Git on it

For our purposes in MSAN, we're going to ignore most of the nontrivial capabilities that programmers use routinely, such as branching and merging. Git is extremely complicated and would not be my first choice if it weren't for the excellent `github.com`.

Introduction to revision control

Revision control is a mechanism to track all changes to a set of files, which we typically associate with the project. The file status call a *repository* and at any given time, my computer has lots and lots of these repositories.

A `git` repository instance is just a directory on your disk that also happens to have a `.git` (hidden) directory, which is effectively a complete database of everything that's happened to the repository since it was created with `git init` (or you `clone`'d it from somewhere).

If you want to throw out the repository, just remove the entire subtree from your disk. There is no central server to notify. Every repository instance is a complete copy so you could have, for example, 10 versions of the repository cloned from an original sitting on the same disk.

After you create a repository, you can create all sorts of files in the local directories managed by git, but git ignores them until you `add` them. When you add files or modify files already known to git, they are in the so-called staging area (this used to be called the index). You can have whatever other files you want laying around, such as development environment preference files. Git will simply ignore them unless you `add` them. This is different than other revision control systems that insist upon knowing about and managing everything under a particular subtree. I like that feature.

Does a solo programmer need revision control?

If you are working solo, from a single machine, and you have a regular backup mechanism in your development environment or from the operating system like Time Machine (OS X), you can get away without a formal revision system.

There are lots of important operations that can be faked without a revision system. It's a good idea to keep track of versions of the software that work or other milestones. In the old days, people would

make a copy of their project directory corresponding to important milestones like "Added feature X and it seems to work." You can do comparisons using a diff tool in between directories.

Whether your IDE does it or a revision control system does it, I find it very important to look back at recent changes to see what changes have introduced a bug. Or I decide to abandon a small piece of what's going on and flip a file back to an old version.

A good example of use of a repository is the repository for this course:

https://github.com/parrt/msan501

It contains all the changes that I've made since I started teaching this course.

These days, revision control systems are meant to be used among multiple computers and multiple developers, but they are still useful even on a single machine.

Solo programmer, sharing across machines

In order to work on that software from your home machine and a laptop for example, you have to make copies. That introduces the possibility that you will overwrite the good version of your software. Or, you will forget that you had made changes on your laptop but have now made a bunch of changes on your desktop. Resolving things can be tricky and error-prone.

As a side benefit, pushing your repository to a remote server gives you a backup automatically.

Multiple programmers

When you add another person to the project, people end up mailing code around but it's difficult to perform a merge. My experience watching students do this reveals that two versions of the software always appear. Both students shout that their version is better and that the other version should be abandoned.

In my experience, no matter how you try to fake multiple states of the source code and share, merging changes to work on the same code base is a nightmare.

Once in a while I go back and I look at the history of changes. Sometimes I want to know who screwed this up or I want to see the sequence of changes that I made or that were made by somebody else.

Every single commercial developer I know uses revision control at work. Every company you will encounter uses it. For that reason alone, you need to learn revision control to be functional in a commercial setting.

Cloning an existing repository

Fire up SourceTree and use the file menu `New / Clone` option. You can clone that starter kit into any directory you like, but you should probably name it `msan501` or something similar. Note that I'm still in the process of setting up student repositories as I'm writing this prior to receiving a list of github ids. The github URL you will clone will not be similar to tt msan-501-starterkit but instead *YourGihubUserid-msan501*.

You'll then see the status of the repository; click on the `master` branch in the left gutter:
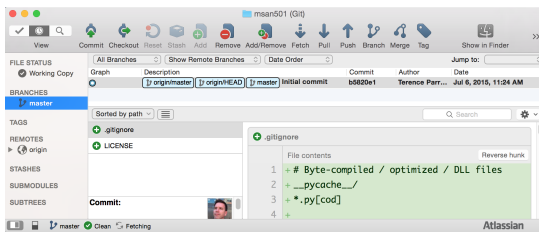
Naturally, this is all much more explicit and quicker from the commandline:

```
$ cd ~
$ git clone git@github.com:parrt/msan501-starterkit.git msan501
```

PyCharm + git = no problem

Most of the time, you can use PyCharm to avoid all of the unpleasantness with the specialized git commands. For example, opening PyCharm on your `msan501` directory after you clone it from github should automatically detect that the directory is a repository. But, you can check this in the preferences. Then, when you create a new file, say, `t.py`, PyCharm will automatically ask if you want to add it to the repository. The two snapshots in the gutter show you what it looks like. You can click the checkbox so that it always just adds files. If you want to manually add them later it's okay. You can do so under the VCS (version control system) menu.

To figure out what changes you have made to the existing repository, click on the Changes tab at the bottom of PyCharm. You will see that t.py is a change (addition) to the repository.

Use menu option VCS > Commit Changes to commit your changes to the local repository. As part of the commit, you can push to the `origin`, which will bring up the Git Push dialog box.
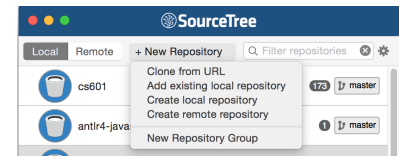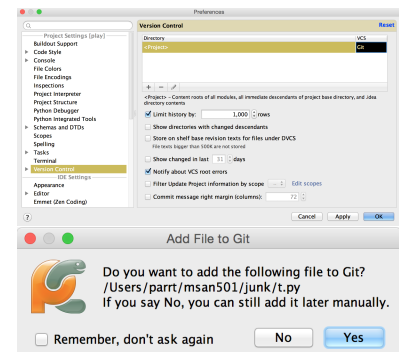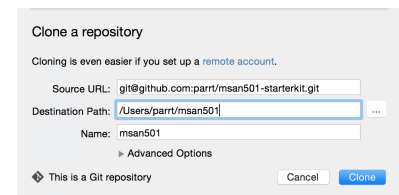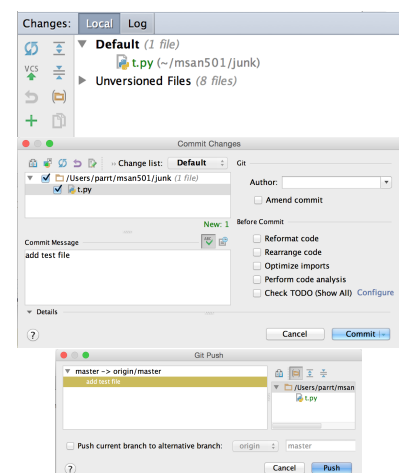

Figure 1: Git config in PyCharm

Figure 3: Commit and push

Git from the command line

Adding to repo. To add files, just create them and do a `commit`:

```
$ cd ~/msan501
$ mkdir junk
$ cd junk
... create t.py ...
$ git add t.py
$ git commit -a -m 'initial add'
```

Make changes. Then you can make changes and do another commit. Make sure use the `-a` command. By the way, deleting a file is also considered a change but you can also use `git rm` *filename*.
Checking differences with repo. If you make a change and want to know how it's different from the current repository version, just use diff:

```
$ ... tweak t.py ...
$ git diff t.py
...
```

Reverting. If you screw up and want to toss out everything from the last commit, do a reset and make sure you use the hard option:

```
$ ... tweak whatever you want ...
$ git reset --hard HEAD
```

which throws out all changes since the last commit. If all you want to do is revert uncommitted changes to a single file, you can run this:

```
$ git checkout -- filename
```

I think they call that funny dash-dash option "sparse mode." See? Git is the assembly code of revision systems. blech.
Correcting commit message. One of the other things I often have to do is to fix the commit message that I just wrote in a commit command.

```
$ git commit --amend -m "I really wanted to say this instead"
```

Adding file you forgot to commit. If you forgot to add one of the files and you wanted in a previous commit, you can also use amend. Just add the file and use amend:

```
$ git add t2.py
$ git commit --amend --no-edit
```

Checking working dir and staging area vs repo. Finally, if you want to figure out what changes you have made such as adding, deleting, or editing files, you can run:

```
$ git status
On branch master
```

```
Your branch is up-to-date with 'origin/master'.

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

    t3.py

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

Don't fear commit*ment*

Every time you *commit* a change (file edit, file add, file delete, ...) to the repository, the revision control system tracks a patch called the *diff* that indicates essentially how to edit the first file version in order to arrive at the current file version. Storing just the differences is very space efficient and lets the revision control system apply the same set of changes to a file on a different computer to keep them both in sync.

Having a complete list of changes is extremely useful. For example, here is a chunk taken out of the middle of my commits on the ANTLR repository as shown by SourceTree:

prepare for changes beyond 4.4	70dd522	Terence Parr...	Jul 16, 2014, 7:45 PM
4.4 add title to javadoc, tweak readme stuff.	90a5aa4	Terence Parr...	Jul 16, 2014, 2:08 PM
add classpath to javadoc	1693c74	Terence Parr...	Jul 16, 2014, 1:45 PM
set to use 4.3 to build 4.4	a61c3ee	Terence Parr...	Jul 16, 2014, 1:34 PM
mkjar was missing runtime in jar and included antlr v2 in complete jar.	54b875c	Terence Parr...	Jul 16, 2014, 1:34 PM
update comment	b807e06	Terence Parr...	Jul 16, 2014, 12:17 PM
include annot in javadoc	677cc27	Terence Parr...	Jul 16, 2014, 12:07 PM
add comment to describe build	7d19889	Terence Parr...	Jul 16, 2014, 10:28 AM
make tests build in their own classpath	1f79785	Terence Parr...	Jul 16, 2014, 10:20 AM

That should look very much like Time Machine on OS X to you. You can go back and look at changes made to the repository for any commit.

You will also notice that I have tagged a particular commit as 4.4 with the tag command. This makes it easy for me to flip the repository back to a specific commit with a name rather than one of those funky commit checksums.

You should commit only logical chunks like feature additions, bug fixes, or comment updates across the project, etc.

Commit messages are important. Do not use meaningless messages, as I see students sometimes do:

```
Add t.py
Alter t.py
Change t.py
...
```

I have even seen git commit messages: one, two, three. Nothing spells laziness or academic dishonesty than that.

In rare cases, when I'm working alone, I sometimes use a private repository as a means of sharing files across multiple computers like

dropbox. In this case, my commits are just to take a snapshot to pull it down on another machine. The commit message doesn't matter (though I might still look back through the changes at some point). When doing this for a real project, it's best to use stash per the next section.

With a remote server like github

When you're working by yourself (and without branches), a remote server acts like a central server that you can push and pull from. For example, I push from my work machine and pull to my home machine or my laptop. And then reverse the process with changes I make at home over the weekend.

For example, once I have commit all of my changes that work and I'm ready to go home, I push to the origin:

```
$ git push origin master
```

From home, I do:

```
$ git pull origin master
```

The origin is the alias for the original server we cloned from and master is our master branch, which we can ignore until we look at branches.

To look at the remote system alias(es), we use:

```
$ git remote -v
origin git@github.com:parrt/msan501.git (fetch)
origin git@github.com:parrt/msan501.git (push)
```