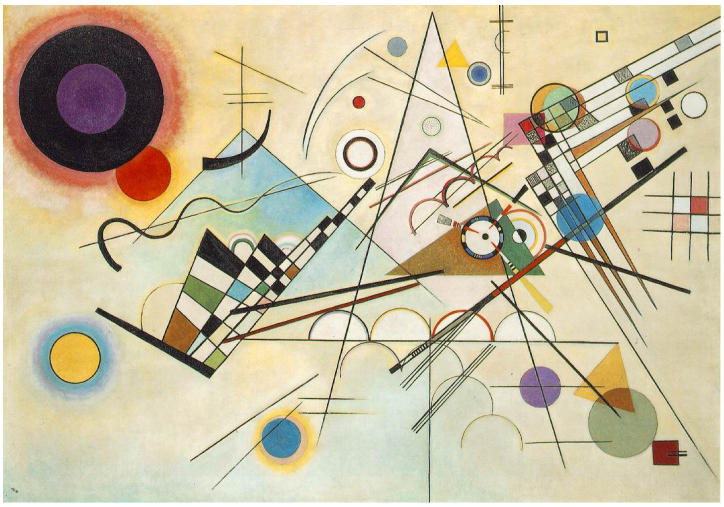

Applied Data Science

Ian Langmore **Daniel Krasner**

Contents

What is data science? With the major technological advances of the last two decades, coupled in part with the internet explosion, a new breed of analysist has emerged. The exact role, background, and skill-set, of a *data scientist* are still in the process of being defined and it is likely that by the time you read this some of what we say will seem archaic.

In very general terms, we view a data scientist as an individual who uses current computational techniques to analyze data. Now you might make the observation that there is nothing particularly novel in this, and subsequenty ask what has forced the definition.[1] After all statisticians, physicists, biologisitcs, finance quants, etc have been looking at data since their respective fields emerged. One short answer comes from the fact that the data sphere has changed and, hence, a new set of skills is required to navigate it effectively. The exponential increase in computational power has provided new means to investigate the ever growing amount of data being collected every second of the day. What this implies is the fact that any modern data analyst will have to make the time investment to learn computational techniques necessary to deal with the volumes and complexity of the data of today. In addition to those of mathemics and statistics, these software skills are domain transfereable and so it makes sense to create a job title that is also transferable. We could also point to the "data hype" created in industry as a culprit for the term *data science* with the *science* creating an aura of validity and facilitating LinkedIn headhunting.

What skills are needed? One neat way we like to visualize the data science skill set is with Drew Conway's Venn Diagram[Con], see figure 1. Math and statistics is what allows us to properly quantify a phenomenon observed in data. For the sake of narrative lets take a complex deterministic situation, such as whether or not someone will make a loan payment, and attempt to answer this question with a limited number of variables and an imperfect understanding of those variables influence on the event we wish to predict. With the exception of your friendly real estate agent we generally acknowldege our lack of soothseer ability and make statements about the probability of this event. These statements take a mathematical form, for example

$$\mathrm{P}[\text{makes-loan-payment}] = e^{\alpha+\beta\cdot\text{creditscore}}.$$

[1] William S. Cleveland decide to coin the term *data science* and write *Data Science: An action plan for expanding the technical areas of the field of statistics* [Cle]. His report outlined six points for a university to follow in developing a data analyst curriculum.

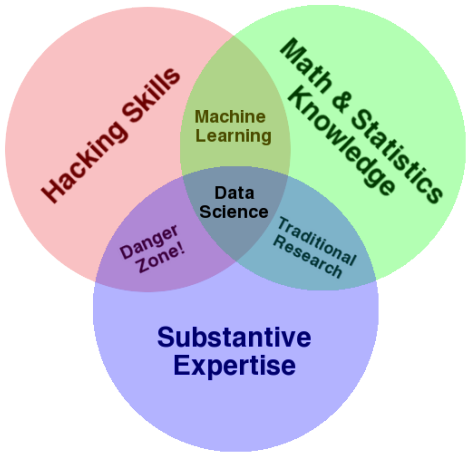

Figure 1: Drew Conway's Venn Diagram

where the above quantifies the *risk* associated with this event. Deciding on the best coefficients α and β can be done quite easily by a host of software packages. In fact anyone with decent hacking skills can do achieve the goal. Of course, a simple model such as this would convince no one and would call for substantive expertise (more commonly called *domain knowledge*) to make real progress. In this case, a domain expert would note that additional variables such as the loan to value ratio and housing price index are needed as they have a huge effect on payment activity. These variables and many others would allow us to arrive at a "better" model

$$\text{P[makes-loan-payment]} = e^{\alpha+\beta\cdot X}. \tag{1}$$

Finally we have arrived at a model capable of fooling someone! We could keep adding variables until the model will almost certainly fit the historic risk quite well. BUT, how do we know that this will allow us to quantify risk in the future? To make some sense of our *uncertainty*[2] about our model we need to know eactly what (1) means. In particular, did we include too many variables and *overfit*? Did our method of solving (1) arrive at a good solution or just numerical noise? Most importantly, how appropriate is the logistic regression model to begin with? Answering these questions is often as much an art as a science, but in our experience, sufficient mathematical understanding is necessary to avoid getting lost.

[2]The distrinction between uncertainty and risk has been talked about quite extensively by Nassim Taleb[Tal05, Tal10]

What is the motivation for, and focus of, this course? Just as common as the hacker with no domain knowledge, or the domain expert with no statistical no-how is the traditional academic with meager computing skills. Academia rewards papers containing original theory. For the most part it does not reward the considerable effort needed to produce high quality, maintainable code that can be used by others and integrated into larger frameworks. As a result, the type of code typically put forward by academics is completely unuseable in industry or by anyone else for that matter. It is often not the purpose or worth the effort to write production level code in an academic environment. The importance of this cannot be overstated. Consider a 20 person start-up that wishes to build a smart-phone app that recommends restaurants to users. The data scientist hired for this job will need to interact with the company database (they will likely not be handed a neat csv file), deal with falsely entered or inconveniently formatted data, and produce legible reports, as well as a working model for the rest of the company to integrate into its production framework. The scientist may be expected to do this work without much in the way of software support. Now, considering how easy it is to blindly run most predictive software, our hypothetical company will be tempted to use a programmer with no statistical knowledge to do this task. Of course, the programmer will fall into analytic traps such as the ones mentioned above but that might not deter anyone from being content with output. This anecdote seems construed, but in reality it is something we have seen time and time again. The current world of data analysis calls for a myriad of skills, and clean programming, database interaction and understand of architecture have all become the minimum to succeed.

The purpose of this course is to take people with strong mathematical/statistical knowledge and teach them software development fundamentals[3]. This course will cover

- Design of small software packages
- Working in a Unix environment
- Designing software in teams
- Fundamental statistical algorithms such as linear and logistic regression

[3]Our view of what constitutes the necessary fundamentals is strongly influenced by the team at software carpentry[Wila]

- Overfitting and how to avoid it
- Working with text data (e.g. regular expressions)
- Time series
- And more...

Part I

Programming Prerequisites

Chapter 1

Unix

Simplicity is the key to brilliance
-Bruce Lee

1.1 History and Culture

The Unix operating system was developed in 1969 at AT&T's Bell Labs. Today Unix lives on through its open source offspring, Linux. This Operating system the dominant force in scientific computing, super computing, and web servers. In addition, mac OSX (which is unix based) and a variety of user friendly Linux operating systems represent a significant portion of the personal computer market. To understand the reasons for this success, some history is needed.

In the 1960s, MIT, AT&T Bell Labs, and General Electric developed a time-sharing (meaning different users could share one system) operating system called Multics. Multics was found to be too complicated. This "failure" led researchers to develop a new operating system that focused on simplicity. This operating system emphasized ease of communication among many simple programs. Kernighan and Pike summarized this as "the idea that the power of a system comes more from the relationships among programs than from the programs themselves."

The Unix community was integrated with the Internet and networked com-

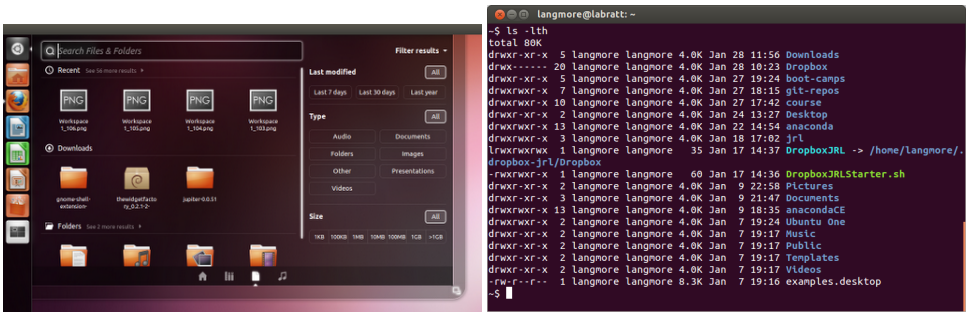

Figure 1.1: Ubuntu's GUI and CLI

puting from the beginning. This, along with the solid fundamental design, could have led to Unix becoming the dominant computing paradigm during the 1980's personal computer revolution. Unfortunately, infighting and poor business decisions kept Unix out of the mainstream.

Unix found a second life, not so much through better business decisions, but through the efforts of Richard Stallman and GNU Project. The goal was to produce a Unix-like operating system that depended only on free software. Free in this case meant, "users are free to run the software, share it, study it, and modify it." The GNU Project succeeded in creating a huge suite of utilities for use with an operating system (e.g. a C compiler) but were lacking the kernel (which handles communication between e.g. hardware and software, or among processes). It just so happened that Linux Torvalds had developed a kernel (the "Linux" kernel) in need of good utilities. Together the Linux operating system was born.

1.2 The Shell

Modern Linux distributions, such as Ubuntu, come with a graphical user interface (GUI) every bit as slick as Windows or Mac OSX. Software is easy to install and with at most a tiny bit of work all non-proprietary applications work fine. The real power of Unix is realized when you start using the *shell*.

Digression 1: Linux without tears

The easiest way to have access to the bash shell and a modern scientific computing environment is to buy hardware that is pre-loaded with Linux. This way, the hardware vendor is takes responsibility for maintaining the proper drivers. Use caution when reading blogs talking about how "easy" it was to get some off-brand laptop computer working with Linux... this could work for you, or you could be left with a giant headache. Currently there are a number of hardware vendors that ship machines with Linux: System76, ZaReason, and Dell (with their "Project Sputnik" campaign). Mac OSX is built on Unix, and also qualifies as a linux machine of sorts. The disadvantage (of a mac) is price, and the fact that the package management system (for installing software) that comes with Ubuntu linux is the cleanest, easiest ever!

The shell allows you to control your computer using commands entered in a keyboard. This sort of interaction is called a *command line interface* (CLI). "The shell" in our case will refer to the *Bourne again* or *bash shell*. The bash shell provides an interface to your computer's OS along with a number of utilties and minilanguages. We will introduce you to the shell during the software carpentry bootcamp. For those unable to attend, we refer you to

Why learn the shell?

- The shell provides a number of utilities that allow you to perform tasks such as interact with your OS or modify a text file.
- The shell provides a number *minilanguages* that allow you to automate these tasks.
- Often programs must communicate with a user or another machine. A CLI is a very simple way to do this. Trust me, you don't want to create a GUI for every script you write.
- Usually the only way to communicate with a remote computer/cluster is using a shell.

Because of this, programs and workflows that *only* work in the shell are common. For this reason alone, a modern scientist must learn to use the shell.

Shell utilities have a common format that is almost always adhered to. This format is: `utilityname options arguments`. The *utilityname* is the name of the utility, such as `cut`, which picks out a column of a csv file. The *options* modify the behavior of the program. In the case of `cut` this could mean

specifying how the file is delimited (tabs, spaces, commas, etc. . .) and which column to pick out. In general, options should in fact be *optional* in that the utility will work without them (but may not give the desired behavior). The *arguments* come last. These are not optional and can often be thought of as the external input to the program. In the case of `cut` this is the file from which to extract a column. Putting this together, if `data.csv` looks like:

```
name,age,weight
ian,1,11
chang,2,22
```

Then

$$\underbrace{\texttt{cut}}_{\text{utilityname}} \quad \underbrace{\texttt{-d, -f1}}_{\text{options}} \quad \underbrace{\texttt{data.csv}}_{\text{arguments}} \tag{1.1}$$

produces (more specifically, prints on the terminal screen)

```
age
1
2
```

1.3 Streams

A *stream* is general term for a sequence of data elements made available over time. This data is processed one element at a time. For example, consider the data file (which we will call `data.csv`):

```
name,age,weight
ian,1,11
chang,2,22
daniel,3,33
```

This data may exist in one contiguous block in memory/disk or not. In either case, to process this data as a stream, you should view it as a contiguous block that looks like

```
name,age,weight\n ian,1,11\n chang,2,22\n daniel,3,33
```

The special character \n is called a *newline* character and represents the start of a new line. The command `cut -d, -f2 data.csv` will pick out the second column of `data.csv`, in other words, it returns

```
age
1
2
3
```

, or, thought of as a stream,

```
age\n 1\n 2\n 3
```

This could be accomplished by reading the file in sequence, starting to store the characters in a buffer once the first comma is hit, then printing when the second comma is hit. Since the newline is such a special character, many languages provide some means for the user to process each line as a separate item.

This is a very simple way to think about data processing. This simplicity is advantageous and allows one to scale stream processing to massive scales. Indeed, the popular Hadoop MapReduce implementation requires that all small tasks operate on streams. Another advantage of stream processing is that memory needs are reduced. Programs that are able to read from stdin and write to stdout are known as *filters*.

1.3.1 Standard streams

While stream is a general term, there are three streaming input and output channels available on (almost) every machine. These are *standard input* (stdin), *standard output* (stdout), and *standard error* (stderr). Together, these *standard streams* provide a means for a *process* to communicate with other processes, or a computer to communicate with other machines (see figure 1.3.1). Standard input is used to allow a process to read data from another source. A Python programmer could read from standard in, then print the same thing to standard out using

```
for line in sys.stdin:
    sys.stdout.write(line)
```

If data is flowing into stdin, then this will result in the same data being written to stdout. If you launch a terminal, then stdout is (by default) connected to your terminal display. So if a program sends something to stdout it is displayed on your terminal. By default stdin is connected to your keyboard. Stderr operates sort of like stdout but all information carries the

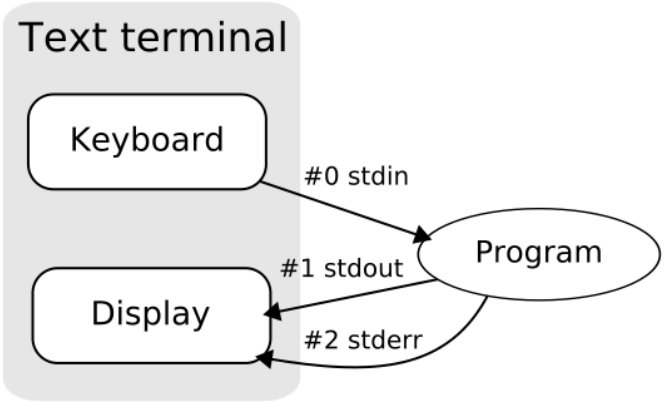

Figure 1.2: Illustration of the standard streams

special tag, "this is an error message." Stderr is therefore used for printing error/debugging information.

1.3.2 Pipes

The standard streams aren't any good if there isn't any way to access them. Unix provides a very simple means to connect the standard output of one process to the standard input of another. This construct called a *pipe* and is written with a vertical bar `|`. Utilities tied together with pipes form what is known as a *pipeline*.

Consider the following pipeline

```
\$ cat infile.csv | cut -d, -f1 | sort | uniq -c
```

The above line reads in a text file and prints it to standard out with `cat`, the pipe "|" redirects this standard out to the standard in of `cut`. `cut` in turn extracts the first column and passes the result to `sort`, which sends its result to `uniq`. `uniq -c` counts the number of unique occurrences of each word.

Let's decompose this step-by-step: First, print *infile.csv* to stdout (which is, by default, the terminal) using cat.

```
\$ cat infile.csv
```

```
ian,1
daniel,2
chang,3
ian,11
```

Second, pipe this to `cut`, which will extract the first *field* (the `-f` option) in this comma delimited (the `-d,` option) file.

```
\$ cat infile.csv | cut -d, -f1

ian
daniel
chang
ian
```

Third, pipe the output of cut to sort

```
\$ cat infile.csv | cut -d, -f1 | sort

chang
daniel
ian
ian
```

Third, redirect the output of sort to uniq.

```
\$ cat infile.csv | cut -d, -f1 | sort | uniq -c

1  chang
1  daniel
2  ian
```

It is important to note that uniq counts unique occurrences in consecutive lines of text. If we did not sort the input to uniq, we would have

```
\$ cat infile.csv | cut -d, -f1 | uniq -c

1  ian
1  daniel
1  chang
1  ian
```

`uniq` processes text streams character-by-character and does not have the ability to look ahead and see that "ian" will occur a second time.

1.4 Text

One surprising thing to some Unix newcomers is the degree to which simple plain text dominates. The preferred file format for most data files and streams is just plain text.

Why not use a compressed binary format that would be quicker to read/write using a special reader application? The reason is in the question: A special reader application would be needed. As time goes on, many data formats and reader applications come in, and then out of favor. Soon your special format data file needs a hard to find application to read it[1]. What about for communication between processes on a machine? The same situation arises: As soon as more than one binary format is used, it is possible for one of them to become obsolete. Even if both are well supported, every process needs to specify what format it is using. Another advantage of working with text streams is the fact that humans can visually inspect them for debugging purposes.

While binary formats live and die on a quick (computer) time-scale, change in human languages changes on the scale of at least a generation. In fact, one summary of the Unix philosophy goes, "This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface."

This, in addition to the fact that programming in general requires manipulation of text files, means that you are required to master decent text processing software. Here is a brief overview of some popular programs

- **Vim** is a powerful text editor designed to allow quick editing of files and minimal hand movement.
- **Emacs** is another powerful text editor. Some people find that it requires users to contort their hands and leads to wrist problems.
- **Gedit, sublime text** are decent text editors available for Linux and Mac. They are not as powerful as Vim/Emacs, but don't require any special skills to use.
- **nano** is a simple unix text editor available on any system. If nano

[1] Any user of Microsoft Word documents from the 90's should be familiar with the headaches that can arise from this situation.

doesn't work, try *pico*.

- **sed** is a text stream processing command line utility available in your shell. It can do simple operations on one line of text at a time. It is useful because of its speed, and the fact that it can handle arbitrarily large files.
- **awk** is an old school minilanguage that allows more complex operations than sed. It is often acknowledged that awk syntax is too complex and that learning to write simple Python scripts is a better game plan.

1.5 Philosophy

The Unix culture carries with it a philosophy about software design. The Unix operating system (and its core utilities) can be seen as examples of this. Let's go over some key rules. With the exception of the rule of collaboration, these appeared previously in [Ray04].

1.5.1 In a nutshell

Rule of Simplicity. Design for simplicity. Add complexity only when you must.

Rule of Collaboration. Make programs that work together. Work together with people to make programs

1.5.2 More nuts and bolts

We can add more rules to the two main rules above, and provide hints as to how they will guide our software development. Our programs will be small, so (hopefully) few compromises will have to be made.

Rule of Simplicity. This is sometimes expressed as K.I.S.S, or "Keep It Simple Stupid." All other philosophical points presented here can be seen as special cases of this. Complex programs are difficult to debug, implement, maintain, or extend. We will keep things simple by, for example: (i) writing CLI utilities that *do one thing well*, (ii) avoiding objects unless using

them results in a simpler, more transparent design, and (iii) in our modules, include only features that will be used right now.

Rule of Collaboration. We will make programs that work together by, for example: (i) writing CLI utilities that work as filters, and (ii) choosing common data structures (such as Numpy arrays, Pandas DataFrames). We will work together with people to make programs by, for example: (i) employing Git as a version control system (using Github to host our code) and, (ii) enforcing code readability standards such as PEP8.

Rule of Modularity. Write simple parts connected by clean interfaces. Humans can hold only a limited amount of information in their head at one time. Make your functions small (simple) enough so that they can be explained in one sentence.

Rule of Clarity. Clarity is better than cleverness. Maintenance and debugging of code is very expensive. Take time to make sure your program logic will be clear to someone reading your code some time in the future (this person might be you). Comments are important. Better yet, code can often be written to read like a story... and no comments are necessary.

```
for row in reader:
    rowsum = sum_row(row)
    row.append(rowsum)
    writer.write(row)
```

Rule of Composition. Design programs to be connected to other programs. The Unix command line utilities are an example of this. They (typically) can read from a file or stdin, and write to stdout. Thus, multiple utilities can be tied together with pipes.

```
cat infile.csv | cut -f1 | sort | uniq -c
```

Rule of Least Surprise. Try to do the least surprising thing. We will follow Unix or Python convention whenever possible. For example, our data files will be in common formats such as csv, xml, json, etc...

1.6 End Notes

Revolution OS is a fun movie about the rise of Linux.

[Ray04] gives a comprehensive exposition of the history and philosophy of

Unix, and provides most of the material you see in our history and philosophy sections.

The quote by Kernighan and Pike can be found in "The Unix programming environment."[KP84]

Software Carpentry held a bootcamp for students in three courses at Columbia University in 2013 [Wilb].

The impact of the inventions to come out of Bell Labs cannot be understated. Also developed there were radio astronomy, the transistor, the laser, the CCD, information theory, and the C/C++ programming languages.[Wik]

Chapter 2

Version Control with Git

Git! That's the vcs that I have to look at Google to use.
- Josef Perktold

2.1 Background

The idea of version control is almost as old as writing itself. Authors writing books and manuscripts all needed logical ways to keep track of the various edits they made throughout the writing process. Version control systems like Git, SVN, Mercurial, or CVS allow you to save different versions of your files, and revert to those earlier versions when necessary. The most modern of these four systems are Git and Mercurial. Each of these have many features designed to facilitate working in large groups and keeping track of many versions of files.

2.2 What is Git

Git is a distributed version control system (DVCS). This means that every user has a complete copy of the repository on their machine. This is nice, since you don't need an internet connection to check out different versions of your code, or save a new version. Multiple users still do need some way to share files. In this class we will use Git along with the website GitHub.

GitHub provides you with a clone of your local repository that is accessible via the internet. Thus, when you change a file you will *push* those changes to GitHub, and then your teammates will *pull* those changes down to their local machines.

2.3 Setting Up

For macs, download from `mac.github.com`. For Linux, type `sudo apt-get install git`. After installation, get an account at `www.github.com`. Then, in your home directory create a file (or edit if it already exists) called `.gitconfig`. It should have the lines:

```
[user]
    name = Ian Langmore
    email = ianlangmore@gmail.com
[credential]
    helper = cache --timeout=3600
[alias]
        lol = log --graph --decorate --pretty=oneline --abbrev-commit
        lola = log --graph --decorate --pretty=oneline --abbrev-commit --all
[color]
        branch = auto
        diff = auto
        interactive = auto
        status = auto
```

Now, when you're in a repository, you can see the project structure by typing `git lola`.

2.4 Online Materials

Lots of materials are available online. Here we list a few. Be advised that these tutorials are usually written for experienced developers who are migrating from other systems to Git. For example, in this class you will not have to use branches.

- `http://git-scm.com/book` has complete documentation with examples. I recommend reading section 1 before proceeding.

- `http://osteele.com/posts/2008/05/commit-policies` is a visualization of how to transport data over the multiple layers of Git.
- `http://marklodato.github.com/visual-git-guide/index-en.html` provides a more complete visual reference.
- `http://learn.github.com` has a number of video tutorials
- `http://www.kernel.org/pub/software/scm/git/docs/user-manual.html` is a reference for commands

2.5 Basic Git Concepts

One difficulty that beginners have with Git is understanding that as you are working on a file, there are at least four different versions of it.

1. The *working-copy* that is saved in your computer's file system.
2. The saved version in Git's *index* (a.k.a. *staging area*). This is where Git keeps the files that will be committed.
3. The commit current at your *HEAD*. Once you commit a file, it (and the other files committed with it) are saved forever. The commit can be identified by a SHA1 hashtag. The last commit in the current checked out branch is called HEAD.
4. The commit in your *remote repository*. Once you have pulled down changes from the remote repository, and pushed your changes to it, your repository is identical to the remote.

2.6 Common Git Workflows

Here we describe common workflows and the steps needed to execute them. You can test out the steps here (except the remote steps) by creating a temporary local repository:

```
cd /tmp
mkdir repo
cd repo
git init
```

After that you will probably want to quickly create a file and add it to the repo. Do this with

```
echo 'line1' > file
git add file
```

Practice this (and subsequent subsections) on your own. Remember to type `git lola`, `git status`, and `git log` frequently to see what is happening.

You can then add other lines with e.g. `echo 'line2' >> file`. When you are done, you can clean up with `rm -rf repo`.

To set up a remote repository on GitHub, follow the directions at: `https://help.github.com/articles/creating-a-new-repository`.

2.6.1 Linear Move from Working to Remote

To turn in homework, you have to move files from 1 to 4. The basic 1-to-4 workflow would be (note that I use `<something>` when you must fill in some obvious replacement for the word "something." If the "something" is optional I write it in `[square brackets]`).

- **working-copy → index** `git add <file>`. To see the files that differ in index and commit use `git status`. To see the differences between working and index files, use `git diff [<file>]`.
- **index → HEAD** `git commit -m "<message>"`. To see the files that differ in index and commit use `git status`. To see the differences between your working-copy and the commit, use `git diff HEAD [<file>]`.
- **HEAD → remote-repo** `git push [origin master]`. This means "push the commits in your master branch to the remote repo named origin." Note that [origin master] is the default, so it isn't necessary. This actually pushes all commits to origin, but in particular it pushes HEAD.

You can add and commit at once with `git commit -am '<message>'`. This will add files that have been previously added. It will not add untracked files (you have to manually add them with `git add <file>`).

2.6.2 Discarding changes in your working copy

You can replace your working copy with the copy in your index using

```
git checkout <file>
```

You can replace your working copy with the copy in HEAD using

```
git checkout HEAD <file>
```

2.6.3 Erasing changes

If you committed something you shouldn't have, and want to completely wipe out the commit: `git reset --hard HEAD^`. This moves your commit back in history and wipes out the most recent commit.

To move the index and HEAD back one commit, use `git reset HEAD^`.

To move the index to a certain commit (designated by a SHA1 hash), use `git reset <hash>`.

If you then want to move changes into your working copy, use `git checkout <filename>`.

To move contents of a particular commit into your working directory, use `git checkout <hash> [<filename>]`.

2.6.4 Remotes

To copy a remote repository, use one of the following

```
git clone <remote url>
git clone <remote url> -b <branchname>
git clone <remote url> -b <branchname> <destination>
```

To get changes from a remote repository and put them into your repo/index/working, use `git pull`. You will get an error if you have uncommitted changes in your index or working, so first save your changes, then `git add <filename>`, then `git commit -m '<message>'`.

To send changes to a remote repository, use

```
git add <file>
git commit '<message>'
git push
```

2.6.5 Merge conflicts

A typical situation is as follows:

1. Your teammate modifies `<file>`
2. Your teammate pushes changes
3. You modify `<file>`
4. You pull with `git pull`

Git will recognize that you have two versions of the same file that are in "conflict." Git will tell you which files are in conflict. You can open these files and see something like the following:

```
<<<<<<< HEAD:filename
<My work>
=======
<My teammate's work>
>>>>>>> iss53:filename
```

The lines above the `=======` are the version in the commit you most recently made. The lines below are those in your teammate's version. You can do a few things:

- Edit the file, by hand, to get in in the state you want it in.
- Keep your version with `git checkout --ours <filename>`
- Keep their version with `git checkout --theirs <filename>`

Chapter 3

Building a Data Cleaning Pipeline with Python

A quotation

One of the most useful things you can do with Python is to (quickly) build CLI utilities that look and feel like standard Unix tools. These utilities can be tied together, using pipes and a shell script, into a *pipeline*. These can be used for many purposes. We will concentrate on the task of data cleaning or data preparation.

3.1 Simple Shell Scripts

A pipeline that sorts and cleans data could be put into a shell script that looks like:

```
#!/bin/bash

# Here is a comment

SRC=../src
DATA=../data

cat $DATA/inputfile.csv \
```

```
| python $SRC/subsample.py -r 0.1 \
| python $SRC/cleandata.py \
> $DATA/outputfile.csv
```

Some points:

- The # `!/bin/bash` is called a *she-bang* and in this case tells your machine to run this script using the command `/bin/bash`. In other words, let bash run this script.
- All other lines starting with # are comments.
- The line `SRC=../src` sets a variable, `SRC`, to the string `../src`. In this case we are referring to a directory containing our source code. To access the value stored in this variable, we use $ `SRC`.
- The lines that end with a backslash \, are in fact interpreted as one long line with no newlines. This is done to improve readability.
- The first couple lines under `cat` start with pipes, and the last line is a redirection.
- The command `cat` is used on the first line and the output is piped to the first program. This is done rather than simply using (as the first line) `python $SRC/subsample.py -r 0.1 $DATA/inputfile.csv`. What advantage does this give? It allows one to easily substitute `head` for `cat` and have a program that reads only the first 10 lines. This is useful for debugging.

Why write shell scripts to run you programs?

- Shell scripts allow you to tie together any program that reads from stdin and writes to stdout. This includes all the existing Unix utilities.
- You can (and should) add the shell scripts to your repository. This keeps a record of how data was generated.
- Anyone who understands Unix will be able to understand how your data was generated.
- If your script pipes together five programs, then all five can run at once. This is a simple way to parallelize things.
- More complex scripts can be written that can automate this process

3.2 Template for a Python CLI Utility

Python can be written to work as a filter. To demonstrate, we write a program that would delete every n^{th} line of a file.

```
from optparse import OptionParser
import sys


def main():
    r"""
    DESCRIPTION
    -----------
    Deletes every nth line of a file or stdin, starting with the
    first line, print to stdout.

    EXAMPLES
    --------
    Delete every second line of a file
    python deleter.py -n 2 infile.csv
    """
    usage = "usage: %prog [options] dataset"
    usage += '\n'+main.__doc__
    parser = OptionParser(usage=usage)
    parser.add_option(
        "-n", "--deletion_rate",
        help="Delete every nth line [default: %default] ",
        action="store", dest='deletion_rate', type=float, default=2)

    (options, args) = parser.parse_args()

    ### Parse args
    # Raise an exception if the length of args is greater than 1
    assert len(args) <= 1
    infilename = args[0] if args else None

    ## Get the infile
```

```
    if infilename:
        infile = open(infilename, 'r')
    else:
        infile = sys.stdin

    ## Call the function that does the real work
    delete(infile, sys.stdout, options.deletion_rate)

    ## Close the infile iff not stdin
    if infilename:
        infile.close()


def delete(infile, outfile, deletion_rate):
    """
    Write later, if module interface is needed.
    """
    for linenumber, line in enumerate(infile):
        if linenumber % deletion_rate != 0:
            outfile.write(line)



if __name__=='__main__':
    main()
```

Note that:

- The *interface* to the external world is inside `main()` and the *implementation* is put in a separate function `delete()`. This separation is useful because interfaces and implementations tend to change at different times. For example, suppose this code was to be placed inside a larger module that no longer read from stdin?
- The `OptionParser` module provides lots of useful support for other types of options or flags.
- Other, more useful utilities would do functions such as subsampling, cutting certain columns out of the data, reformatting text, or filling missing values. See the homework!