviewing our change, it's time to commit it:

```
$ git commit -m "Add concerns about effects of Mars on Wolfman"
$ git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
     directory)

  modified:   mars.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Whoops: Git won't commit because we didn't use `git add` first. Let's fix that:

```
$ git add mars.txt
$ git commit -m "Add concerns about effects of Mars on Wolfman"
```

```
[master 34961b1] Add concerns about effects of Mars on Wolfman
 1 file changed, 1 insertion(+)
```

Git insists that we add files to the set we want to commit before actually committing anything because we may not want to commit everything at once. For example, suppose we're adding a few citations to our supervisor's work to our thesis. We might want to commit those additions, and the corresponding addition to the bibliography, but *not* commit the work we're doing on the conclusion (which we haven't finished yet).

To allow for this, Git has a special *staging area* where it keeps track of things that have been added to the current *change set* but not yet committed.

1.4.2 Staging area

If you think of Git as taking snapshots of changes over the life of a project, `git add` specifies *what* will go in a snapshot (putting things in the staging area), and `git commit` then *actually takes* the snapshot, and makes a permanent record of it (as a commit). If you don't have anything staged when you type `git commit`, Git will prompt you to use `git commit -a` or `git commit --all`, which is kind of like gathering *everyone* for the picture! However, it's almost always better to explicitly add things to the staging area, because you might commit changes you forgot you made. (Going back to snapshots, you might get the extra with incomplete makeup walking on the stage for the snapshot because you used -a!) Try to stage things manually, or you might find yourself searching for “git undo commit” more than you would like!

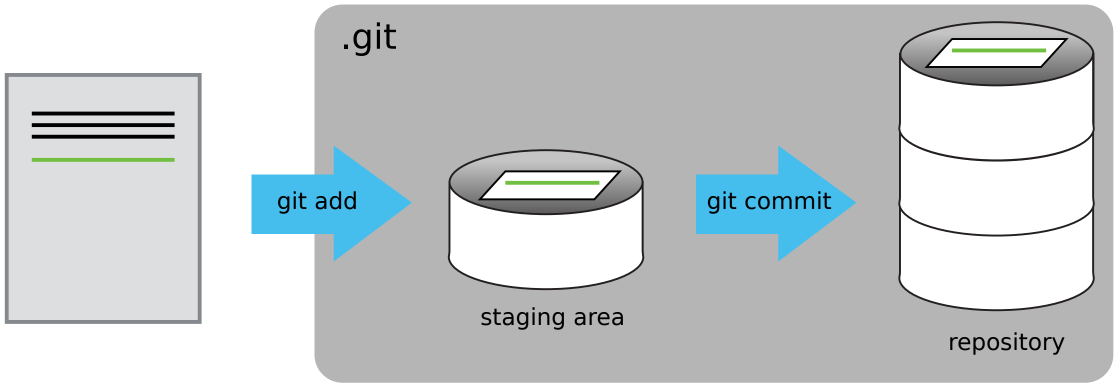


Figure 1.4: The Git Staging Area

Let’s watch as our changes to a file move from our editor to the staging area and into long-term storage. First, we’ll add another line to the file:

```
$ nano mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

```
$ git diff
```

```
diff --git a/mars.txt b/mars.txt
index 315bf3a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 +1,3 @@
 Cold and dry, but everything is my favorite color
 The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

So far, so good: we’ve added one line to the end of the file (shown with a + in the first column). Now let’s put that change in the staging area and see what `git diff` reports:

```
$ git add mars.txt
$ git diff
```

There is no output: as far as Git can tell, there's no difference between what it's been asked to save permanently and what's currently in the directory. However, if we do this:

```
$ git diff --staged
```

```
diff --git a/mars.txt b/mars.txt
index 315bf3a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 +1,3 @@
 Cold and dry, but everything is my favorite color
 The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

it shows us the difference between the last committed change and what's in the staging area. Let's save our changes:

```
$ git commit -m "Discuss concerns about Mars' climate for Mummy"
```

```
[master 005937f] Discuss concerns about Mars' climate for Mummy
 1 file changed, 1 insertion(+)
```

check our status:

```
$ git status
```

```
On branch master
nothing to commit, working directory clean
```

and look at the history of what we've done so far:

```
$ git log
```

```
commit 005937fbe2a98fb83f0ade869025dc2636b4dad5
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:14:07 2013 -0400

    Discuss concerns about Mars' climate for Mummy

commit 34961b159c27df3b475cfe4415d94a6d1fcd064d
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 10:07:21 2013 -0400

    Add concerns about effects of Mars' moons on Wolfman

commit f22b25e3233b4645dabd0d81e651fe074bd8e73b
Author: Vlad Dracula <vlad@tran.sylvan.ia>
Date:   Thu Aug 22 09:51:46 2013 -0400

    Start notes on Mars as a base
```

To recap, when we want to add changes to our repository, we first need to add the changed files to the staging area (`git add`) and then commit the staged changes to the repository (`git commit`):

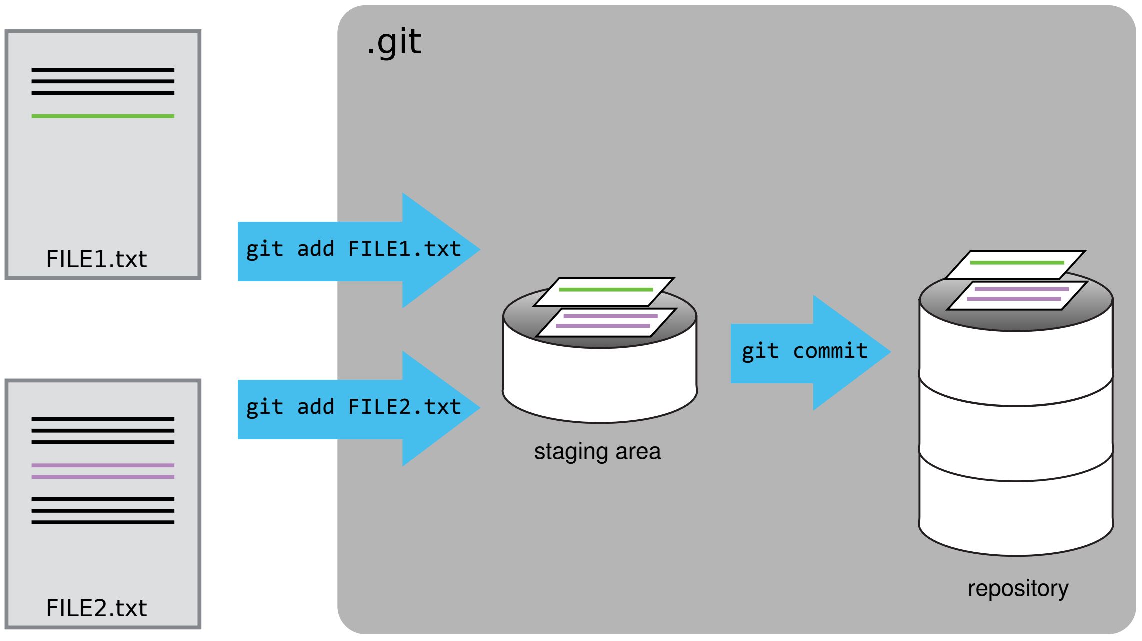


Figure 1.5: The Git Commit Workflow

1.4.3 Questions

Choosing a commit message: Which of the following commit messages would be most appropriate for the last commit made to `mars.txt`?

1. "Changes"
2. "Added line 'But the Mummy will appreciate the lack of humidity' to mars.txt"
3. "Discuss effects of Mars' climate on the Mummy"

Committing Changes to Git: Which command(s) below would save the changes of `myfile.txt` to my local Git repository?

1.
```
$ git commit -m "my recent changes"
```
2.
```
$ git init myfile.txt
$ git commit -m "my recent changes"
```
3.
```
$ git add myfile.txt
$ git commit -m "my recent changes"
```
4.
```
$ git commit -m myfile.txt "my recent changes"
```

1.5 Exploring History

In this section we learn how to (a) identify and use Git commit numbers; (b) compare various versions of tracked files; (c) restore old versions of files.

If we want to see what we changed at different steps, we can use `git diff` again, but with the notation HEAD~1, HEAD~2, and so on, to refer to old commits:

```
$ git diff HEAD~1 mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index 315bf3a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1,2 +1,3 @@
 Cold and dry, but everything is my favorite color
 The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

```
$ git diff HEAD~2 mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,3 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

In this way, we can build up a chain of commits. The most recent end of the chain is referred to as HEAD; we can refer to previous commits using the ~ notation, so HEAD~1 (pronounced "head minus one") means "the previous commit", while HEAD~123 goes back 123 commits from where we are now.

We can also refer to commits using those long strings of digits and letters that `git log` displays. These are unique IDs for the changes, and "unique" really does mean unique: every change to any set of files on any computer has a unique 40-character identifier. Our first commit was given the ID f22b25e3233b4645dabd0d81e651fe074bd8e73b, so let's try this:

```
$ git diff f22b25e3233b4645dabd0d81e651fe074bd8e73b mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,3 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

That's the right answer, but typing out random 40-character strings is annoying, so Git lets us use just the first few characters:

```
$ git diff f22b25e mars.txt
```

```
diff --git a/mars.txt b/mars.txt
index df0654a..b36abfd 100644
--- a/mars.txt
+++ b/mars.txt
@@ -1 +1,3 @@
 Cold and dry, but everything is my favorite color
+The two moons may be a problem for Wolfman
+But the Mummy will appreciate the lack of humidity
```

All right! So we can save changes to files and see what we've changed—now how can we restore older versions of things? Let's suppose we accidentally overwrite our file:

```
$ nano mars.txt
$ cat mars.txt
```

```
We will need to manufacture our own oxygen
```

`git status` now tells us that the file has been changed, but those changes haven't been staged:

```
$ git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
     directory)

  modified:   mars.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

We can put things back the way they were by using `git checkout`:

```
$ git checkout HEAD mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

As you might guess from its name, `git checkout` checks out (i.e., restores) an old version of a file. In this case, we're telling Git that we want to recover the version of the file recorded in `HEAD`, which is the last saved commit. If we want to go back even further, we can use a commit identifier instead:

```
$ git checkout f22b25e mars.txt
```

1.5.1 Don't lose your HEAD

Above we used

```
$ git checkout f22b25e mars.txt
```

to revert mars.txt to its state after the commit f22b25e. If you forget `mars.txt` in that command, git will tell you that "You are in 'detached HEAD' state." In this state, you shouldn't make any changes. You can fix this by reattaching your head using `git checkout master`

It's important to remember that we must use the commit number that identifies the state of the repository *before* the change we're trying to undo. A common mistake is to use the number of the commit in which we made the change we're trying to get rid of. In the example below, we want to retrieve the state from before the most recent commit (`HEAD~1`), which is commit `f22b25e`:

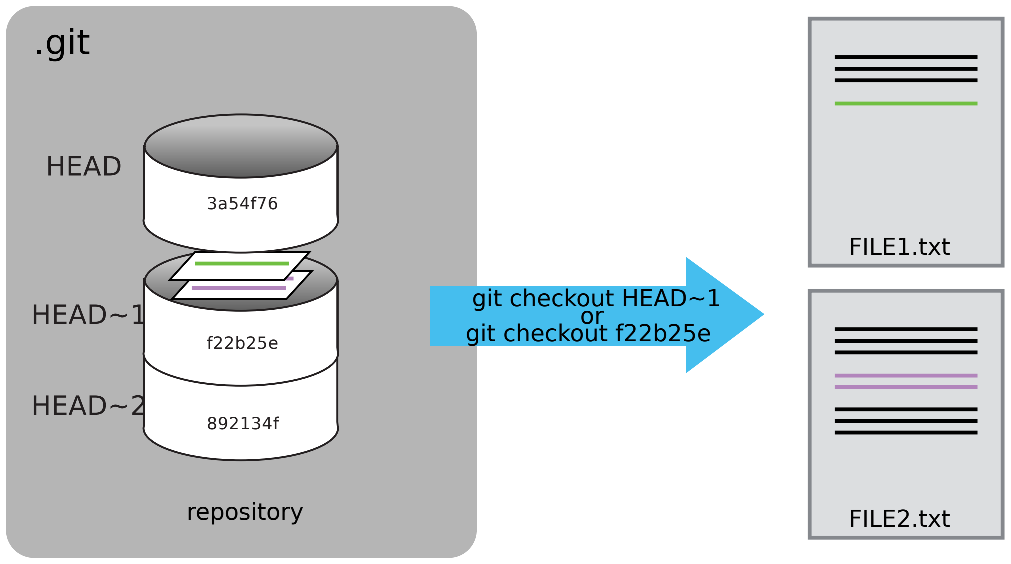


Figure 1.6: Git Checkout

1.5.2 Simplifying the Common Case

If you read the output of `git status` carefully, you'll see that it includes this hint:

(use "git checkout – <file>..." to discard changes in working directory)

As it says, `git checkout` without a version identifier restores files to the state saved in `HEAD`. The double dash `--` is needed to separate the names of the files being recovered from the command itself: without it, Git would try to use the name of the file as the commit identifier.

The fact that files can be reverted one by one tends to change the way people organize their work. If everything is in one large document, it's hard (but not impossible) to undo changes to the introduction without also undoing changes made later to the conclusion. If the introduction and conclusion are stored in separate files, on the other hand, moving backward and forward in time becomes much easier.

1.5.3 Questions

Recovering Older Versions of a File: Jennifer has made changes to the Python script that she has been working on for weeks, and the modifications she made this morning "broke" the script and it no longer runs. She has spent ~ 1hr trying to fix it, with no luck...

Luckily, she has been keeping track of her project's versions using Git! Which commands below will let her recover the last committed version of her Python script called `data_cruncher.py`?

1. `$ git checkout HEAD`
2. `$ git checkout HEAD data_cruncher.py`
3. `$ git checkout HEAD~1 data_cruncher.py`
4. `$ git checkout <unique ID of last commit> data_cruncher.py`
5. Both 2 & 4

Understanding Workflow and History: What is the output of cat venus.txt at the end of this set of commands?

```
$ cd planets
$ nano venus.txt #input the following text:
                 Venus is beautiful and full of love
$ git add venus.txt
$ nano venus.txt #add the following text:
                 Venus is too hot to be suitable as a base
$ git commit -m "comments on Venus as an unsuitable base"
$ git checkout HEAD venus.txt
$ cat venus.txt  #this will print the contents of venus.txt
                 #to the screen
```

1. `Venus is too hot to be suitable as a base`
2. `Venus is beautiful and full of love`
3. `Venus is beautiful and full of love`
   `Venus is too hot to be suitable as a base`
4. Error because you have changed venus.txt without committing the changes.

1.6 Ignoring Things

In this section we learn how to (a) configure Git to ignore specific files; (b) explain why ignoring files can be useful.

What if we have files that we do not want Git to track for us, like backup files created by our editor or intermediate files created during data analysis. Let's create a few dummy files:

```
$ mkdir results
$ touch a.dat b.dat c.dat results/a.out results/b.out
```

and see what Git says:

```
$ git status
```

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

  a.dat
  b.dat
  c.dat
  results/
nothing added to commit but untracked files present (use "git
   add" to track)
```

Putting these files under version control would be a waste of disk space. What's worse, having them all listed could distract us from changes that actually matter, so let's tell Git to ignore them.

We do this by creating a file in the root directory of our project called `.gitignore`:

```
$ nano .gitignore
$ cat .gitignore
```

```
*.dat
results/
```

These patterns tell Git to ignore any file whose name ends in `.dat` and everything in the `results` directory. (If any of these files were already being tracked, Git would continue to track them.)

Once we have created this file, the output of `git status` is much cleaner:

```
$ git status
```

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

  .gitignore
nothing added to commit but untracked files present (use "git
   add" to track)
```

The only thing Git notices now is the newly-created `.gitignore` file. You might think we wouldn't want to track it, but everyone we're sharing our repository with will probably want to ignore the same things that we're ignoring. Let's add and commit `.gitignore`:

```
$ git add .gitignore
$ git commit -m "Add the ignore file"
$ git status
```

```
On branch master
nothing to commit, working directory clean
```

As a bonus, using `.gitignore` helps us avoid accidentally adding to the repository files that we don't want to track:

```
$ git add a.dat
```

```
The following paths are ignored by one of your .gitignore files:
a.dat
Use -f if you really want to add them.
fatal: no files added
```

If we really want to override our ignore settings, we can use `git add -f` to force Git to add something. We can also always see the status of ignored files if we want:

```
$ git status --ignored
On branch master
Ignored files:
 (use "git add -f <file>..." to include in what will be committed)

       a.dat
       b.dat
       c.dat
       results/

nothing to commit, working directory clean
```

1.6.1 Questions

Ignoring nested files: Given a directory structure that looks like:

```
results/data
results/plots
```

How would you ignore only `results/plots` and not `results/data`?

Including specific files: How would you ignore all `.data` files in your root directory except for `final.data`? Hint: Find out what `!` (the exclamation point operator) does.

Ignoring files deep in a directory: Given a directory structure that looks like:

```
results/data/position/gps/useless.data
results/plots
```

What's the shortest `.gitignore` rule you could write to ignore all `.data` files in `result/data/position/gps` Hint: What does appending `**` to a rule accomplish?

The order of rules: Given a `.gitignore` file with the following contents:

```
*.data
!*.data
```

What will be the result?

Log-files: You wrote a script that creates many intermediate log-files of the form log_01, log_02, log_03, etc. You want to keep them but you do not want to track them through `git`.

1. Write **one** `.gitignore` entry that excludes files of the form `log_01`, `log_02`, etc.
2. Test your "ignore pattern" by creating some dummy files of the form `log_01`, etc.
3. You find that the file `log_01` is very important after all, add it to the tracked files without changing the `.gitignore` again.
4. Discuss with your neighbor what other types of files could reside in your directory that you do not want to track and thus would exclude via `.gitignore`.

1.7 Remote repositories

In this section we (a) explain what remote repositories are and why they are useful, and (b) learn how to push to or pull from a remote repository.

Version control really comes into its own when we begin to collaborate with other people. We already have most of the machinery we need to do this; the only thing missing is to copy changes from one repository to another.

Systems like Git allow us to move work between any two repositories. In practice, though, it's easiest to use one copy as a central hub, and to keep it online rather than on someone's laptop. Hosting services like GitHub, GitLab or BitBucket are available to hold those master copies; we'll explore the pros and cons of this in this lesson.

1.7.1 GitHub

Let's start by sharing the changes we've made to our current project with the world. Log in to GitHub, then click on the icon in the top right corner to create a new repository (see Fig. **??**).

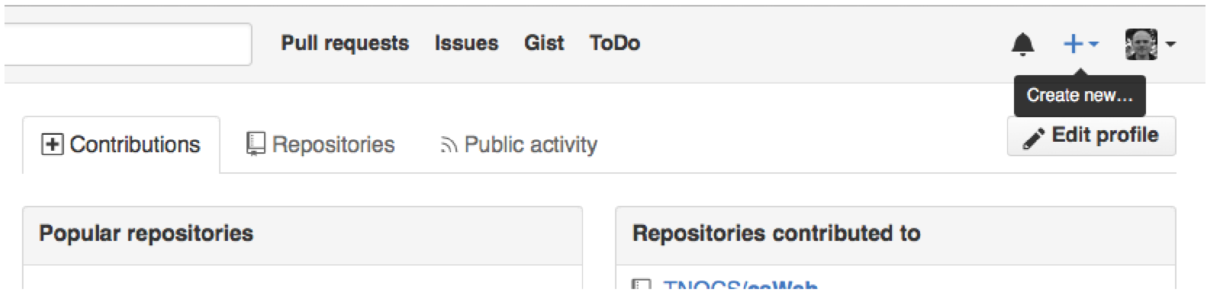


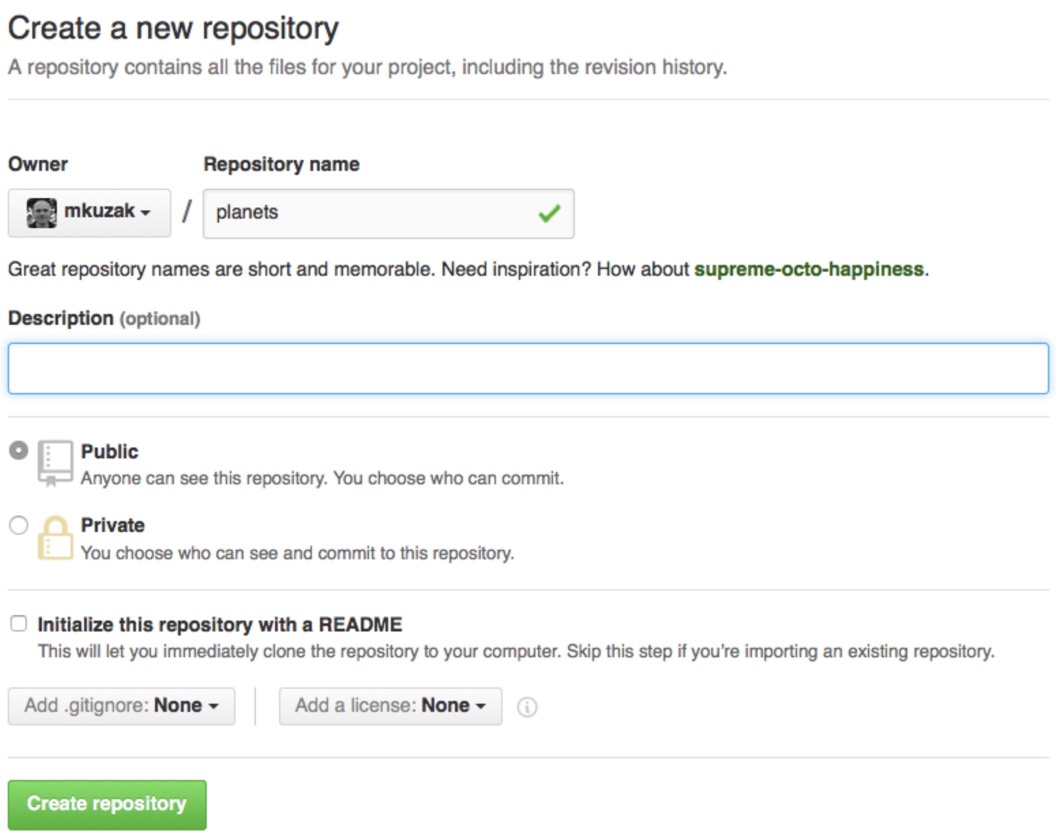
Figure 1.7: Creating a Repository on GitHub (Step 1)

Name your repository "planets" and then click "Create Repository" (see Fig. **??**).

As soon as the repository is created, GitHub displays a page with a URL and some information on how to configure your local repository (Fig. **??**).

Our local repository still contains our earlier work on `mars.txt`, but the remote repository on GitHub doesn't contain any files yet.

The next step is to connect the two repositories. We do this by making the GitHub repository a *remote* for the local repository. The home page of the repository on GitHub includes the string we need to identify it (Fig. **??**).


Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: mkuzak / Repository name: planets

Great repository names are short and memorable. Need inspiration? How about **supreme-octo-happiness**.

Description (optional)

[✓ Public] Anyone can see this repository. You choose who can commit.

Private — You choose who can see and commit to this repository.

Initialize this repository with a README

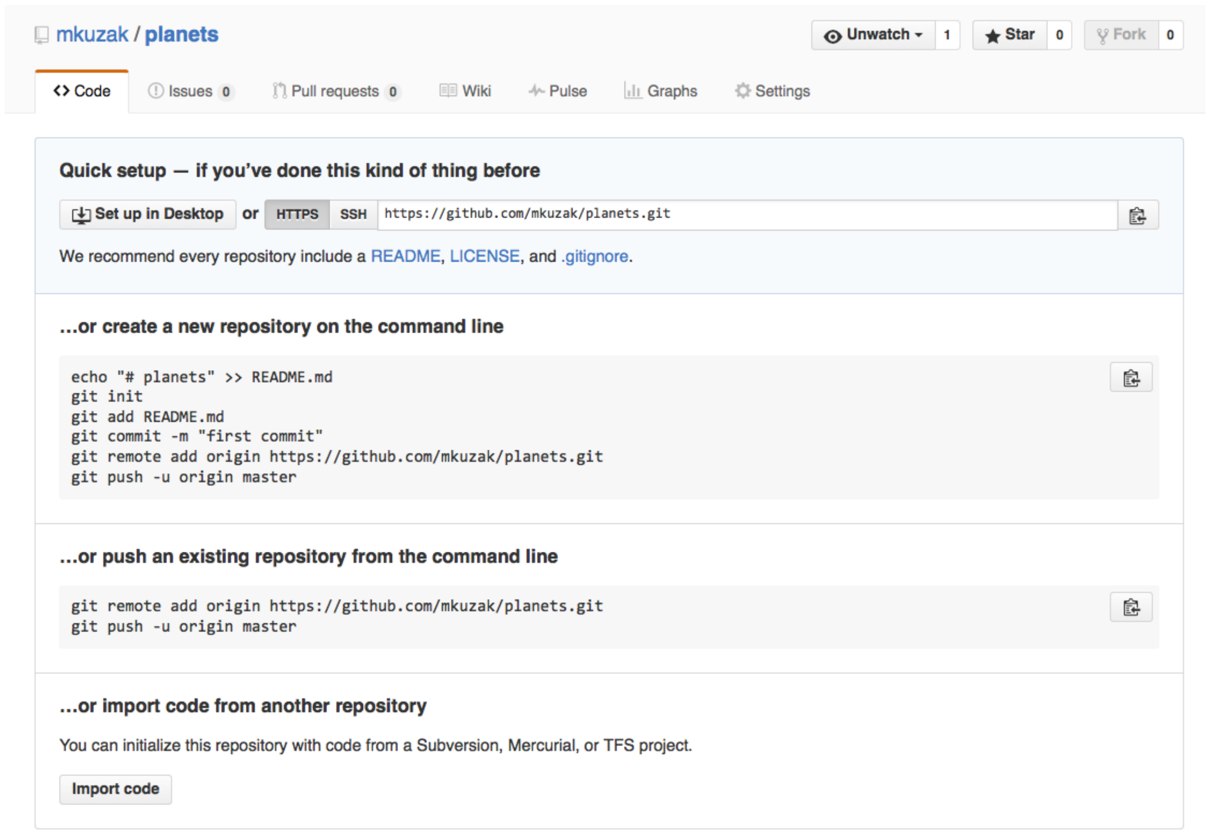
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None

Add a license: None

Create repository

Figure 1.8: Creating a Repository on GitHub (Step 2)



Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH https://github.com/mkuzak/planets.git

We recommend every repository include a README, LICENSE, and .gitignore.

…or create a new repository on the command line

```
echo "# planets" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/mkuzak/planets.git
git push -u origin master
```

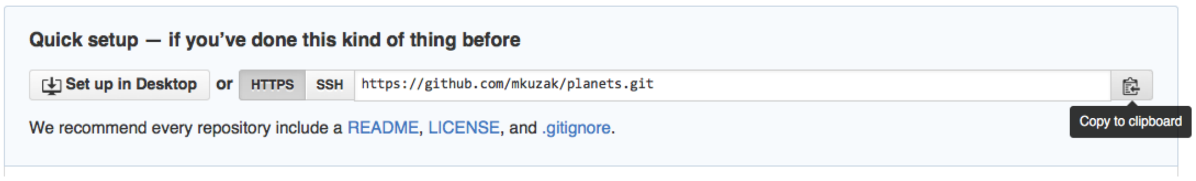
…or push an existing repository from the command line

```
git remote add origin https://github.com/mkuzak/planets.git
git push -u origin master
```

…or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Figure 1.9: Creating a Repository on GitHub (Step 3)



Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH https://github.com/mkuzak/planets.git

We recommend every repository include a README, LICENSE, and .gitignore.

Figure 1.10: Where to Find Repository URL on GitHub

Copy that URL from the browser, go into the local `planets` repository, and run this command:

```
$ git remote add origin https://github.com/vlad/planets.git
```

Make sure to use the URL for **your** repository; the only difference should be your username instead of `vlad`.

We can check that the command has worked by running `git remote -v`:

```
$ git remote -v
origin   https://github.com/vlad/planets.git (push)
origin   https://github.com/vlad/planets.git (fetch)
```

The name `origin` is a local nickname for your remote repository; we could use something else if we wanted to, but `origin` is by far the most common choice.

Once the nickname `origin` is set up, this command will push the changes from our local repository to the repository on GitHub:

```
$ git push origin master
Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (9/9), 821 bytes, done.
Total 9 (delta 2), reused 0 (delta 0)
To https://github.com/vlad/planets
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

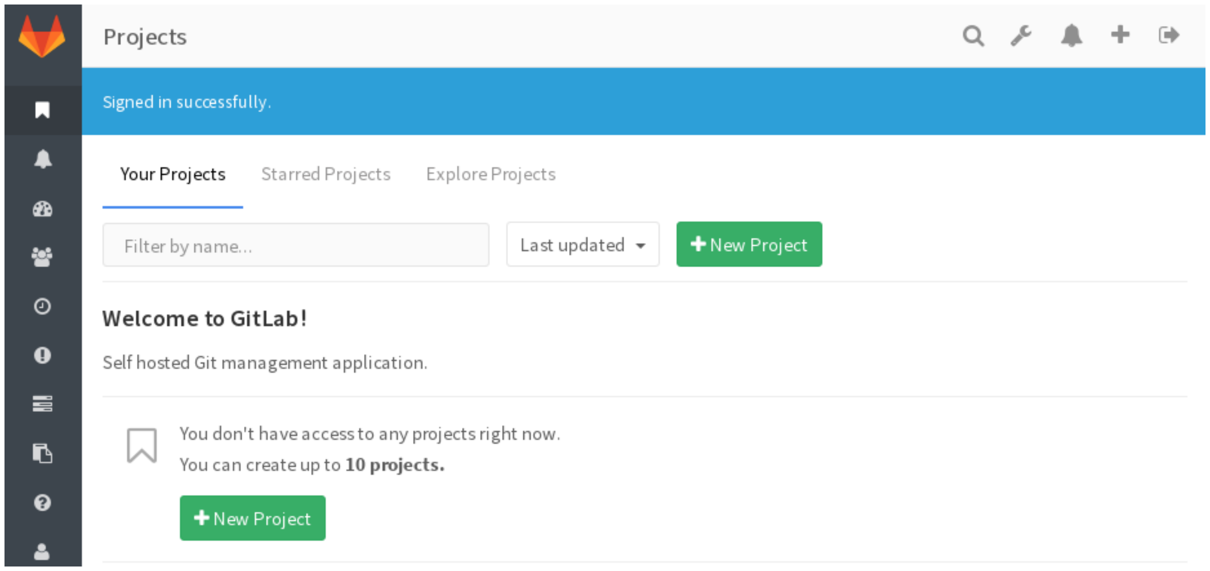
We can pull changes from the remote repository to the local one as well:

```
$ git pull origin master
From https://github.com/vlad/planets
 * branch            master     -> FETCH_HEAD
Already up-to-date.
```

Pulling has no effect in this case because the two repositories are already synchronized. If someone else had pushed some changes to the repository on GitHub, though, this command would download them to our local repository.

1.7.2 GitLab

To create a new project, sign in to GitLab. Go to your Dashboard and click on "new project" on the right side of your screen (see Fig. **??**).



Signed in successfully.

Welcome to GitLab!

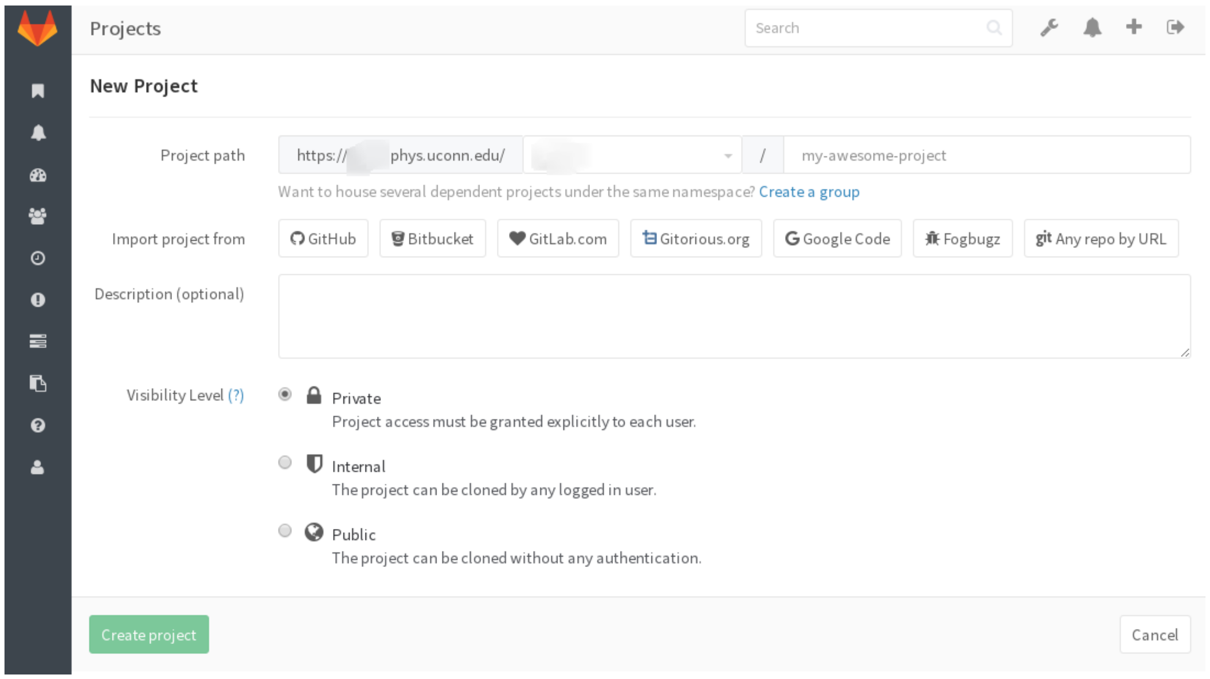
Self hosted Git management application.

You don't have access to any projects right now.
You can create up to **10 projects.**

Figure 1.11: GitLab dashboard

On the next screen (see Fig. **??**) fill out the required information: the name of your project (you can't add spaces but you can use hyphens or underscores), your project's description, a visibility level. Click on "create project"

Follow the command line instructions on the next screen (see Fig. **??**) to create a new repository from scratch or import an existing local git repository.


Projects

New Project

Project path: https:// phys.uconn.edu/ / my-awesome-project

Want to house several dependent projects under the same namespace? Create a group

Import project from: GitHub | Bitbucket | GitLab.com | Gitorious.org | Google Code | Fogbugz | git Any repo by URL

Description (optional)

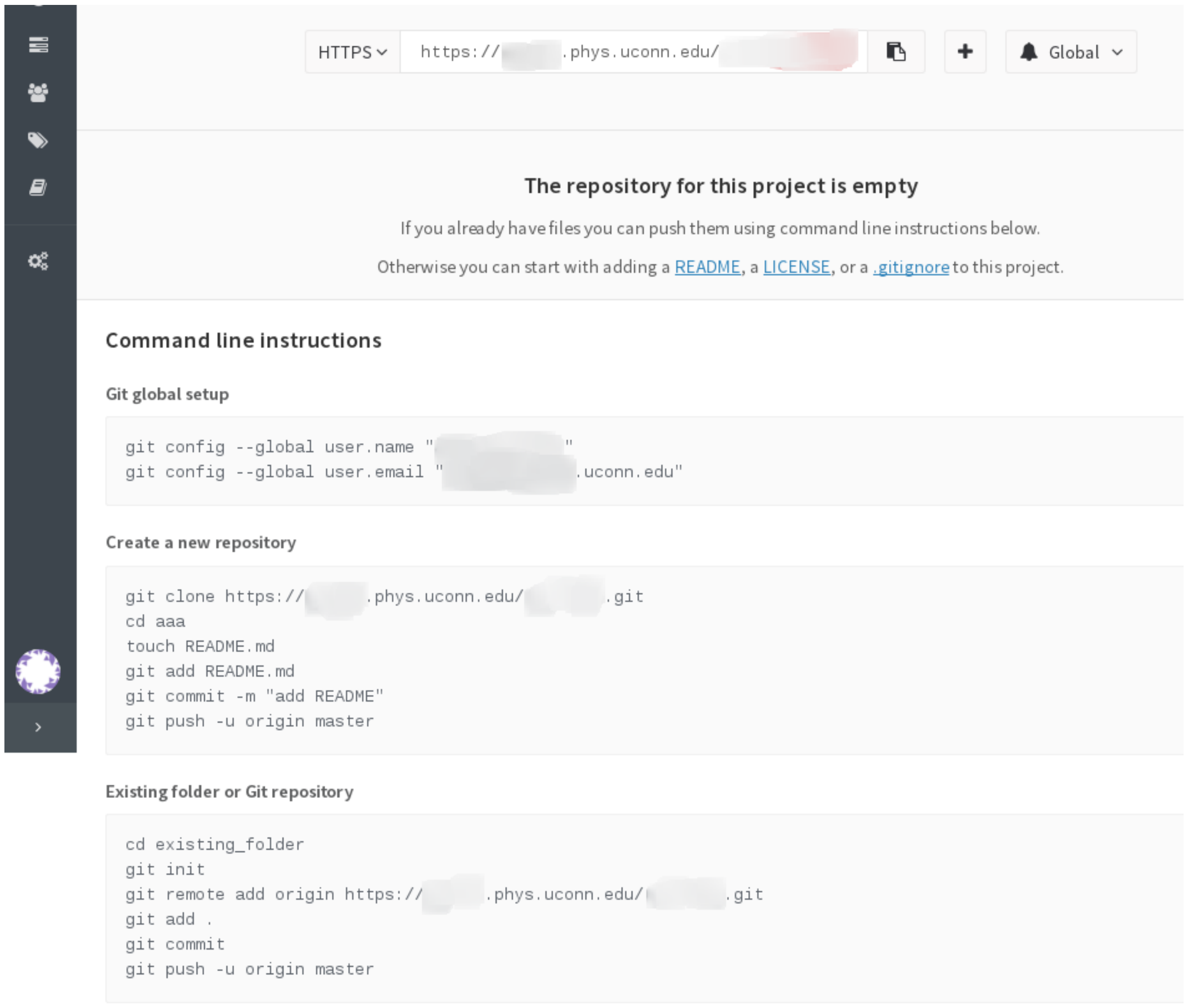
Visibility Level (?)

[✓ Private]
Project access must be granted explicitly to each user.

Internal
The project can be cloned by any logged in user.

Public
The project can be cloned without any authentication.

Figure 1.12: GitLab create new project, Step 2.



The repository for this project is empty

If you already have files you can push them using command line instructions below.

Otherwise you can start with adding a README, a LICENSE, or a .gitignore to this project.

Command line instructions

Git global setup

```
git config --global user.name "[redacted]"
git config --global user.email "[redacted].uconn.edu"
```

Create a new repository

```
git clone https://[redacted].phys.uconn.edu/[redacted].git
cd aaa
touch README.md
git add README.md
git commit -m "add README"
git push -u origin master
```

Existing folder or Git repository

```
cd existing_folder
git init
git remote add origin https://[redacted].phys.uconn.edu/[redacted].git
git add .
git commit
git push -u origin master
```

Figure 1.13: GitLab create new project, Step 3.

1.7.3 Storing credentials for remote repositories

If you're cloning Git repositories using HTTPS, you can use a *credential helper* to tell Git to remember your username and password.

The point of this helper is to reduce the number of times you must type your username or password. For example:

```
$ git config credential.helper store
$ git push http://example.com/repo.git
```

Username: <type your username> Password: <type your password>

[several days later]

```
$ git push http://example.com/repo.git
```

[your credentials are used automatically]

1.7.4 Questions

GitHub GUI: Browse to your `planets` repository on GitHub. Under the Code tab, find and click on the text that says "XX commits" (where "XX" is some number). Hover over, and click on, the three buttons to the right of each commit. What information can you gather/explore from these buttons? How would you get that same information in the shell?

GitHub Timestamp: Create a remote repository on GitHub. Push the contents of your local repository to the remote. Make changes to your local repository and push these changes. Go to the repo you just created on Github and check the timestamps of the files. How does GitHub record times, and why?

Push vs. commit: In this lesson, we introduced the "git push" command. How is "git push" different from "git commit"?

Fixing up remote settings: It happens quite often in practice that you made a typo in the remote URL. This exercice is about how to fix this kind of issues. First start by adding a remote with an invalid URL:

```
$ git remote add broken https://github.com/this/url/isinvalid
```

Do you get an error when adding the remote? Can you think of a command that would make it obvious that your remote URL was not valid? Can you figure out how to fix the URL (tip: use `git remote -h`)? Don't forget to clean up and remove this remote once you are done with this exercise.

1.8 Collaborating

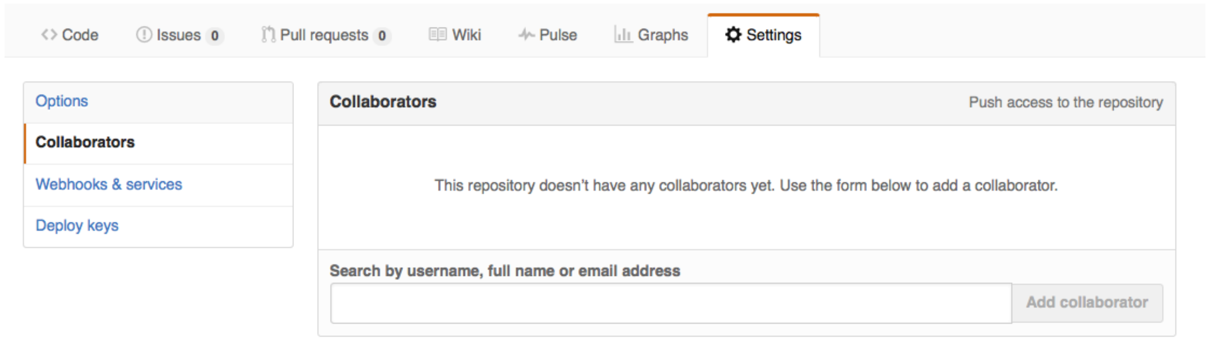
In this section we learn how to (a) clone a remote repository, and (b) collaborate pushing to a common repository.

For the next step, get into pairs. One person will be the "Owner" (this is the person whose Github repository will be used to start the exercise) and the other person will be the "Collaborator" (this is the person who will be cloning the Owner's repository and making changes to it).

1.8.1 Practicing by yourself

If you're working through this lesson on your own, you can carry on by opening a second terminal window, and switching to another directory (e.g. `/tmp`). This window will represent your partner, working on another computer. You won't need to give anyone access on GitHub, because both 'partners' are you.

The Owner needs to give the Collaborator access. On GitHub, click the settings button on the right, then select Collaborators, and enter your partner's username.



This repository doesn't have any collaborators yet. Use the form below to add a collaborator.

Figure 1.14: Adding collaborators on GitHub

The Collaborator needs to work on this project locally. He or she should `cd` to another directory (so `ls` doesn't show a `planets` folder), and then make a copy of the Owner's repository:

```
$ git clone https://github.com/vlad/planets.git
```

Replace 'vlad' with the Owner's username.

`git clone` creates a fresh local copy of a remote repository.

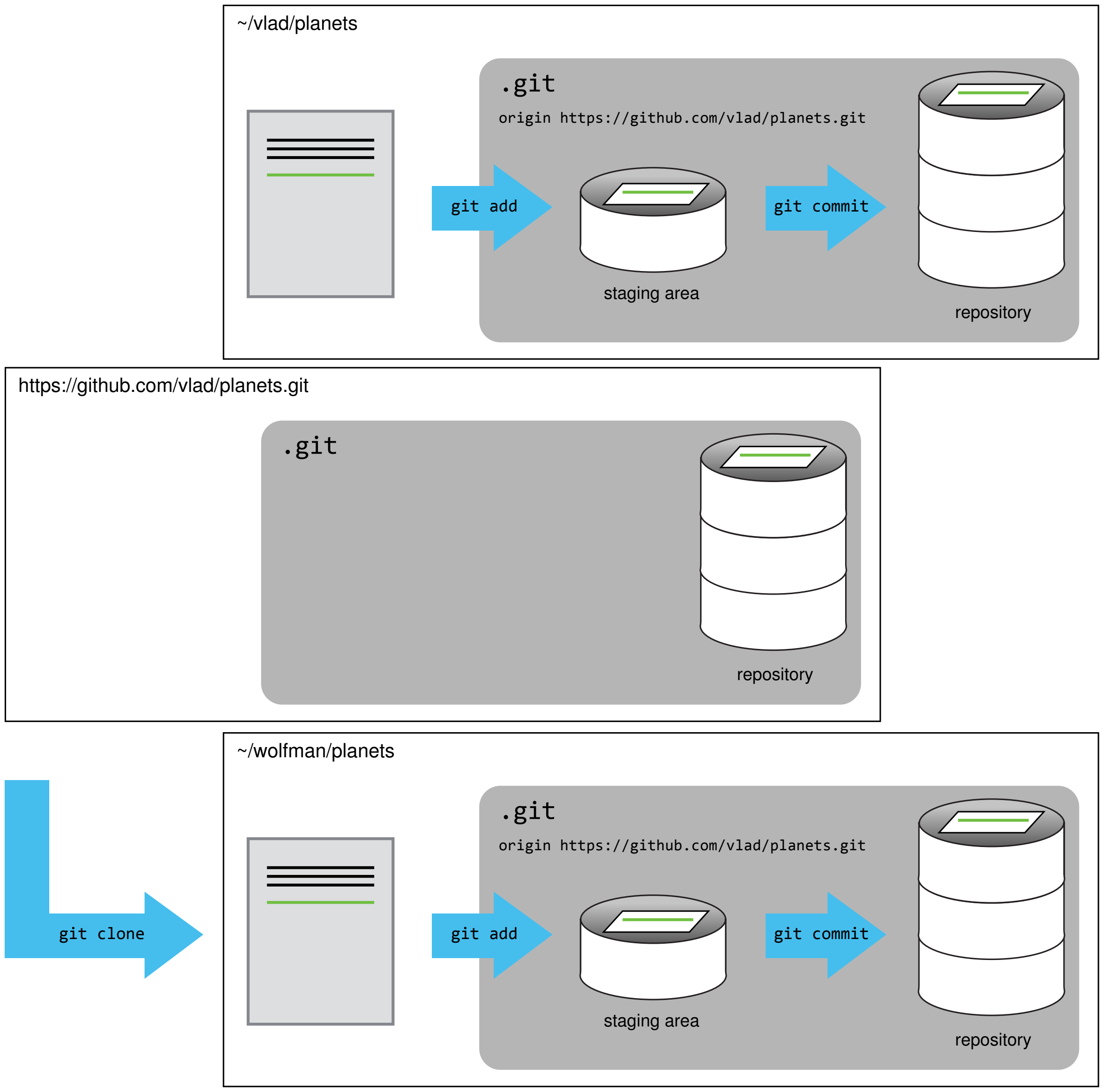


Figure 1.15: After Creating Clone of Repository

The Collaborator can now make a change in his or her copy of the repository:

```
$ cd planets
$ nano pluto.txt
$ cat pluto.txt
```

```
It is so a planet!
```

```
$ git add pluto.txt
$ git commit -m "Some notes about Pluto"
```

```
 1 file changed, 1 insertion(+)
 create mode 100644 pluto.txt
```

Then push the change to GitHub:

```
$ git push origin master
```

```
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 306 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/vlad/planets.git
   9272da5..29aba7c  master -> master
```

Note that we didn't have to create a remote called `origin`: Git does this automatically, using that name, when we clone a repository. (This is why `origin` was a sensible choice earlier when we were setting up remotes by hand.)

We can now download changes into the original repository on our machine:

```
$ git pull origin master
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/vlad/planets
 * branch            master     -> FETCH_HEAD
Updating 9272da5..29aba7c
Fast-forward
 pluto.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 pluto.txt
```

1.8.2 Questions

Review changes. The Owner push commits to the repository without giving any information to the Collaborator. How can the Collaborator find out what has changed with command line? And on GitHub?

1.9 Conflicts

This section (a) explains what conflicts are and when they can occur, and resolve conflicts resulting from a merge.

As soon as people can work in parallel, it's likely someone's going to step on someone else's toes. This will even happen with a single person: if we are working on a piece of software on both our laptop and a server in the lab, we could make different changes to each copy. Version control helps us manage these *conflicts* by giving us tools to *resolve* overlapping changes.

To see how we can resolve conflicts, we must first create one. The file `mars.txt` currently looks like this in both partners' copies of our `planets` repository:

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
```

Let's add a line to one partner's copy only:

```
$ nano mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
This line added to Wolfman's copy
```

and then push the change to GitHub:

```
$ git add mars.txt
$ git commit -m "Adding a line in our home copy"
```

```
[master 5ae9631] Adding a line in our home copy
 1 file changed, 1 insertion(+)
```

```
$ git push origin master
```

```
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 352 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/vlad/planets
   29aba7c..dabb4c8  master -> master
```

Now let's have the other partner make a different change to their copy *without* updating from GitHub:

```
$ nano mars.txt
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
We added a different line in the other copy
```

We can commit the change locally:

```
$ git add mars.txt
$ git commit -m "Adding a line in my copy"
```

```
[master 07ebc69] Adding a line in my copy
 1 file changed, 1 insertion(+)
```

but Git won't let us push it to GitHub:

```
$ git push origin master
To https://github.com/vlad/planets.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to
   'https://github.com/vlad/planets.git'
hint: Updates were rejected because the tip of your current
   branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git
   pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for
   details.
```

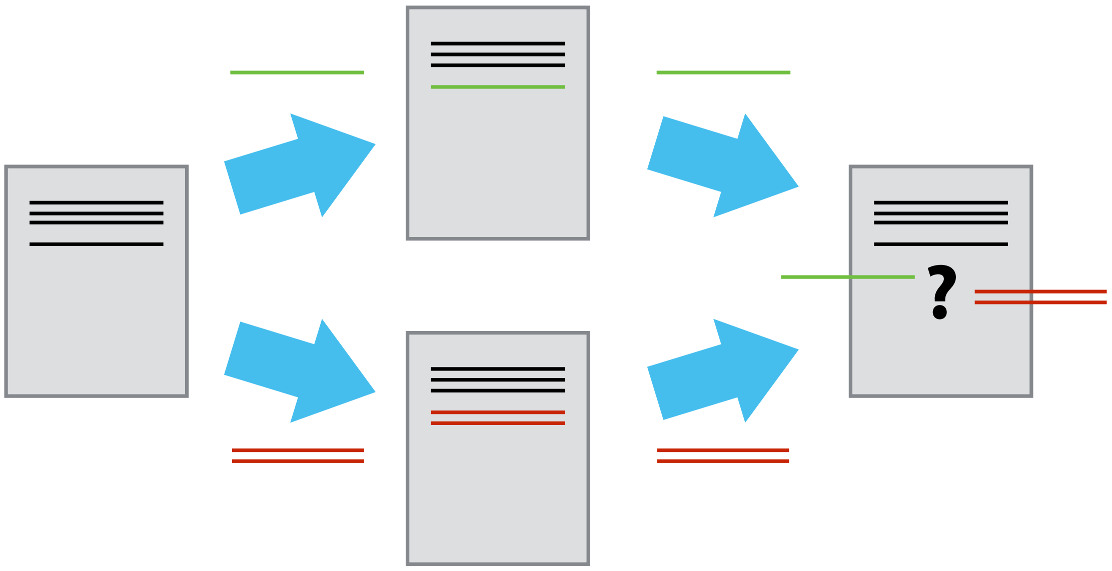


Figure 1.16: The conflicting changes

Git detects that the changes made in one copy overlap with those made in the other and stops us from trampling on our previous work. What we have to do is pull the changes from GitHub, merge them into the copy we're currently working in, and then push that. Let's start by pulling:

```
$ git pull origin master
```

```
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From https://github.com/vlad/planets
 * branch            master     -> FETCH_HEAD
Auto-merging mars.txt
CONFLICT (content): Merge conflict in mars.txt
Automatic merge failed; fix conflicts and then commit the result.
```

`git pull` tells us there's a conflict, and marks that conflict in the affected file:

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
<<<<<<< HEAD
We added a different line in the other copy
=======
This line added to Wolfman's copy
>>>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d
```

Our change—the one in HEAD—is preceded by <<<<<<<. Git has then inserted ======= as a separator between the conflicting changes and marked the end of the content downloaded from GitHub with >>>>>>>. (The string of letters and digits after that marker identifies the commit we've just downloaded.)

It is now up to us to edit this file to remove these markers and reconcile the changes. We can do anything we want: keep the change made in the local repository, keep the change made in the remote repository, write something new to replace both, or get rid of the change entirely. Let's replace both so that the file looks like this:

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
We removed the conflict on this line
```

To finish merging, we add `mars.txt` to the changes being made by the merge and then commit:

```
$ git add mars.txt
$ git status
```

```
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

  modified:   mars.txt
```

```
$ git commit -m "Merging changes from GitHub"
```

```
[master 2abf2b1] Merging changes from GitHub
```

Now we can push our changes to GitHub:

```
$ git push origin master
```

```
Counting objects: 10, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 697 bytes, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/vlad/planets.git
   dabb4c8..2abf2b1  master -> master
```

Git keeps track of what we've merged with what, so we don't have to fix things by hand again when the collaborator who made the first change pulls again:

```
$ git pull origin master
```

```
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 2), reused 6 (delta 2)
Unpacking objects: 100% (6/6), done.
From https://github.com/vlad/planets
 * branch            master     -> FETCH_HEAD
Updating dabb4c8..2abf2b1
Fast-forward
 mars.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

We get the merged file:

```
$ cat mars.txt
```

```
Cold and dry, but everything is my favorite color
The two moons may be a problem for Wolfman
But the Mummy will appreciate the lack of humidity
We removed the conflict on this line
```

We don't need to merge again because Git knows someone has already done that.

Version control's ability to merge conflicting changes is another reason users tend to divide their programs and papers into multiple files instead of storing everything in one large file. There's another benefit too: whenever there are repeated conflicts in a particular file, the version control system is essentially trying to tell its users that they ought to clarify who's responsible for what, or find a way to divide the work up differently.

1.9.1 Questions

Solving Conflicts that You Create: Clone the repository created by your instructor. Add a new file to it, and modify an existing file (your instructor will tell you which one). When asked by your instructor, pull her changes from the repository to create a conflict, then resolve it.

Conflicts on Non-textual files: What does Git do when there is a conflict in an image or some other non-textual file that is stored in version control?

A typical work session: You sit down at your computer to work on a shared project that is tracked in a remote Git repository. During your work session, you take the following actions, but not in this order:

- *Make changes* by appending the number `100` to a text file `numbers.txt`
- *Update remote* repository to match the local repository
- *Celebrate* your success
- *Update local* repository to match the remote repository
- *Stage changes* to be committed
- *Commit changes* to the local repository

In what order should you perform these actions to minimize the chances of conflicts?

Chapter 2

Automation with Make

2.1 Introduction. Zipf's Law

In this section we explain (a) what Make is for, and (b) how Make differs from shell scripts.

Let's imagine that we're interested in testing Zipf's Law. Zipf's law states that given a corpus of natural language texts, the frequency of any word is inversely proportional to its rank in the frequency table. Thus the most frequent word will occur approximately twice as often as the second most frequent word, three times as often as the third most frequent word, etc.

We've compiled our raw data that we want to analyze - the books in public domain under U.S. copyright law that are collected by Project Gutenberg - and have prepared several scripts that together make up our analysis pipeline.

Our directory has the scripts and data files we we will be working with:

```
$ tree --noreport --charset=ascii zipf
```

```
|-- gutenberg
|   |-- austen-emma.txt
|   |-- austen-persuasion.txt
|   |-- austen-sense.txt
... ...
|-- merged.gp
`-- wordcount.jl
```

The first step is to count the frequency of each word in a book.

```
$ julia wordcount.jl count emma.dat gutenberg/austen-emma.txt
```

Let's take a quick peek at the result.

```
$ head -5 emma.dat
```

This shows us the top 5 lines in the output file:

```
                     to         5242           3.2363
                    the         5204           3.2128
                    and         4897           3.0233
                     of         4293           2.6504
                      i         3192           1.9707
```

We can see that the file consists of one row per word. Each row shows the word itself, the number of occurrences of that word, and the number of occurrences as a percentage of the total number of words in the text file.

We can do the same thing for a different book:

```
$ julia wordcount.jl count persuasion.dat \
       gutenberg/austen-persuasion.txt
$ head -5 persuasion.dat
```

```
                    the         3329           3.9573
                     to         2808           3.3380
                    and         2801           3.3296
                     of         2570           3.0551
                      a         1595           1.8960
```

Let's visualize the results.

```
$ julia wordcount.jl merge austen.res emma.dat persuasion.dat
$ gnuplot -c merged.gp pdf austen
```

The graph prodiced bay the last command is shown in Fig. **??**.

As we see, the general distribution tendency is not too far off from Zipf's law.

Together these scripts implement a common workflow:

1. Read a data file.
2. Perform an analysis on this data file.

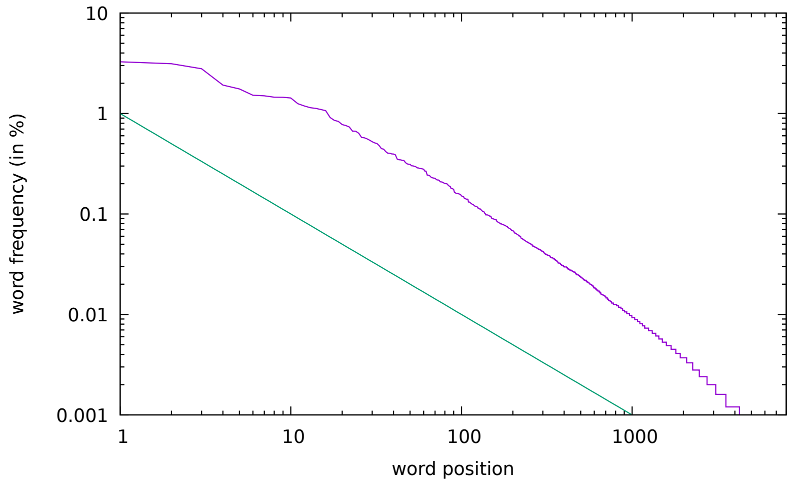


Figure 2.1: Word frequency distribution in Jane Austen novels.

3. Write the analysis results to a new file.
4. Plot a graph of the analysis results.
5. Save the graph as an image, so we can put it in a paper.
6. Make a summary table of the analyses

Running `wordcount.py` and `plotcount.py` at the shell prompt, as we have been doing, is fine for one or two files. If, however, we had 5 or 10 or 20 text files, or if the number of steps in the pipeline were to expand, this could turn into a lot of work. Plus, no one wants to sit and wait for a command to finish, even just for 30 seconds.

The most common solution to the tedium of data processing is to write a shell script that runs the whole pipeline from start to finish.

Using your text editor of choice, add the following to a new file named `run_pipeline.sh`.

```
# USAGE: bash run_pipeline.sh
# Zipf's law tests

$ julia wordcount.jl count emma.dat gutenberg/austen-emma.txt
$ julia wordcount.jl count persuasion.dat \
        gutenberg/austen-persuasion.txt

# Generate summary table
$ julia wordcount.jl merge austen.res austen-emma.dat \
        austen-persuasion.dat

# Plot the results
$ gnuplot -c merged.gp pdf austen
```

Runs the script and check that the output is the same as before:

```
$ sh run_pipeline.sh
```

This shell script solves several problems in computational reproducibility:

1. It explicitly documents our pipeline, making communication with colleagues (and our future selves) more efficient.
2. It allows us to type a single command, `sh run_pipeline.sh`, to reproduce the full analysis.
3. It prevents us from *repeating* typos or mistakes. You might not get it right the first time, but once you fix something it'll stay fixed.

Despite these benefits it has a few shortcomings.

Let's suppose that we adjusted the axis labels in our graph. Now we want to recreate our figures. We *could* just `sh run_pipeline.sh` again. That would work, but it could also be a big pain if counting words takes more than a few seconds. The word counting routine hasn't changed; we shouldn't need to recreate those files.

Alternatively, we could manually rerun the plotting for each word-count file.

With this approach, however, we don't get many of the benefits of having a shell script in the first place.

Another popular option is to comment out a subset of the lines in `run_pipeline.sh`:

Then, we would run our modified shell script using `sh run_pipeline.sh`.

But commenting out these lines, and subsequently uncommenting them, can be a hassle and source of errors in complicated pipelines.

What we really want is an executable *description* of our pipeline that allows software to do the tricky part for us: figuring out what steps need to be rerun.

Make was developed by Stuart Feldman in 1977 as a Bell Labs summer intern, and remains in widespread use today. Make can execute the commands needed to run our analysis and plot our results. Like shell scripts it allows us to execute complex sequences of commands via a single shell command. Unlike shell scripts it explicitly records the dependencies between files - what files are needed to create what other files - and so can determine when to recreate our data files or image files, if our text files change. Make can be used for any commands that follow the general pattern of processing files to create new files, for example:

- Run analysis scripts on raw data files to get data files that summarise the raw data (e.g. creating files with word counts from book text).
- Run visualisation scripts on data files to produce plots (e.g. creating images of word counts).
- Parse and combine text files and plots to create papers.
- Compile source code into executable programs or libraries.

Researchers are also finding Make of use in implementing reproducible research workflows, automating data analysis and visualisation and combining plots with text to produce reports and papers for publication.

2.2 Makefiles

In this section we (a) learn about the key parts of a Makefile: rules, targets, dependencies and actions, (b) write a simple Makefile, (c) run Make from the shell, (d) explain when and why to mark targets as .PHONY, (e) explain constraints on dependencies.

Create a file, called `Makefile`, with the following content:

```
# Count words.
blake-poems.dat : gutenberg/blake-poems.txt
	julia wordcount.jl count blake-poems.dat \
	      gutenberg/blake-poems.txt
```

This is a file executed by Make. Note how it resembles one of the lines from our shell script.

Let us go through each line in turn:

- # denotes a *comment*. Any text from # to the end of the line is ignored by Make.
- `blake-poems.dat` is a *target*, a file to be created, or built.
- `gutenberg/blake-poems.txt` is a *dependency*, a file that is needed to build or update the target. Targets can have zero or more dependencies.
- A colon, `:`, separates targets from dependencies.
- `julia wordcount.jl count blake-poems.dat gutenberg/blake-poems.txt` is an *action*, a command to run to build or update the target using the dependencies. Targets can have zero or more actions. These actions form a recipe to build the target from its dependencies and can be considered to be a shell script.
- Actions are indented using the TAB character, *not* 8 spaces.
- Together, the target, dependencies, and actions form a rule.

Our rule above describes how to build the target `blake-poems.dat` using the action `julia wordcount.jl` and the dependency `gutenberg/blake-poems.txt`.

Information that was implicit in our shell script - that we are generating a file called `blake-poems.dat` and that creating this file requires `gutenberg/blake-poems.txt` - is now made explicit by Make's syntax.

Let's first sure we start from scratch and delete the `.dat` and `.pdf` files we created earlier:

```
$ rm *.dat *.pdf
```

By default, Make looks for a Makefile, called `Makefile`, and we can run Make as follows:

```
$ make
```

By default, Make prints out the actions it executes:

```
julia wordcount.jl count blake-poems.dat gutenberg/blake-poems.txt
```

If we see,

```
Makefile:3: *** missing separator.  Stop.
```

then we have used a space instead of a TAB characters to indent one of our actions.

Let's see if we got what we expected.

```
head -5 blake-poems.dat
```

The first 5 lines of `isles.dat` should look exactly like before.

We don't have to call our Makefile `Makefile`. However, if we call it something else we need to tell Make where to find it. This we can do using `-f` flag. For example, if our Makefile is named `MyOtherMakefile`:

```
$ make -f MyOtherMakefile
```

When we re-run our Makefile, Make now informs us that:

```
make: ‘blake-poems.dat’ is up to date.
```

This is because our target, `isles.dat`, has now been created, and Make will not create it again. To see how this works, let's pretend to update one of the text files. Rather than opening the file in an editor, we can use the shell `touch` command to update its timestamp (which would happen if we did edit the file):

```
$ touch gutenberg/blake-poems.txt
```

If we compare the timestamps of `gutenberg/blake-poems.txt` and `blake-poems.dat`,

```
$ ls -l gutenberg/blake-poems.txt blake-poems.dat
```

then we see that `blake-poems.dat`, the target, is now older than `gutenberg/blake-poems.txt`, its dependency:

```
-rw-r--r--  1 mjj  mjj   323972 Jun 12 10:35 gutenberg/blake-poems
   .txt
-rw-r--r--  1 mjj  mjj   182273 Jun 12 09:58 blake-poems.dat
```

If we run Make again,

```
$ make
```

then it recreates `blake-poems.dat`:

```
julia wordcount.jl count blake-poems.dat gutenberg/blake-poems.txt
```

When it is asked to build a target, Make checks the 'last modification time' of both the target and its dependencies. If any dependency has been updated since the target, then the actions are re-run to update the target. Using this approach, Make knows to only rebuild the files that, either directly or indirectly, depend on the file that changed. This is called an *incremental build*.

2.2.1 Makefiles as documentation

By explicitly recording the inputs to and outputs from steps in our analysis and the dependencies between files, Makefiles act as a type of documentation, reducing the number of things we have to remember.

Let's add another rule to the end of `Makefile`:

```
abyss.dat : gutenberg/bryant-stories.txt
        julia wordcount.jl count bryant-stories.dat gutenberg/
           bryant-stories.txt
```

If we run Make,

```
$ make
```

then we get:

```
make: 'blake-poems.dat' is up to date.
```

Nothing happens because Make attempts to build the first target it finds in the Makefile, the *default target*, which is `blake-poems.dat` which is already up-to-date. We need to explicitly tell Make we want to build `abyss.dat`:

```
$ make bryant-stories.dat
```

Now, we get:

```
julia wordcount.jl count bryant-stories.dat gutenberg/bryant-
   stories.txt
```

We may want to remove all our data files so we can explicitly recreate them all. We can introduce a new target, and associated rule, to do this. We will call it `clean`, as this is a common name for rules that delete auto-generated files, like our `.dat` files:

```
clean :
        rm -f *.dat
```

This is an example of a rule that has no dependencies. `clean` has no dependencies on any `.dat` file as it makes no sense to create these just to remove them. We just want to remove the data files whether or not they exist. If we run Make and specify this target,

```
$ make clean
```

then we get:

```
rm -f *.dat
```

There is no actual thing built called `clean`. Rather, it is a short-hand that we can use to execute a useful sequence of actions. Such targets, though very useful, can lead to problems. For example, let us recreate our data files, create a directory called `clean`, then run Make:

```
$ make blake-poems.dat bryant-stories.dat
$ mkdir clean
$ make clean
```

We get:

```
make: ‘clean’ is up to date.
```

Make finds a file (or directory) called `clean` and, as its `clean` rule has no dependencies, assumes that `clean` has been built and is up-to-date and so does not execute the rule's actions. As we are using `clean` as a short-hand, we need to tell Make to always execute this rule if we run `make clean`, by telling Make that this is a *phony target*, that it does not build anything. This we do by marking the target as `.PHONY`:

```
.PHONY : clean
clean :
        rm -f *.dat
```

If we run Make,

```
$ make clean
```

then we get:

```
rm -f *.dat
```

We can add a similar command to create all the data files. We can put this at the top of our Makefile so that it is the *default target*, which is executed by default if no target is given to the `make` command:

```
.PHONY : dats
dats : blake-poems.dat bryant-stories.dat
```

This is an example of a rule that has dependencies that are targets of other rules. When Make runs, it will check to see if the dependencies exist and, if not, will see if rules are available that will create these. If such rules exist it will invoke these first, otherwise Make will raise an error.

This rule is also an example of a rule that has no actions. It is used purely to trigger the build of its dependencies, if needed.

If we run,

```
$ make dats
```

then Make creates the data files:

```
julia wordcount.jl count blake-poems.dat gutenberg/blake-poems.txt
julia wordcount.jl count bryant-stories.dat gutenberg/bryant-
   stories.txt
```

If we run `dats` again,

```
$ make dats
```

then Make sees that the data files exist:

```
make: Nothing to be done for 'dats'.
```

Our Makefile now looks like this:

```
# Count words.
.PHONY : dats
dats : blake-poems.dat bryant-stories.dat

blake-poems.dat : gutenberg/blake-poems.txt
        julia wordcount.jl count blake-poems.dat gutenberg/blake-
           poems.txt

bryant-stories.dat : gutenberg/bryant-stories.txt
        julia wordcount.jl count bryant-stories.dat gutenberg/
           bryant-stories.txt

.PHONY : clean
clean :
        rm -f *.dat
```

The following figure shows a graph of the dependencies embodied within our Makefile, involved in building the `dats` target:

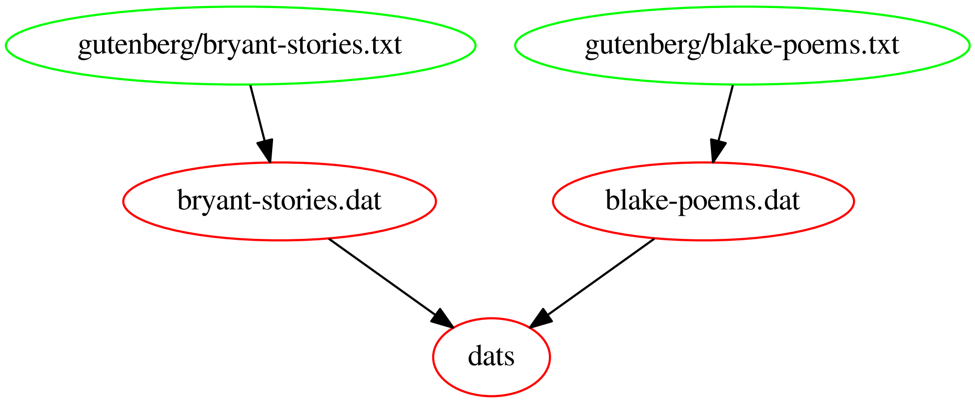


Figure 2.2: Dependencies represented within the Makefile

2.2.2 Questions

- Write a new rule for `last.dat`, created from `gutenberg/last.txt`.
- Update the `dats` rule with this target.
- Write a new rule for `merge`, which creates the summary table. The rule needs to:
- - Depend upon each of the three `.dat` files.
  - Invoke the action `julia wordcount.jl merge results.txt bryant-stories.dat isles.dat last.dat`.

  Put this rule at the top of the Makefile so that it is the default target.
- Update `clean` so that it removes `results.txt`.

The following figure shows the dependencies embodied within our Makefile, involved in building the `results.txt` target:

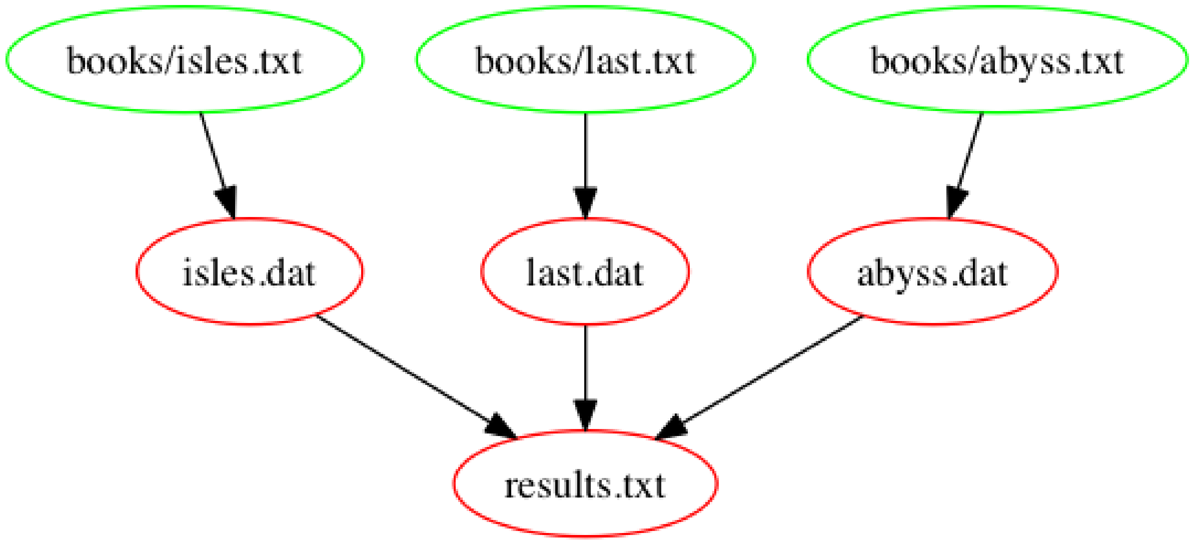


Figure 2.3: results.txt dependencies represented within the Makefile

2.3 Automatic variables

In this section we learn to (a) use Make automatic variables to remove duplication in a Makefile, (b) use $@ to refer to the target of the current rule, (c) use $^ to refer to the dependencies of the current rule, (d) use $< to refer to the first dependency of the current rule, and (e) explain why bash wild-cards in dependencies can cause problems.

After the exercise at the end of the previous section, our Makefile look like this:

```
# Generate summary table.
results.txt : isles.dat abyss.dat last.dat
        python zipf_test.py abyss.dat isles.dat last.dat > results
           .txt

# Count words.
.PHONY : dats
dats : isles.dat abyss.dat last.dat

isles.dat : books/isles.txt
        python wordcount.py books/isles.txt isles.dat

abyss.dat : books/abyss.txt
        python wordcount.py books/abyss.txt abyss.dat

last.dat : books/last.txt
        python wordcount.py books/last.txt last.dat

.PHONY : clean
clean :
        rm -f *.dat
        rm -f results.txt
```

Our Makefile has a lot of duplication. For example, the names of text files and data files are repeated in many places throughout the Makefile. Makefiles are a form of code and, in any code, repeated code can lead to problems e.g. we rename a data file in one part of the Makefile but forget to rename it elsewhere.

2.3.1 D.R.Y. (Don't Repeat Yourself)

In many programming languages, the bulk of the language features are there to allow the programmer to describe long-winded computational routines as short, expressive, beautiful code. Features like user-defined variables and functions are useful in part because they mean we don't have to write out (or think about) all of the details over and over again. This good habit of writing things out only once is known as the "Don't Repeat Yourself" principle or D.R.Y.

Let us set about removing some of the repetition from our Makefile.

In our `results.txt` rule we duplicate the data file names and the name of the results file name:

```
results.txt : isles.dat abyss.dat last.dat
      python zipf_test.py abyss.dat isles.dat last.dat>results.txt
```

Looking at the results file name first, we can replace it in the action with $@:

```
results.txt : isles.dat abyss.dat last.dat
        python zipf_test.py abyss.dat isles.dat last.dat > $@
```

$@ is a Make *automatic variable* which means "the target of the current rule". When Make is run it will replace this variable with the target name.

We can replace the dependencies in the action with $^:

```
results.txt : isles.dat abyss.dat last.dat
        python zipf_test.py $^ > $@
```

$^ is another automatic variable which means 'all the dependencies of the current rule'. Again, when Make is run it will replace this variable with the dependencies.

Let's update our text files and re-run our rule:

```
$ touch books/*.txt
$ make results.txt
```

We get:

```
python wordcount.py books/isles.txt isles.dat
python wordcount.py books/abyss.txt abyss.dat
python wordcount.py books/last.txt last.dat
python zipf_test.py isles.dat abyss.dat last.dat > results.txt
```

We can use the bash wild-card in our dependency list:

```
results.txt : *.dat
        python zipf_test.py $^ > $@
```

Let's update our text files and re-run our rule:

```
$ touch books/*.txt
$ make results.txt
```

We get the same as above.

Now let's delete the data files and re-run our rule:

```
$ make clean
$ make results.txt
```

We get:

```
make: *** No rule to make target '*.dat', needed by 'results.txt'.
     Stop.
```

As there are no files that match the pattern `*.dat` the name `*.dat` is used as a file name itself, and there is no file matching that, nor any rule so we get an error. We need to explicitly rebuild the `.dat` files first:

```
$ make dats
$ make results.txt
```

2.3.2 Questions

Update dependencies. What will happen if you now execute:

```
$ touch *.dat
$ make results.txt
```

1. nothing
2. all files recreated
3. only `.dat` files recreated
4. only `results.txt` recreated

Rewrite .dat rules to use automatic variables. As we saw, `$^` means 'all the dependencies of the current rule'. This works well for `results.txt` as its action treats all the dependencies the same - as the input for the `zipf_test.py` script.

However, for some rules, we may want to treat the first dependency differently. For example, our rules for `.dat` use their first (and only) dependency specifically as the input file to `wordcount.py`. If we add additional dependencies (as we will soon do) then we don't want these being passed as input files to `wordcount.py` as it expects only one input file to be named when it is invoked.

Make provides an automatic variable for this, `$<` which means 'the first dependency of the current rule'.

Rewrite each `.dat` rule to use the automatic variables `$@` ('the target of the current rule') and `$<` ('the first dependency of the current rule').

2.4 Dependencies on data and code

In this section we consider the case when (a) output files depend not only upon input files but on the scripts or code that created the output files, and (b) recognise and avoid false dependencies.

Our Makefile now looks like this:

```
# Generate summary table.
results.txt : *.dat
	python zipf_test.py $^ > $@

# Count words.
.PHONY : dats
dats : isles.dat abyss.dat last.dat

isles.dat : books/isles.txt
	python wordcount.py $< $@

abyss.dat : books/abyss.txt
	python wordcount.py $< $@

last.dat : books/last.txt
	python wordcount.py $< $@

.PHONY : clean
clean :
	rm -f *.dat
	rm -f results.txt
```

Our data files are a product not only of our text files but the script, `wordcount.py`, that processes the text files and creates the data files. A change to `wordcount.py` (e.g. to add a new column of summary data or remove an existing one) results in changes to the `.dat` files it outputs. So, let's pretend to edit `wordcount.py`, using `touch`, and re-run Make:

```
$ make dats
$ touch wordcount.py
$ make dats
```

Nothing happens! Though we've updated `wordcount.py` our data files are not updated because our rules for creating `.dat` files don't record any dependencies on `wordcount.py`.

We need to add `wordcount.py` as a dependency of each of our data files also:

```
isles.dat : books/isles.txt wordcount.py
        python wordcount.py $< $@

abyss.dat : books/abyss.txt wordcount.py
        python wordcount.py $< $@

last.dat : books/last.txt wordcount.py
        python wordcount.py $< $@
```

If we pretend to edit `wordcount.py` and re-run Make,

```
$ touch wordcount.py
$ make dats
```

then we get:

```
python wordcount.py books/isles.txt isles.dat
python wordcount.py books/abyss.txt abyss.dat
python wordcount.py books/last.txt last.dat
```

The following figure shows the dependencies embodied within our Makefile, involved in building the `results.txt` target, after adding `wordcount.py` as a dependency to the `.dat` files:

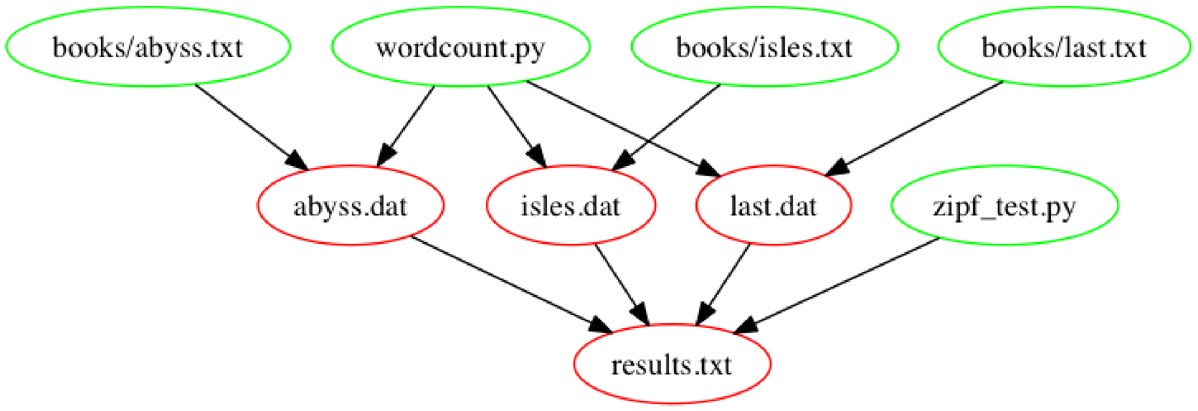


Figure 2.4: results.txt dependencies after adding wordcount.py as a dependency

Intuitively, we should also add `wordcount.py` as dependency for `results.txt`, as the final table should be rebuilt as we remake the `.dat` files. However, it turns out we don't have to! Let's see what happens to `results.txt` when we update `wordcount.py`:

```
$ touch wordcount.py
$ make results.txt
```

then we get:

```
python wordcount.py books/abyss.txt abyss.dat
python wordcount.py books/isles.txt isles.dat
python wordcount.py books/last.txt last.dat
python zipf_test.py abyss.dat isles.dat last.dat > results.txt
```

The whole pipeline is triggered, even the creation of the `results.txt` file! To understand this, note that according to the dependency figure, `results.txt` depends on the `.dat` files. The update of `wordcount.py` triggers an update of the `*.dat` files. Thus, `make` sees that the dependencies (the `.dat` files) are newer than the target file (`results.txt`) and thus it recreates `results.txt`. This is an example of the power of `make`: updating a subset of the files in the pipeline triggers rerunning the appropriate downstream steps.

2.4.1 Questions

Updating one input file What will happen if you now execute:

```
$ touch books/last.txt
$ make results.txt
```

1. only `last.dat` is recreated
2. all `.dat` files are recreated
3. only `last.dat` and `results.txt` are recreated
4. all `.dat` and `results.txt` are recreated

Wordcount as dependency of results.txt. What would happen if you actually added `wordcount.py` as dependency of `results.txt`, and why?

We still have to add the `zipf-test.py` script as dependency to `results.txt`. Given the answer to the challenge above, we cannot use `$^` for the rule. We'll go back to using `*.dat`:

```
results.txt : *.dat zip_test.py
        python zipf_test.py *.dat > $@
```

2.5 Pattern rules

In this section we (a) write Make pattern rules, (b) use the Make wild-card `%` in targets and dependencies, and (c) use the Make special variable `$*` in actions.

Our Makefile still has repeated content. The rules for each `.dat` file are identical apart from the text and data file names. We can replace these rules with a single *pattern rule* which can be used to build any `.dat` file from a `.txt` file in `books/`:

```
%.dat : books/%.txt wordcount.py
        python wordcount.py $< $*.dat
```

`%` is a Make *wild-card*. `$*` is a special variable which gets replaced by the *stem* with which the rule matched.

This rule can be interpreted as:

In order to build a file named `[something].dat` (the target) find a file named `books/[that same something].txt` (the dependency) and run `wordcount.py [the dependency] [the target]`.

If we re-run Make,

```
$ make clean
$ make dats
```

then we get:

```
python wordcount.py books/isles.txt isles.dat
python wordcount.py books/abyss.txt abyss.dat
python wordcount.py books/last.txt last.dat
```

The Make `%` wild-card can only be used in a target and in its dependencies. It cannot be used in actions. In actions, you may however use `$*`, which will be replaced by the stem with which the rule matched.

Our Makefile is now much shorter and cleaner:

```
# Generate summary table.
results.txt : *.dat zipf_test.py python
        zipf_test.py *.dat > $@

# Count words.
.PHONY : dats
dats : isles.dat abyss.dat last.dat

% .dat : books/%.txt wordcount.py
        python wordcount.py $< $*.dat

.PHONY : clean
clean :
        rm -f *.dat
        rm -f results.txt
```

2.6 Variables

(a) Use variables in a Makefile. (b) Assign values to variables. (c) Reference variables.

Despite our efforts, our Makefile still has repeated content, namely the name of our script, `wordcount.py`. If we renamed our script we'd have to update our Makefile in multiple places.

We can introduce a Make *variable* (called a *macro* in some versions of Make) to hold our script name:

```
COUNT_SRC=wordcount.py
```

This is a variable *assignment* - `COUNT_SRC` is assigned the value `wordcount.py`.

`wordcount.py` is our script and it is invoked by passing it to `python`. We can introduce another variable to represent this execution:

```
COUNT_EXE=python $(COUNT_SRC)
```

`$(...)` tells Make to replace a variable with its value when Make is run. This is a variable *reference*. At any place where we want to use the value of a variable we have to write it, or reference it, in this way.

Here we reference the variable `COUNT_SRC`. This tells Make to replace the variable `COUNT_SRC` with its value `wordcount.py`. When Make is run it will assign to `COUNT_EXE` the value `python wordcount.py`.

Defining the variable `COUNT_EXE` in this way allows us to easily change how our script is run (if, for example, we changed the language used to implement our script from Python to R).

2.6.1 Use variables

Update `Makefile` so that the `%.dat` rule references the variables `COUNT_SRC` and `COUNT_EXE`. Then do the same for the `zipf-test.py` script and the `results.txt` rule, using `ZIPF_SRC` and `ZIPF_EXE` as variable names

2.7 Functions

In this section we (a) use Make's `wildcard` function to get lists of files matching a pattern, and (b) use Make's `patsubst` function to rewrite file names.

At this point, we have the following Makefile:

```
# Count words script.
COUNT_SRC=wordcount.py
COUNT_EXE=python $(COUNT_SRC)

# Test Zipf's rule
ZIPF_SRC=zipf_test.py
ZIPF_EXE=python $(ZIPF_SRC)

# Generate summary table.
results.txt : *.dat $(ZIPF_SRC)
        $(ZIPF_EXE) *.dat > $@

# Count words.
.PHONY : dats
dats : isles.dat abyss.dat last.dat

%.dat : books/%.txt $(COUNT_SRC)
        $(COUNT_EXE) $< $*.dat

.PHONY : clean
clean :
        rm -f *.dat
        rm -f results.txt
```

Make has many *functions* which can be used to write more complex rules. One example is `wildcard`. `wildcard` gets a list of files matching some pattern, which we can then save in a variable. So, for example, we can get a list of all our text files (files ending in `.txt`) and save these in a variable by adding this at the beginning of our makefile:

```
TXT_FILES=$(wildcard books/*.txt)
```

We can add `.PHONY` target and rule to show the variable's value:

```
.PHONY : variables
variables:
    @echo TXT_FILES: $(TXT_FILES)
```

Make prints actions as it executes them. Using `@` at the start of an action tells Make not to print this action. So, by using `@echo` instead of `echo`, we can see the result of `echo` (the variable's value being printed) but not the `echo` command itself.

If we run Make:

```
$ make variables
```

We get:

```
TXT_FILES: books/abyss.txt books/isles.txt books/last.txt books/
   sierra.txt
```

Note how `sierra.txt` is now included too.

The following figure shows the dependencies embodied within our Makefile, involved in building the `results.txt` target, once we have introduced our function:

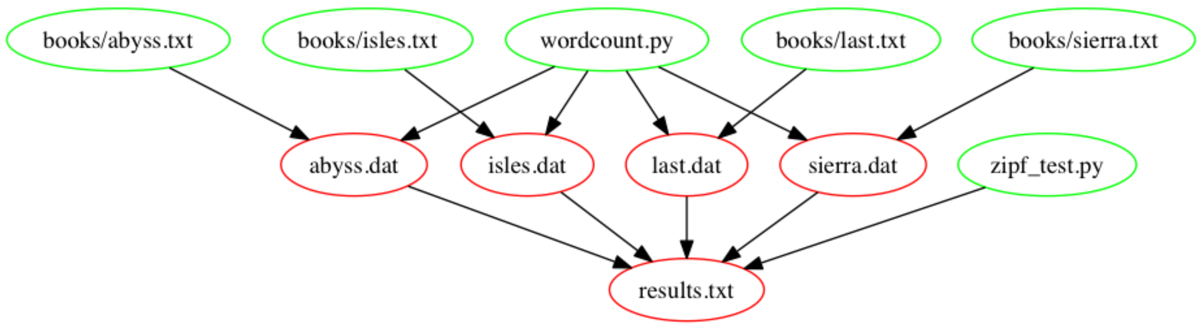


Figure 2.5: `results.txt` dependencies after introducing a function

`patsubst` ('pattern substitution') takes a pattern, a replacement string and a list of names in that order; each name in the list that matches the pattern is replaced by the replacement string. Again, we can save the result in a variable. So, for example, we can rewrite our list of text files into a list of data files (files ending in `.dat`) and save these in a variable:

```
DAT_FILES=$(patsubst books/%.txt, %.dat, $(TXT_FILES))
```

We can extend `variables` to show the value of `DAT_FILES` too:

```
.PHONY : variables
variables:
        @echo TXT_FILES: $(TXT_FILES)
        @echo DAT_FILES: $(DAT_FILES)
```

If we run Make,

```
$ make variables
```

then we get:

```
TXT_FILES: books/abyss.txt books/isles.txt books/last.txt books/
   sierra.txt
DAT_FILES: abyss.dat isles.dat last.dat sierra.dat
\end{verbatim}

Now, \texttt{sierra.txt} is processed too.

With these we can rewrite \texttt{clean} and \texttt{dats}:

\begin{lstlisting}[style=makefile]
.PHONY : dats
dats : $(DAT_FILES)

.PHONY : clean
clean :
        rm -f $(DAT_FILES)
        rm -f results.txt
```

Let's check:

```
$ make clean
$ make dats
```

We get:

```
python wordcount.py books/abyss.txt abyss.dat
python wordcount.py books/isles.txt isles.dat
python wordcount.py books/last.txt last.dat
python wordcount.py books/sierra.txt sierra.dat
```

We can also rewrite `results.txt`:

```
results.txt : $(DAT_FILES) $(ZIPF_SRC)
        $(ZIPF_EXE) *.dat > $@
```

If we re-run Make:

```
$ make clean
$ make results.txt
```

We get:

```
python wordcount.py books/abyss.txt abyss.dat
python wordcount.py books/isles.txt isles.dat
python wordcount.py books/last.txt last.dat
python wordcount.py books/sierra.txt sierra.dat
python zipf_test.py *.dat > results.txt
```

We see that the problem we had when using the bash wild-card, `*.dat`, which required us to run `make dats` before `make results.txt` has now disappeared, since our functions allow us to create `.dat` file names from those `.txt` file names in `books/`.

Let's check the `results.txt` file:

```
$ cat results.txt
```

```
Book    First   Second  Ratio
abyss   4044    2807    1.44
isles   3822    2460    1.55
last    12244   5566    2.20
sierra  4242    2469    1.72
```

So the range of the ratios of occurrences of the two most frequent words in our books is indeed around 2, as predicted by Zipf's law: most frequently-occurring word occurs approximately twice as often as the second most frequent word

Here is our final Makefile:

```
# Count words script.
COUNT_SRC=wordcount.py
COUNT_EXE=python $(COUNT_SRC)

# Test Zipf's rule
ZIPF_SRC=zipf_test.py
ZIPF_EXE=python $(ZIPF_SRC)

TXT_FILES=$(wildcard books/*.txt)
DAT_FILES=$(patsubst books/%.txt, %.dat, $(TXT_FILES))

# Generate summary table.
results.txt : $(DAT_FILES) $(ZIPF_SRC)
    $(ZIPF_EXE) *.dat > $@

# Count words.
.PHONY : dats
dats : $(DAT_FILES)

%.dat : books/%.txt $(COUNT_SRC)
    $(COUNT_EXE) $< $*.dat

.PHONY : clean
clean :
    rm -f $(DAT_FILES)
    rm -f results.txt

.PHONY : variables
variables:
    @echo TXT_FILES: $(TXT_FILES)
    @echo DAT_FILES: $(DAT_FILES)
```

2.8 Self-documenting Makefiles

In this section we learn about writing self-documenting Makefiles with built-in help.

Many linux commands, and programs that people have written that can be run from within shell, support a `--help` flag to display more information on how to use the commands or programs. In this spirit, it can be useful, both for ourselves and for others, to provide a `help` target in our Makefiles. This can provide a summary of the names of the key targets and what they do, so we don't need to look at the Makefile itself unless we want to. For our Makefile, running a `help` target might print:

```
$ make help
results.txt : Generate Zipf summary table.
dats        : Count words in text files.
clean       : Remove auto-generated files.
```

So, how would we implement this? We could write a rule like:

```
.PHONY : help
help :
        @echo "results.txt : Generate Zipf summary table."
        @echo "dats        : Count words in text files."
        @echo "clean       : Remove auto-generated files."
```

But every time we add or remove a rule, or change the description of a rule, we would have to update this rule too. It would be better if we could keep the descriptions of the rules by the rules themselves and extract these descriptions automatically.

The linux can help us here. It provides a command called *sed* which stands for 'stream editor'. `sed` reads in some text, does some filtering, and writes out the filtered text.

So, we could write comments for our rules, and mark then up in a way which `sed` can detect. Since Make uses # for comments, we can use ## for comments that describe what a rule does and that we want `sed` to detect. For example:

```
## results.txt : Generate Zipf summary table.
results.txt : $(DAT_FILES) $(ZIPF_SRC)
        $(ZIPF_EXE) *.dat > $@

## dats        : Count words in text files.
.PHONY : dats
dats : $(DAT_FILES)

%.dat : books/%.txt $(COUNT_SRC)
        $(COUNT_EXE) $< $*.dat

## clean       : Remove auto-generated files.
.PHONY : clean
clean :
        rm -f $(DAT_FILES)
        rm -f results.txt

## print       : Print variables.
.PHONY : variables
variables:
        @echo TXT_FILES: $(TXT_FILES)
        @echo DAT_FILES: $(DAT_FILES)
```

We use ## so we can distinguish between comments that we want sed to automatically filter, and other comments that may describe what other rules do, or that describe variables.

We can then write a help target that applies sed to our Makefile:

```
.PHONY : help
help : Makefile
        @sed -n 's/^##//p' $<
```

This rule depends upon the Makefile itself. It runs sed on the first dependency of the rule, which is our Makefile, and tells sed to get all the lines that begin with ##, which sed then prints for us.

If we now run

```
$ make help
```

we get:

```
results.txt : Generate Zipf summary table.
dats        : Count words in text files.
clean       : Remove auto-generated files.
print       : Print variables.
```

If we add, change or remove a target or rule, we now only need to remember to add, update or remove a comment next to the rule. So long as we respect our convention of using ## for such comments, then our `help` rule will take care of detecting these comments and printing them for us.

Chapter 3

The Unix Shell

3.1 Introduction

In this section we (a) explain how the shell relates to the keyboard, the screen, the operating system, and users' programs, and (b) explain when and why command-line interfaces should be used instead of graphical interfaces.

The Unix shell has been around longer than most of its users have been alive. It has survived so long because it's a power tool that allows people to do complex things with just a few keystrokes. More importantly, it helps them automate repetitive tasks so they aren't typing the same things (or clicking the same buttons) over and over again. Use of the shell is fundamental to using a wide range of other powerful tools and computing resources (including "high-performance computing"). These lessons will start you on a path towards using these resources effectively.

3.1.1 Prerequisites

This lesson guides you through the basics of file systems and the shell. If you have stored files on a computer at all and recognize the word "file" and either "directory" or "folder" (two common words for the same thing), you're ready for this lesson.

You need to download some files to follow this lesson:

1. Download shell-novice-data.zip and move the file to your Desktop.
2. Unzip/extract the file (ask your instructor if you need help with this step). You should end up with a new folder called data-shell on your Desktop.

3. Open a terminal and type:

```
$ cd
```

3.1.2 Introducing the Shell

At a high level, computers do four things:

- run programs
- store data
- communicate with each other
- interact with us

They can do the last of these in many different ways, including direct brain-computer links and speech interfaces. Since these are still in their infancy, most of us use windows, icons, mice, and pointers. These technologies didn't become widespread until the 1980s.

From the 1950s to the 1980s, most people used line printers to get their results from computers. Those devices only allowed output of the letters, numbers, and punctuation found on a standard keyboard, so programming languages and interfaces had to be designed around that constraint.

This kind of interface is called a **command-line interface**, or CLI, to distinguish it from a **graphical user interface**, or GUI, which most people now use. The heart of a CLI is a **read-evaluate-print loop**, or REPL: when the user types a command and then presses the Enter key, the computer reads it, executes it, and prints its output. The user then types another command, and so on until the user logs off.

This description makes it sound as though the user sends commands directly to the computer, and the computer sends output directly to the user. In fact, there is usually a program in between called a **command shell**. What the user types goes into the shell, which then figures out what commands to run and orders the computer to execute them. Note, the shell is called *the shell* because it encloses the operating system in order to hide some of its complexity and make it simpler to interact with.

A shell is a program like any other. What's special about it is that its job is to run other programs rather than to do calculations itself. The most popular Unix shell is Bash, the Bourne Again SHell (so-called because it's derived from a shell written by Stephen Bourne). Bash is the default shell on most modern implementations of Unix and in most packages that provide Unix-like tools for Windows.

Using Bash or any other shell sometimes feels more like programming than like using a mouse. Commands are terse (often only a couple of characters long), their names are frequently cryptic, and their output is lines of text rather than something visual like a graph. On the other hand, the shell allows us to combine existing tools in powerful ways with only a few keystrokes and to set up pipelines to handle large volumes of data automatically thus improving productivity and reproducibility. In addition, the command line is often the easiest way to interact with remote machines and supercomputers. Familiarity with the shell is near essential to run a variety of specialized tools and resources including high-performance computing systems. As clusters and cloud computing systems become more popular for scientific data crunching, being able to interact with them is becoming a necessary skill. We can build on the command-line skills covered here to tackle a wide range of scientific questions and computational challenges.

3.1.3 Nelle's Pipeline: Starting Point

Nelle Nemo, a marine biologist, has just returned from a six-month survey of the North Pacific Gyre, where she has been sampling gelatinous marine life. She has 300 samples in all, and now needs to:

1. Run each sample through an assay machine[1] that will measure the relative abundance of 300 different proteins. The machine's output for a single sample is a file with one line for each protein.
2. Calculate statistics for each of the proteins separately using a program called `goostat`.
3. Compare the statistics for each protein with corresponding statistics for each other protein using a program called `goodiff`.
4. Write up results. Her supervisor would really like her to do this by the end of the month so that her paper can appear in an upcoming special issue of *Aquatic Goo Letters*.

It takes about half an hour for the assay machine to process each sample. The good news is that it only takes two minutes to set each one up. Since her lab has eight assay machines that she can use in parallel, this step will "only" take about two weeks.

The bad news is that if she has to run `goostat` and `goodiff` by hand, she'll have to enter filenames and click "OK" 45,150 times (300 runs of `goostat`, plus 300*299/2 (half of 300

[1] Per Wikipedia, an assay is an analytic procedure in laboratory medicine, pharmacology, environmental biology and molecular biology for qualitatively assessing or quantitatively measuring the presence or amount or the functional activity of a target entity.

times 299) runs of `goodiff`). At 30 seconds each, that will take more than two weeks. Not only would she miss her paper deadline, the chances of her typing all of those commands right are practically zero.

The next few lessons will explore what she should do instead. More specifically, they explain how she can use a command shell to automate the repetitive steps in her processing pipeline so that her computer can work 24 hours a day while she writes her paper. As a bonus, once she has put a processing pipeline together, she will be able to use it again whenever she collects more data.

3.2 Files and Directories

In this section we (a) learn the similarities and differences between a file and a directory, (b) translate an absolute path into a relative path and vice versa, (c) construct absolute and relative paths that identify specific files and directories, (d) explain the steps in the shell's read-run-print cycle, (e) identify the actual command, flags, and filenames in a command-line call, (f) 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:

3.2.1 Preparation Magic

If you type the command: `PS1='$ '` into your shell, 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.

3.2.2 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 `/home/nelle`, which is Nelle's **home directory**:

```
$ pwd
```

```
/home/nelle
```

3.2.3 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:\Users\nelle`. (Note that it may look slightly different for different versions of Windows.)

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 `/home/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.

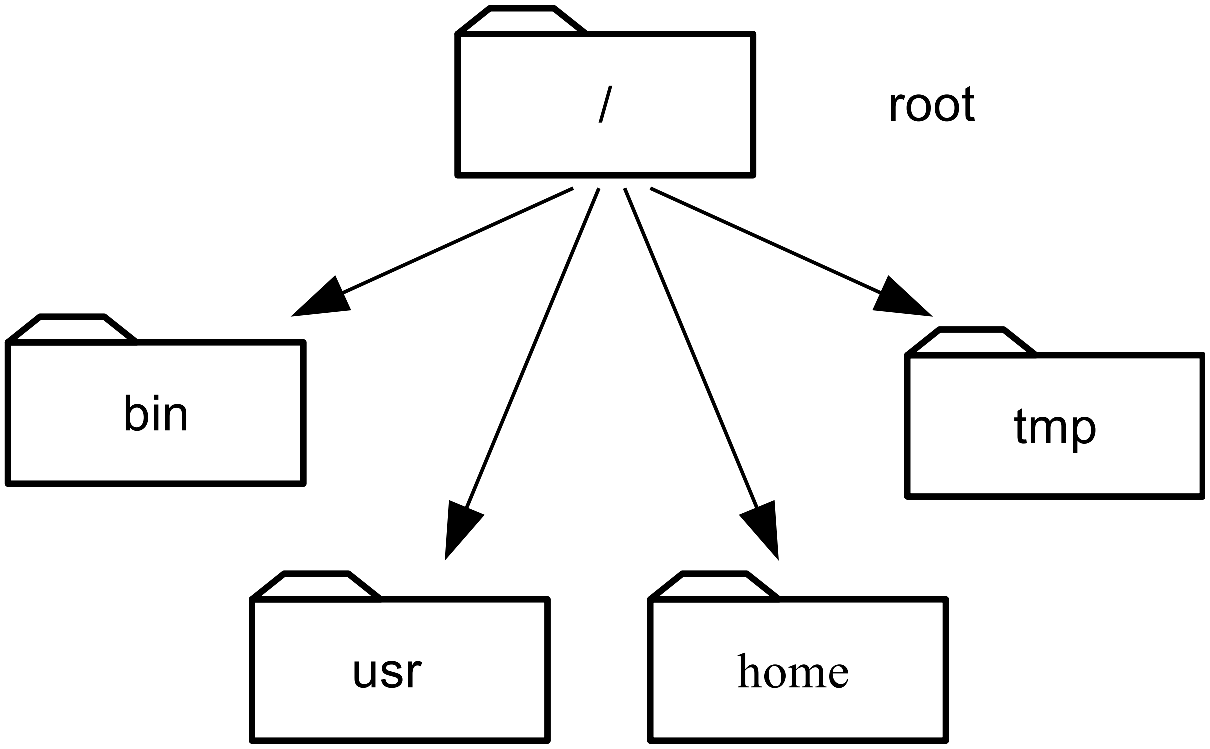


Figure 3.1: The File System

We know that our current working directory `/home/nelle` 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 `/`.

3.2.4 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 `/home`, 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 `/home/imhotep`, Wolfman's in `/home/larry`, and Nelle's in `/home/nelle`. Because Nelle is the user in our examples here, this is why we get `/home/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/
```

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.

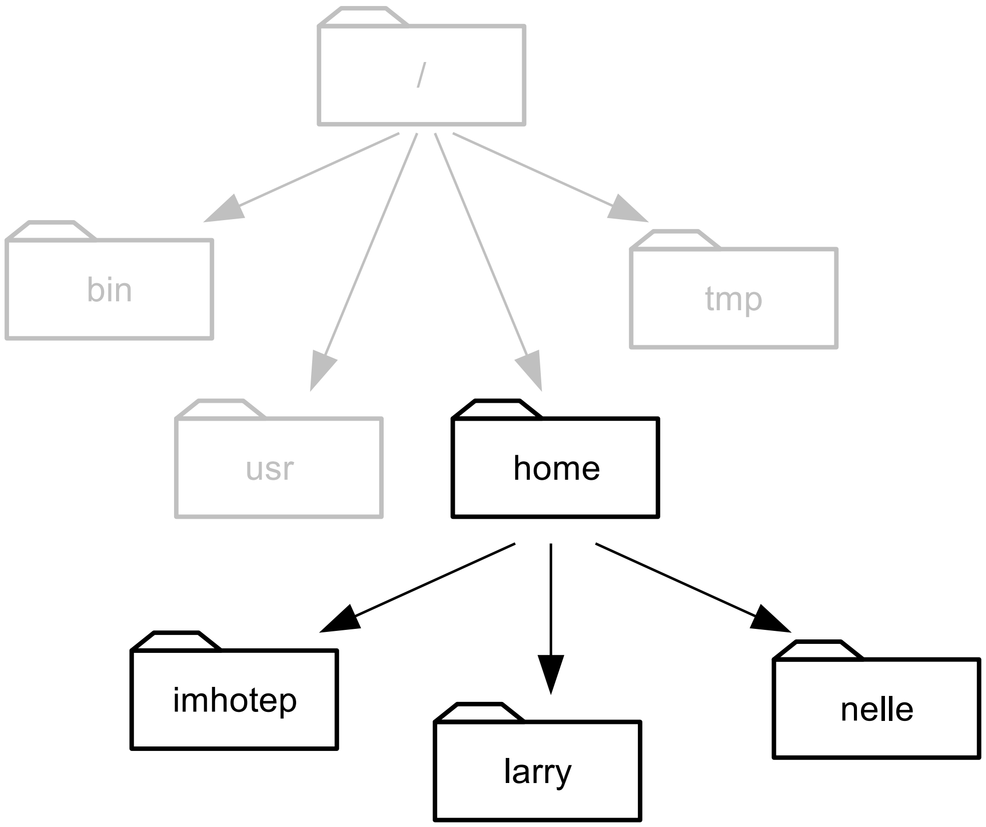


Figure 3.2: Home Directories

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 `/home/nelle/Desktop/data-shell/data`. If we run `ls` without arguments now, it lists the contents of `/home/nelle/Desktop/data-shell/data`, because that's where we now are:

```
$ pwd
/home/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 `/home/nelle/Desktop/data-shell`:

```
$ pwd
/home/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 /home/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.

3.2.5 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.

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
/home/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
```

```
/home/nelle/Desktop/data-shell/data
```

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

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

3.2.6 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 `/home/nelle`, then `~/data` is equivalent to `/home/nelle/data`. This only works if it is the first character in the path: `here/there/~/elsewhere` is *not* `here/there/home/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`.)

3.2.7 Output sorting

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.

If she is in her home directory, 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 \textbf{tab completion}. If she types:

\begin{lstlisting}[style=commandprompt]
$ 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.

3.2.8 Questions

Many ways to do the same thing - absolute vs relative paths: Starting from a filesystem location of `/home/amanda/data/`, which of the following commands could Amanda use to navigate to her home directory, which is `/home/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 `/home/thing`, what will `ls ../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** Assuming a directory structure as in the above Figure (File System for Challenge Questions), if `pwd` displays `/home/backup`, and `-r` tells `ls` to display things in reverse order, what command will display:

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

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

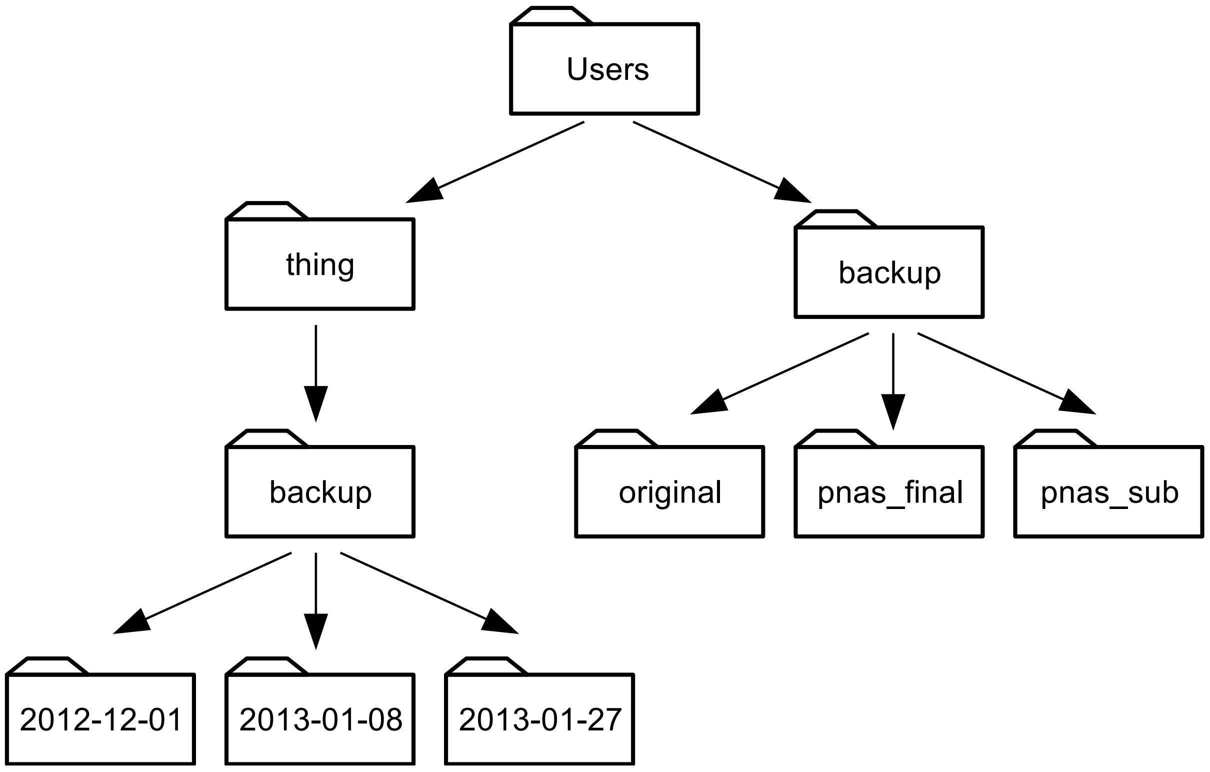


Figure 3.3: File System for Challenge Questions

Exploring more `ls` arguments What does the command `ls` do when used with the `-s` and `-h` arguments?

Listing Recursively and By Time The command ls -R lists the contents of directories recursively, i.e., lists their sub-directories, sub-sub-directories, and so on in alphabetical order at each level. The command ls -t lists things by time of last change, with most recently changed files or directories first. In what order does ls -R -t display things? (Hint: ls -l use a long listing format to view timestamps.)

3.3 Creating Things

In this section we learn how to (a) create a directory hierarchy that matches a given diagram, (b) create files in that hierarchy using an editor or by copying and renaming existing files, (c) display the contents of a directory using the command line, and (d) delete specified files and/or directories.

We now know how to explore files and directories, but how do we create them in the first place? Let's go back to our `data-shell` directory on the Desktop and use `ls -F` to see what it contains:

```
$ pwd
/home/nelle/Desktop/data-shell
```

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

Let's create a new directory called `thesis` using the command `mkdir thesis` (which has no output):

```
$ mkdir thesis
```

As you might (or might not) guess from its name, `mkdir` means "make directory". Since `thesis` is a relative path (i.e., doesn't have a leading slash), the new directory is created in the current working directory:

```
$ ls -F
creatures/  north-pacific-gyre/  thesis/
data/       notes.txt            writing/
Desktop/    pizza.cfg
molecules/  solar.pdf
```

However, there's nothing in it yet:

```
$ ls -F thesis
```

Let's change our working directory to `thesis` using `cd`, then run a text editor called Nano to create a file called `draft.txt`:

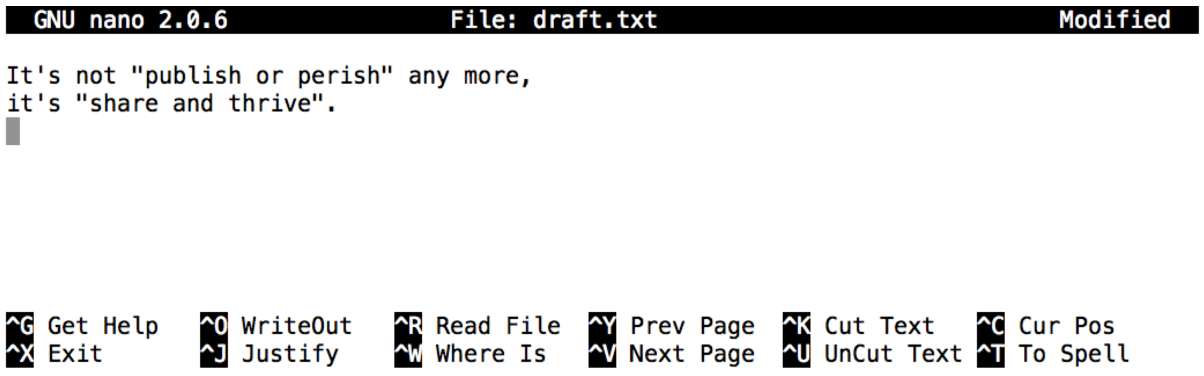
```
$ cd thesis
$ nano draft.txt
```

3.3.1 Which Editor?

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. We use it in examples because almost 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.

No matter what editor you use, you will need to know where it searches for and saves files. If you start it from the shell, it will (probably) use your current working directory as its default location. If you use your computer's start menu, it may want to save files in your desktop or documents directory instead. You can change this by navigating to another directory the first time you "Save As..."

Let's type in a few lines of text. Once we're happy with out text, we can press Ctrl-O (press the Ctrl key and, while holding it down, press the O key) to write our data to disk (we'll be asked what file we want to save this to: press Return to accept the suggested default of `draft.txt`).



```
  GNU nano 2.0.6                File: draft.txt                          Modified

It's not "publish or perish" any more,
it's "share and thrive".
```

Figure 3.4: Nano in action

Once our file is saved, we can use Ctrl-X to quit the editor and return to the shell.

nano doesn't leave any output on the screen after it exits, but `ls` now shows that we have created a file called `draft.txt`:

```
$ ls
draft.txt
\end{verbatim}

Let's tidy up by running \texttt{rm\ draft.txt}:

\begin{lstlisting}[style=promptonly]
$ rm draft.txt
```

This command removes files (`rm` is short for "remove"). If we run `ls` again, its output is empty once more, which tells us that our file is gone:

```
$ ls
```

3.3.2 Deleting Is Forever

The Unix shell doesn't have a trash bin that we can recover deleted files from (though most graphical interfaces to Unix do). Instead, when we delete files, they are unhooked from the file system so that their storage space on disk can be recycled. Tools for finding and recovering deleted files do exist, but there's no guarantee they'll work in any particular situation, since the computer may recycle the file's disk space right away.

Let's re-create that file and then move up one directory to `/home/nelle` using `cd ..`:

```
$ pwd
/home/nelle/thesis
```

```
$ nano draft.txt
$ ls
draft.txt
```

```
$ cd ..
```

If we try to remove the entire `thesis` directory using `rm thesis`, we get an error message:

```
$ rm thesis
rm: cannot remove `thesis': Is a directory
```

This happens because `rm` only works on files, not directories. The right command is `rmdir`, which is short for "remove directory". It doesn't work yet either, though, because the directory we're trying to remove isn't empty:

```
$ rmdir thesis
```

```
rmdir: failed to remove `thesis': Directory not empty
```

This little safety feature can save you a lot of grief, particularly if you are a bad typist. To really get rid of `thesis` we must first delete the file `draft.txt`:

```
$ rm thesis/draft.txt
```

The directory is now empty, so `rmdir` can delete it:

```
$ rmdir thesis
```

3.3.3 With Great Power Comes Great Responsibility

Removing the files in a directory just so that we can remove the directory quickly becomes tedious. Instead, we can use `rm` with the `-r` flag (which stands for "recursive"):

```
$ rm -r thesis
```

This removes everything in the directory, then the directory itself. If the directory contains sub-directories, `rm -r` does the same thing to them, and so on. It's very handy, but can do a lot of damage if used without care.

Let's create that directory and file one more time. (Note that this time we're running nano with the path `thesis/draft.txt`, rather than going into the `thesis` directory and running nano on `draft.txt` there.)

```
$ pwd
```

```
/home/nelle
```

```
$ mkdir thesis
```

```
$ nano thesis/draft.txt
$ ls thesis
```

```
draft.txt
```

`draft.txt` isn't a particularly informative name, so let's change the file's name using `mv`, which is short for "move":

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

The first parameter tells `mv` what we're "moving", while the second is where it's to go. In this case, we're moving `thesis/draft.txt` to `thesis/quotes.txt`, which has the same effect as renaming the file. Sure enough, `ls` shows us that `thesis` now contains one file called `quotes.txt`:

```
$ ls thesis
quotes.txt
```

One has to be careful when specifying the target file name, since `mv` will silently overwrite any existing file with the same name, which could lead to data loss. An additional flag, `mv -i` (or `mv --interactive`), can be used to make `mv` ask you for confirmation before overwriting.

Just for the sake of inconsistency, `mv` also works on directories — there is no separate `mvdir` command.

Let's move `quotes.txt` into the current working directory. We use `mv` once again, but this time we'll just use the name of a directory as the second parameter to tell `mv` that we want to keep the filename, but put the file somewhere new. (This is why the command is called "move".) In this case, the directory name we use is the special directory name `.` that we mentioned earlier.

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

The effect is to move the file from the directory it was in to the current working directory. `ls` now shows us that `thesis` is empty:

```
$ ls thesis
```

Further, `ls` with a filename or directory name as a parameter only lists that file or directory. We can use this to see that `quotes.txt` is still in our current directory:

```
$ ls quotes.txt
quotes.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, `ls` can be given thousands of paths at once:

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

```
quotes.txt   thesis/quotations.txt
```

To prove that we made a copy, let's delete the `quotes.txt` file in the current directory and then run that same `ls` again.

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

```
ls: cannot access quotes.txt: No such file or directory
thesis/quotations.txt
```

This time it tells us that it can't find `quotes.txt` in the current directory, but it does find the copy in `thesis` that we didn't delete.

3.3.4 What's In A Name?

You may have noticed that all of Nelle's files' names are "something dot something", and in this part of the lesson, we always used the extension `.txt`. 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, `.png` is a PNG image, 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 plain text files, PDF documents, configuration files, 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.

3.3.5 Questions

Renaming files: Suppose that you created a `.txt` 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?

```
$ pwd
/home/jamie/data
$ ls
proteins.dat
$ mkdir recombine
$ mv proteins.dat recombine
$ cp recombine/proteins.dat ../proteins-saved.dat
$ ls
```

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

Organizing Directories and Files: Jamie is working on a project and she sees that her files aren't very well organized:

```
$ ls -F
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?

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

Copy with Multiple Filenames: What does cp do when given several filenames and a directory name, as in:

```
$ mkdir backup
$ cp thesis/citations.txt thesis/quotations.txt backup
```

What does cp do when given three or more filenames, as in:

```
$ ls -F
intro.txt    methods.txt    survey.txt
$ cp intro.txt methods.txt survey.txt
```

Listing Recursively and By Time: The command `ls -R` lists the contents of directories recursively, i.e., lists their sub-directories, sub-sub-directories, and so on in alphabetical order at each level. The command `ls -t` lists things by time of last change, with most recently changed files or directories first. In what order does `ls -R -t` display things?

3.4 Pipes and Filters

In this section we learn how tp (a) redirect a command's output to a file, (b) process a file instead of keyboard input using redirection, (c) construct command pipelines with two or more stages, (d) explain what usually happens if a program or pipeline isn't given any input to process, (e) explain Unix's "small pieces, loosely joined" philosophy.

Now that we know a few basic commands, we can finally look at the shell's most powerful feature: the ease with which it lets us combine existing programs in new ways. We'll start with a directory called `molecules` that contains six files describing some simple organic molecules. The `.pdb` extension indicates that these files are in Protein Data Bank format, a simple text format that specifies the type and position of each atom in the molecule.

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

Let's go into that directory with `cd` and run the command `wc *.pdb`. `wc` is the "word count" command: it counts the number of lines, words, and characters in files. The `*` in `*.pdb` matches zero or more characters, so the shell turns `*.pdb` into a list of all `.pdb` files in the current directory:

```
$ 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
```

3.4.1 Wildcards

`*` 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`. We can use any number of wildcards at a time: for example, `p*.p?*` matches anything that starts with a 'p' and ends with '.', 'p', and at least one more character (since the ? has to match one character, and the final * can match any number of characters). Thus, `p*.p?*` would match `preferred.practice`, and even `p.pi` (since the first * can match no characters at all), but not `quality.practice` (doesn't start with 'p') or `preferred.p` (there isn't at least one character after the '.p').

When the shell sees a wildcard, it expands the wildcard to create a list of matching filenames *before* running the command that was asked for. As an exception, if a wildcard expression does not match any file, Bash will pass the expression as a parameter to the command as it is. For example typing `ls *.pdf` in the `molecules` directory (which contains only files with names ending with `.pdb`) results in an error message that there is no file called `*.pdf`. However, generally commands like `wc` and `ls` see the lists of file names matching these expressions, but not the wildcards themselves. It is the shell, not the other programs, that deals with expanding wildcards, and this is another example of orthogonal design.

3.4.2 Using wildcards

When run in the `molecules` directory, which `ls` command will produce this output?

```
ethane.pdb   methane.pdb
```

1. `ls *t*ane.pdb`
2. `ls *t?ne.*`
3. `ls *t??ne.pdb`
4. `ls ethane.*`

If we run `wc -l` instead of just `wc`, the output shows only the number of lines per file:

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

We can also use `-w` to get only the number of words, or `-c` to get only the number of characters.

Which of these files is shortest? It's an easy question to answer when there are only six files, but what if there were 6000? Our first step toward a solution is to run the command:

```
$ wc -l *.pdb \textgreater lengths.txt
```

The greater than symbol, >, tells the shell to **redirect** the command's output to a file instead of printing it to the screen. (This is why there is no screen output: everything that `wc` would have printed has gone into the file `lengths.txt` instead.) The shell will create the file if it doesn't exist. If the file exists, it will be silently overwritten, which may lead to data loss and thus requires some caution. `ls lengths.txt` confirms that the file exists:

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

We can now send the content of `lengths.txt` to the screen using `cat lengths.txt`. `cat` stands for "concatenate": it prints the contents of files one after another. There's only one file in this case, so `cat` just shows us what it contains:

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

3.4.3 Output page by page

We'll continue to use `cat` in this lesson, for convenience and consistency, but it has the disadvantage that it always dumps the whole file onto your screen. More useful in practice is the command `less`, which you use with `$ less lengths.txt`. This displays a screenful of the file, and then stops. You can go forward one screenful by pressing the spacebar, or back one by pressing `b`. Press `q` to quit.

Now let's use the `sort` command to sort its contents. We will also use the `-n` flag to specify that the sort is numerical instead of alphabetical. This does *not* change the file; instead, it sends the sorted result to the screen:

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

We can put the sorted list of lines in another temporary file called `sorted-lengths.txt` by putting `> sorted-lengths.txt` after the command, just as we used `> lengths.txt` to put the output of `wc` into `lengths.txt`. Once we've done that, we can run another command called `head` to get the first few lines in `sorted-lengths.txt`:

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

Using the parameter `-n 1` with `head` tells it that we only want the first line of the file; `-n 20` would get the first 20, and so on. Since `sorted-lengths.txt` contains the lengths of our files ordered from least to greatest, the output of `head` must be the file with the fewest lines.

If you think this is confusing, you're in good company: even once you understand what `wc`, `sort`, and `head` do, all those intermediate files make it hard to follow what's going on. We can make it easier to understand by running `sort` and `head` together:

```
$ sort -n lengths.txt | head -n 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.

Nothing prevents us from chaining pipes consecutively. That is, we can for example send the output of `wc` directly to `sort`, and then the resulting output to `head`. Thus we first use a pipe to send the output of `wc` to `sort`:

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

```
   9 methane.pdb
  12 ethane.pdb
  15 propane.pdb
  20 cubane.pdb
  21 pentane.pdb
  30 octane.pdb
 107 total
```

And now we send the output ot this pipe, through another pipe, to `head`, so that the full pipeline becomes:

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

This is exactly like a mathematician nesting functions like *log(3x)* and saying "the log of three times *x*". In our case, the calculation is "head of sort of line count of `*.pdb`".

Here's what actually happens behind the scenes when we create a pipe. When a computer runs a program — any program — it creates a **process** in memory to hold the program's software and its current state. Every process has an input channel called **standard input**. (By this point, you may be surprised that the name is so memorable, but don't worry: most Unix programmers call it "stdin". Every process also has a default output channel called **standard output** (or "stdout").

The shell is actually just another program. Under normal circumstances, whatever we type on the keyboard is sent to the shell on its standard input, and whatever it produces on standard output is displayed on our screen. When we tell the shell to run a program, it creates a new process and temporarily sends whatever we type on our keyboard to that process's standard input, and whatever the process sends to standard output to the screen.

Here's what happens when we run `wc -l *.pdb > lengths.txt`. The shell starts by telling the computer to create a new process to run the `wc` program. Since we've provided some filenames as parameters, `wc` reads from them instead of from standard input. And since we've used > to redirect output to a file, the shell connects the process's standard output to that file.

If we run `wc -l *.pdb | sort -n` instead, the shell creates two processes (one for each process in the pipe) so that `wc` and `sort` run simultaneously. The standard output of `wc` is fed directly to the standard input of `sort`; since there's no redirection with `>`, `sort`'s output goes to the screen. And if we run `wc -l *.pdb | sort -n | head -n 1`, we get three processes with data flowing from the files, through wc to sort, and from sort through head to the screen.

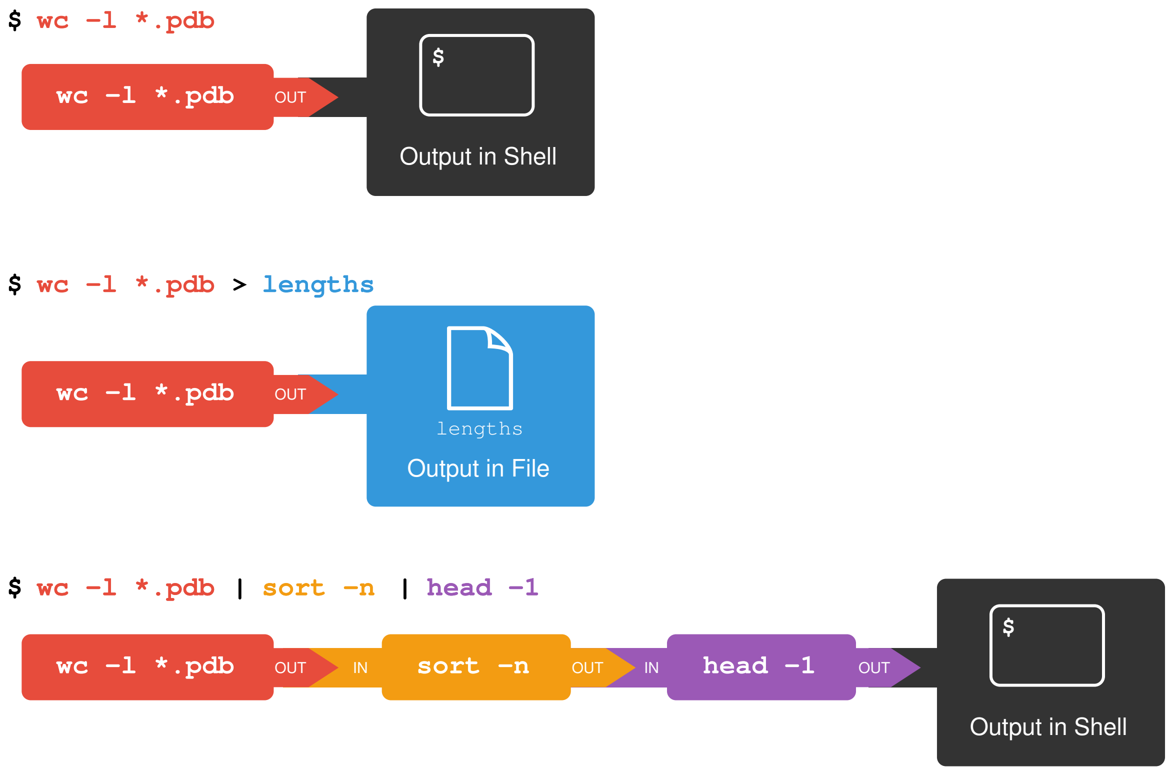


Figure 3.5: Redirects and Pipes

This simple idea is why Unix has been so successful. Instead of creating enormous programs that try to do many different things, Unix programmers focus on creating lots of simple tools that each do one job well, and that work well with each other. This programming model is called "pipes and filters". We've already seen pipes; a **filter** is a program like wc or sort that transforms a stream of input into a stream of output. Almost all of the standard Unix tools can work this way: unless told to do otherwise, they read from standard input, do something with what they've read, and write to standard output.

The key is that any program that reads lines of text from standard input and writes lines of text to standard output can be combined with every other program that behaves this way as well. You can *and should* write your programs this way so that you and other people can put those programs into pipes to multiply their power.

3.4.4 Redirecting Input

As well as using > to redirect a program's output, we can use < to redirect its input, i.e., to read from a file instead of from standard input. For example, instead of writing `wc ammonia.pdb`, we could write `wc < ammonia.pdb`. In the first case, `wc` gets a command line parameter telling it what file to open. In the second, `wc` doesn't have any command line parameters, so it reads from standard input, but we have told the shell to send the contents of `ammonia.pdb` to `wc`'s standard input.

3.4.5 Nelle's Pipeline: Checking Files

Nelle has run her samples through the assay machines and created 1520 files in the `north-pacific-gyre/2012-07-03` directory described earlier. As a quick sanity check, starting from her home directory, Nelle types:

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

The output is 1520 lines that look like this:

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

Now she types this:

```
$ wc -l *.txt | sort -n | head -n 5
```

```
 240 NENE02018B.txt
 300 NENE01729A.txt
 300 NENE01729B.txt
 300 NENE01736A.txt
 300 NENE01751A.txt
```

Whoops: one of the files is 60 lines shorter than the others. When she goes back and checks it, she sees that she did that assay at 8:00 on a Monday morning — someone was probably in using the machine on the weekend, and she forgot to reset it. Before re-running that sample, she checks to see if any files have too much data:

```
$ wc -l *.txt | sort -n | tail -n 5
 300 NENE02040B.txt
 300 NENE02040Z.txt
 300 NENE02043A.txt
 300 NENE02043B.txt
5082 total
```

Those numbers look good — but what's that 'Z' doing there in the third-to-last line? All of her samples should be marked 'A' or 'B'; by convention, her lab uses 'Z' to indicate samples with missing information. To find others like it, she does this:

```
$ ls *Z.txt
NENE01971Z.txt    NENE02040Z.txt
```

Sure enough, when she checks the log on her laptop, there's no depth recorded for either of those samples. Since it's too late to get the information any other way, she must exclude those two files from her analysis. She could just delete them using `rm`, but there are actually some analyses she might do later where depth doesn't matter, so instead, she'll just be careful later on to select files using the wildcard expression `*[AB].txt`. As always, the `*` matches any number of characters; the expression `[AB]` matches either an 'A' or a 'B', so this matches all the valid data files she has.

3.4.6 Questions

What does `sort -n` do? If we run `sort` on this file:

```
10
2
19
22
6
```

the output is:

```
10
19
2
22
6
```

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

```
2
6
10
19
22
```

Explain why `-n` has this effect.

What does < mean? What is the difference between:

```
wc -l < mydata.dat
```

and:

```
wc -l mydata.dat
```

What does >> mean? What is the difference between:

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

and:

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

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

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`

Why does `uniq` only remove adjacent duplicates? The command `uniq` removes adjacent duplicated lines from its input. For example, if a file `salmon.txt` contains:

```
coho
coho
steelhead
coho
steelhead
steelhead
```

then `uniq salmon.txt` produces:

```
coho
steelhead
coho
steelhead
```

Why do you think `uniq` only removes *adjacent* duplicated lines? (Hint: think about very large data sets.) What other command could you combine with it in a pipe to remove all duplicated lines?

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

```
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?

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

Pipe construction: For the file `animals.txt` from the previous exercise, the command:

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

produces the following output:

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

What other command(s) could be added to this in a pipeline to find out what animals the file contains (without any duplicates in their names)?

3.5 Finding Things

In this section we learn to (a) use grep to select lines from text files that match simple patterns, (b) use find to find files whose names match simple patterns, (c) use the output of one command as the command-line parameters to another command, (d) explain what is meant by "text" and "binary" files, and why many common tools don't handle the latter well.

In the same way that many of us now use "Google" as a verb meaning "to find", Unix programmers often use the word "grep". "grep" is a contraction of "global/regular expression/print", a common sequence of operations in early Unix text editors. It is also the name of a very useful command-line program.

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.
```

Let's find lines that contain the word "not":

```
$ grep not haiku.txt
```

```
Is not the true Tao, until
"My Thesis" not found
Today it is not working
```

Here, not is the pattern we're searching for. It's pretty simple: every alphanumeric character matches against itself. After the pattern comes the name or names of the files we're searching in. The output is the three lines in the file that contain the letters "not".

Let's try a different pattern: "day".

```
$ grep day haiku.txt
```

```
Yesterday it worked
Today it is not working
```

This time, two lines that include the letters "day" are outputted. However, these letters are contained within larger words. 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
```

In this case, there aren't any, so `grep`'s output is empty. Sometimes we don't want to search for a single word, but a phrase. This is also easy to do with `grep` by putting the phrase in quotes.

```
$ grep -w "is not" haiku.txt
```

```
Today it is not working
```

We've now seen that you don't have to have quotes around single words, but it is useful to use quotes when searching for multiple words. It also helps to make it easier to distinguish between the search term or phrase and the file being searched. We will use quotes in the remaining examples.

Another useful option is `-n`, which numbers the lines that match:

```
$ grep -n "it" haiku.txt
```

```
5:With searching comes loss
9:Yesterday it worked
10:Today it is not working
```

Here, we can see that lines 5, 9, and 10 contain the letters "it".

We can combine options (i.e. flags) as we do with other Unix commands. For example, let's find the lines that contain the word "the". We can combine the option `-w` to find the lines that contain the word "the" and `-n` to number the lines that match:

```
$ grep -n -w "the" haiku.txt
```

```
2:Is not the true Tao, until
6:and the presence of absence:
```

Now we want to use the option `-i` to make our search case-insensitive:

```
$ 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:
```

Now, we want to use the option `-v` to invert our search, i.e., we want to output the lines that do not contain the word “the”.

```
$ 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.
```

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

```
$ grep --help
```

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 `grep` we can type `man grep`. `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.

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”.

3.5.1 Wildcards

`grep`‘s real power doesn’t come from its options, though; it comes from the fact that patterns can include wildcards. (The technical name for these is **regular expressions**, which is what the “re” in “grep” stands for.) Regular expressions are both complex and powerful; if you want to do complex searches, please look at the lesson on our website. As a taster, we can find lines that have an ’o’ in the second position like this:

```
$ grep -E '^.o' haiku.txt
```

```
You bring fresh toner.
Today it is not working
Software is like that.
```

We use the -E flag and put the pattern in quotes to prevent the shell from trying to interpret it. (If the pattern contained a *, for example, the shell would try to expand it before running grep.) The ^ in the pattern anchors the match to the start of the line. The . matches a single character (just like ? in the shell), while the o matches an actual ‘o’.

While grep finds lines in files, the 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.

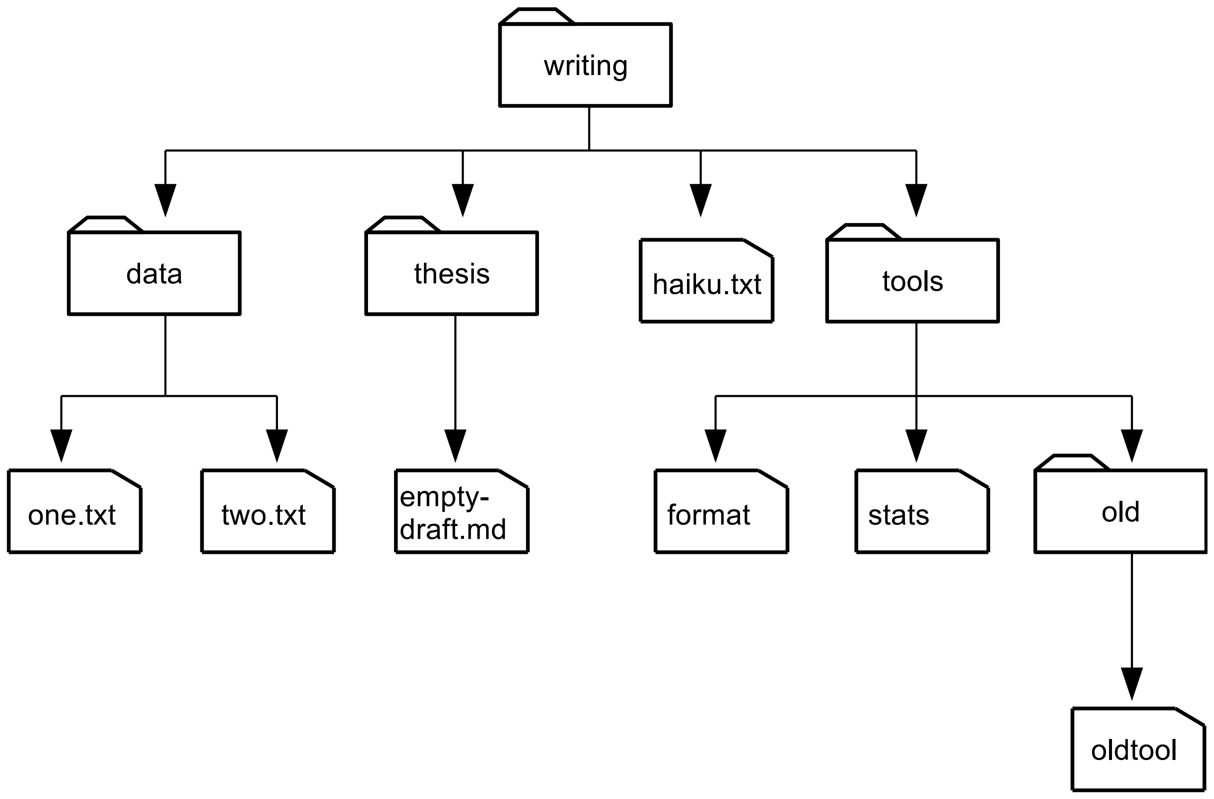


Figure 3.6: File Tree for Find Example

Nelle’s writing directory contains one file called haiku.txt and four subdirectories:

`thesis` (which contains a sadly empty file, `empty-draft.md`), `data` (which contains two files `one.txt` and `two.txt`), a `tools` directory that contains the programs `format` and `stats`, and a subdirectory called `old`, with a file `oldtool`.

For our first command, let's run `find . -type d`. As always, the `.` on its own means the current working directory, which is where we want our search to start; `-type d` means "things that are directories". Sure enough, `find`'s output is the names of the five directories in our little tree (including `.`):

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

```
./
./data
./thesis
./tools
./tools/old
```

If we change `-type d` to `-type f`, we get a listing of all the files instead:

```
$ find . -type f
```

```
./haiku.txt
./tools/stats
./tools/old/oldtool
./tools/format
./thesis/empty-draft.md
./data/one.txt
./data/two.txt
```

`find` automatically goes into subdirectories, their subdirectories, and so on to find everything that matches the pattern we've given it. If we don't want it to, we can use `-maxdepth` to restrict the depth of search:

```
$ find . -maxdepth 1 -type f
```

```
./haiku.txt
```

The opposite of `-maxdepth` is `-mindepth`, which tells `find` to only report things that are at or below a certain depth. `-mindepth 2` therefore finds all the files that are two or more levels below us:

```
$ find . -mindepth 2 -type f
```

```
./data/one.txt
./data/two.txt
```

```
./tools/format
./tools/stats
```

Now let's try matching by name:

```
$ find . -name *.txt
./haiku.txt
\end{verbatim}

We expected it to find all the text files, but it only prints out
\texttt{./haiku.txt}. The problem is that the shell expands
   wildcard
characters like \texttt{*} \emph{before} commands run. Since
\texttt{*.txt} in the current directory expands to
   \texttt{haiku.txt},
the command we actually ran was:

\begin{lstlisting}[style=promptonly]
$ find . -name haiku.txt
```

`find` did what we asked; we just asked for the wrong thing.

To get what we want, let's do what we did with `grep`: put `*.txt` in single quotes to prevent the shell from expanding the `*` wildcard. This way, `find` actually gets the pattern `*.txt`, not the expanded filename `haiku.txt`:

```
$ find . -name '*.txt'
./data/one.txt
./data/two.txt
./haiku.txt
```

3.5.2 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.

As we said earlier, the command line's power lies in combining tools. We've seen how to do that with pipes; let's look at another technique. As we just saw, `find . -name '*.txt'` gives us a list of all text files in or below the current directory. How can we combine that with `wc -l` to count the lines in all those files?

The simplest way is to put the `find` command inside `$()`:

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

When the shell executes this command, the first thing it does is run whatever is inside the `$()`. It then replaces the `$()` expression with that command's output. Since the output of `find` is the three filenames `./data/one.txt`, `./data/two.txt`, and `./haiku.txt`, the shell constructs the command:

```
$ wc -l ./data/one.txt ./data/two.txt ./haiku.txt
```

which is what we wanted. This expansion is exactly what the shell does when it expands wildcards like `*` and `?`, but lets us use any command we want as our own "wildcard".

It's very common to use `find` and `grep` together. The first finds files that match a pattern; the second looks for lines inside those files that match another pattern. Here, for example, we can find PDB files that contain iron atoms by looking for the string "FE" in all the `.pdb` files above the current directory:

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

3.5.3 Binary Files

We have focused exclusively on finding things in text files. What if your data is stored as images, in databases, or in some other format? One option would be to extend tools like `grep` to handle those formats. This hasn't happened, and probably won't, because there are too many formats to support.

The second option is to convert the data to text, or extract the text-ish bits from the data. This is probably the most common approach, since it only requires people to build one tool per data format (to extract information). On the one hand, it makes simple things easy to do. On the negative side, complex things are usually impossible. For example, it's easy enough to write a program that will extract X and Y dimensions from image files for `grep` to play with, but how would you write something to find values in a spreadsheet whose cells contained formulas?

The third choice is to recognize that the shell and text processing have their limits, and to use a programming language such as Python instead. When the time comes to do this, don't be too hard on the shell: many modern programming languages, Python included, have borrowed a lot of ideas from it, and imitation is also the sincerest form of praise.

The Unix shell is older than most of the people who use it. It has survived so long because it is one of the most productive programming environments ever created — maybe even *the* most productive. Its syntax may be cryptic, but people who have mastered it can experiment with different commands interactively, then use what they have learned to automate their work. Graphical user interfaces may be better at the first, but the shell is still unbeaten at the second. And as Alfred North Whitehead wrote in 1911, "Civilization advances by extending the number of important operations which we can perform without thinking about them."

3.5.4 Questions

Using grep

```
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.
```

From the above text, contained in the file `haiku.txt`, which command would result in the following output:

```
and the presence of absence:
```

1. `grep "of" haiku.txt`
2. `grep -E "of" haiku.txt`

3. `grep -w "of" haiku.txt`
4. `grep -i "of" haiku.txt`

`find` pipeline reading comprehension Write a short explanatory comment for the following shell script:

```
wc -l $(find . -name '*.dat') | sort -n
```

Matching `ose.dat` but not `temp` The `-v` flag to `grep` inverts pattern matching, so that only lines which do *not* match the pattern are printed. Given that, which of the following commands will find all files in `/data` whose names end in `ose.dat` (e.g., `sucrose.dat` or `maltose.dat`), but do *not* contain the word `temp`?

1. `find /data -name '*.dat' | grep ose | grep -v temp`
2. `find /data -name ose.dat | grep -v temp`
3. `grep -v "temp" $(find /data -name '*ose.dat')`
4. None of the above.

Part II

C Programming