Math 4610 Lecture Notes

Using Git to Work Locally *

Joe Koebbe

September 15, 2019

*These notes are part of an Open Resource Educational project sponsored by Utah State University

Math 4610 Contents: Using Git to Work Locally.

In this part of the notes, a brief primer for **git** that will help you get started using repositories locally. You will also learn to clone and pull repositories from Github to work on existing repositories.

In order to efficiently use **git** you will need to be working in a command terminal. You can use Cygwin or the command windows in Windows or on your Apple computer/laptop. This has already been covered in previous lectures. So, open a terminal and at the prompt, type the following version of the " which" command.

```
% which git
```

The reason for doing this is to determine if **git** is installed on your computer. If so, you can proceed and if not, you will need to install **git** on your computer or use the computers in the Engineering lab. Note that there are a number of ways to install and access the software.

Assuming **git** is installed, in the command terminal make a temporary folder using

```
% mkdir gitexample
```

to create a temporary place to work. Then change directories and look at what is in the folder.

```
% cd gitexample
% ls
```

The folder should (but does not need to be) empty. Next, we will initialize a repository in the folder. There are lots of options and flags that can be used with git. We will just use a few in this primer. So, type

```
% git init
```

The command only takes a second or two and will identify the folder as a repository folder. The output of the command will look something like the following.

```
Initialized empty Git repository in /cygdrive/m/gitexample/.git/
```

Note that the path shown for the folder is dependent on your computer and where you are doing this work.

To see what has been put in the folder you can use the command

```
% ls
```

Unless you have options set, the folder will still look empty. So, instead, type

```
% ls -a
```

to display the hidden files. The command will show a subfolder named **.git** that contains all of the repository bookkeeping for the repository. You never really need to know the contents of this folder and it is highly recommended that the contents are not modified. At least meke sure you know what you are doing if you choose to poke around in there.

Now, let's put a file in the folder and see what happens in **git**. Type the command

```
% touch hello.f
```

This command creates an empty file that can be modified and worked with. Before modifying the file, type in the **git** command

```
% git status
```

to determine how things are accounted for in the folder. The output from the status command is the following.

```
On branch master

No commits yet

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

        hello.f

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

The output shows that a file is waiting to be included in the repository. To include the file created.

To add a file to the repository, the **git add** and **commit** command are used to do the work. That is,

```
% git add hello.f

% git commit -a
```

During the execution of the commit command, an editor session is started. You must include a comment to the commit to have the command complete the work. All you need is a short comment like The -a tag is used to include all commits that are listed.
The commit command has lots of options for being selective in how to add files and folders. The output looks like the following.

```
% adding hello.f to the repository
```

The output after the editor comment is entered is something like the following.

```
[master (root-commit) 6a6297f] aaaaaa
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 hello.f
```

This indicates a single file has been included. In the development of code and documentation the commands above are about all you will need to use **git**. However, there is a lot more to the **git** environment. Also, note that the exact form of the output will change slightly due to the fact you are adding different bits of content on different machines.

For example, your entire repository can be pushed up to Github or other VCS sites. We will come back to this. Let's do one more example before continuing. Create a folder called src in the repository.

```
% mkdir src
```

Next, move the file into the folder.

```
% mv hello.f src
```

Move into the folder using

```
% cd src
```

and then edit the file to do the hello world example. That is, using

```
% vim hello.f
```

and edit in the lines as shown below.

```
      program main
      print *, "hello world"
      stop
      end
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

Save the file and then compile the file. That is,

```
% gfortran hello.f
```

Note that another file will be created named **a.exe** that can be executed as in earlier lectures. Now that we have done a little work, we can use the status command to see how things have changed. Use

```
% git status
```

which results in

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

        deleted:    ../hello.f

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

        ./

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

It should be noted that the git VCS can be invoked in any folder within the repository folder. This allows you to continue to work without going back to the root folder for the repository.
The result indicates that we need to add the current folder. So, type

```
% git add ./
```

Finally, commit the changes using

```
% git commit -a
```

and adding a comment in the editor as above. Once this is done the work is now committed to the repository. It always is a good idea to use the status command to make sure everything has been included or excluded.

The second part of this lesson involves cloning a repository from Github. There are several steps that need to be taken care of first. There are a couple of configuration parameters that need to be set. First move to a clean directory - say we name it tempWork. We do not want to perform an initialization for the repository. Instead, we will need to configure some things in **git**. In particular, you will need to configure the user name and email. So, type

```
% git config user.name yourChoice
```

and

```
% git config user.email yourChoice@some.email.isp
```

You will need to fill in the names. Note that the user name is something you can choose. As your instructor, I would suggest something simple and all in lower case letters. It is a bit easier to remember this. For the email, choose an email address you read most often. That way you can monitor what, when, and where things are happening when you work with either **git** locally, or with **Github** out in the real world.

Once you have the configuration step done, you are ready to clone a repository locally. So, start by changing to a working directory. For example, you can create a directory, say

```
% mkdir tempWork
```

and then

```
% cd tempWork
```

Now for the cloning. Type in

```
% git clone https://www.github.com/Github_username/Github_repository_name
```

When the command is launched, the repository will be created locally and then will clone the entire contents of the repository you have chosen. For students in Math 4610, it would be a good idea to clone the " math4610"repository that you have started for the course. The command for the Math 4610 repository should look like

```
% git clone https://www.github.com/Github_username/math4610
```

The result will be a repository that you can work on locally.

The last bit is to make sure you know how to make changes either locally or on your github account and make sure the local repository and the Github repository are exactly the same. If you make changes locally on your copy of the repository you should first add and remove any files using

```
% git add ...
% git remove ....
```

following by a commit

```
% git commit -a
```

Then the big step is to use the **push** command. The syntax is the following.

```
% git push
```

You will be prompted for your user name (on Github) and then your password. The command will then proceed to merge your local work with the repository on Github. If you modify your repository on Github, you pull content to the local repository using

```
% git pull
```

This makes sure change on Github are reflected on the local repository. Note that your will again be prompted for your user name and password on Github.

Students should note that there are a lot of git command with tons of options to do all kinds of modifications to your repositories. To find out what is available type

```
% git
```

for a list of commands and options. The output from the command looks like the following.

```
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect     Use binary search to find the commit that introduced a bug
   grep       Print lines matching a pattern
   log        Show commit logs
   show       Show various types of objects
   status     Show the working tree status

grow, mark and tweak your common history
   branch     List, create, or delete branches
   checkout   Switch branches or restore working tree files
   commit     Record changes to the repository
   diff       Show changes between commits, commit and working tree, etc
   merge      Join two or more development histories together
   rebase     Reapply commits on top of another base tip
   tag        Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
```

```
   fetch      Download objects and refs from another repository
   pull       Fetch from and integrate with another repository or a local branch
   push       Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
```

There are also a wealth of on-line resources and books that you can get if you intend to do a lot more with this computational utility.