Team Tools

git: version control system

Version control refers to a system that keeps track of all changes to a set of documents (i.e., the revisions, or *versions*). Its principal advantages are: 1) the ability to restore a team's codebase to any point in history, and 2) the ability to *branch* a parallel codebase in order to do development without disturbing the **main** branch, and then *merge* a development branch back into the **main** branch.

Git was developed by Guido van Rossum to create an open-source version of **BitKeeper**, which is a distributed, lightweight and very fast VCS. Other popular systems include **CSV**, **SVN** and **ClearCase**.

Git's docs (https://git-scm.com/) are very clear and extensive. Here is a quick rundown of the highlights:

Creating or cloning a new repository (https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository)

git init to create a brand-new repository
git clone to create a local copy of a repository on the remote server (a *github*)

Adding, changing and committing files to the local repository (https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository)

git add to add a new (*or changed*) file to the *staging area*
git status to see what files have been changed but not added, or changed and added
git diff to see what changes have been made but not committed, or changes between commits
git commit to commit the added changes to the local repository

Reviewing commit history (https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History) with git log

git log to review individual commits

The git commit cycle: add file, commit, push

git is unlike other repository schemes (**csv**, **subversion**) in that each user has his or her own *local repository*, which is kept in sync with the *remote repository*. This allows a user to make commits to a repo without a network connection (one will be needed to eventually sync the commits).

The process of committing changes to a team repository is 3-fold: stage a file, commit the file locally, then push the changes to remote.

git clone a repository
We start by copying down a repository to our local **git** instance. This creates a local repository, a copy (clone) of the remote repo.

```
$ git clone https://github.com/NYU-Python/david-blaikie-solutions
Cloning into 'david-blaikie-solutions'...
remote: Counting objects: 46, done.
remote: Total 46 (delta 0), reused 0 (delta 0), pack-reused 46
Unpacking objects: 100% (46/46), done.
Checking connectivity... done.
$
```

add or change a file (in this case I modified **test-1.py**)

git status to see that the file has changed
This command shows us the status of all files on our system: modified but not staged, staged but committed and committed but not pushed.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add ..." to update what will be committed)
  (use "git checkout -- ..." to discard changes in working directory)

        modified:   test-1.py

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

git add the file (whether added or changed) to the staging area

```
$ git add test-1.py
$
```

git status to see that the file has been added

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD ..." to unstage)

        modified:   test-1.py

$
```

git commit the file to the *local repository*

```
$ git commit -m 'made a trivial change'
[master e6309c9] made a trivial change
 1 file changed, 4 insertions(+)
$
```

git status to see that the file has been committed, and that our *local* repository is now "one commit ahead" of the *remote* repository (known as *origin*

```
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
```

git pull to pull down any changes that have been made by other contributors

```
$ git pull
Already up-to-date.
$
```

git push to push local commit(s) to the remote repo
The *remote repo* in our case is **github.com**, although many companies choose to host their own private remote repository.

```
$ git push
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 318 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To https://github.com/NYU-Python/david-blaikie-solutions
   2ce8e49..e6309c9  master -> master
$
```

Once we've pushed changes, we should be able to see the changes on **github.com** (or our company's private remote repo).

git branching basics

A branch is a "line of development", made up of a series of committed changes toward a particular new feature or bug fix.

see what branches are available	**git branch**
create a new branch	**git branch *newfeature***
switch to the new branch	**git checkout *newfeature***
add and/or change file(s)	[edit file]
commit changes	git commit -m 'making changes'
push to the branch	git push
if needed, push the branch to the remote repo	**git push origin HEAD**
switch back to master branch	**git checkout master**
merge the changed branch back to master	**git merge *newfeature***
push the branch to remote	**git push origin master**
delete the *newfeature* branch locally	**git branch -d *newfeature***
delete the *newfeature* branch remotely	**git push origin :*newfeature***

git's tutorial on branches (https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell) is an extremely clear

approach to documenting this sometimes challenging topic.

PEP257: docstrings

PEP257 (https://www.python.org/dev/peps/pep-0257/) defines the overall spec for docstrings.

If you violate these conventions, the worst you'll get is some dirty looks. But some software (such as the Docutils docstring processing system) will be aware of the conventions, so following them will get you the best results.

-- David Goodger, Guido van Rossum

The actual format is discretionary; below I provide a basic format.

A **docstring** is a string that appears, *standalone*, as the first string in one of the following:

a script/module
a class
a function or method

The presence of this string *as the very first statement* in the module/class/function/method populates the __doc__ attribute of that object.

```
#!/usr/bin/env python

"""myprog.py:  show a sample of docstrings"""

def myfunc():
    """this is myfunc"""

print __doc__                # 'myprog.py:  show a sample of docstrings'
print myfunc.__doc__         # 'this is myfunc'
```

All modules should normally have docstrings, and all functions and classes exported by a module should also have docstrings. Public methods (including the __init__ constructor) should also have docstrings. A package may be documented in the module docstring of the __init__.py file in the package directory.

It is also allowable to add a docstring after each variable assignment inside a module, function or class. Everything except for the format of the initial docstring (which came from a module I wrote for AppNexus) is an example from PEP257:

```
"""
NAME
    populate_account_tables.py

DESCRIPTION
    select data from SalesForce, vertica and mysql
    make selected changes
    insert into fiba mysql tables

SYNOPSIS
    populate_account_tables.py
    populate_account_tables.py table1 table2     # process some tables, not others

VERSION
    2.0 (using Casserole)

AUTHOR
    David Blaikie (dblaikie@appnexus.com)
"""

num_tried = 3
"""Number of times to try the database before failing out."""

class Process(object):

    """ Class to process data. """

    c = 'class attribute'
    """This is Process.c's docstring."""

    def __init__(self):
        """Method __init__'s docstring."""

        self.i = 'instance attribute'
        """This is self.i's docstring."""

def split(x, delim):
    """Split strings according to a supplied delimeter."""

    splitlist = x.split(delim)
    """the resulting list to be returned"""

    return splitlist
```

These docstrings can be automatically read and formatted by a docstring reader. **docutils** is a built-in processer for docs, although the documentation for *this* module is a bit involved. We'll discuss the more-easy-to-use **Sphinx** next.

Sphinx: easy docs from docstrings

Docstrings are read directly from Python code, and automatically built into documentation documents in the desired format (default HTML).

install Sphinx

Sphinx is included with Anaconda Python. It requires **docutils** and **Jinja2**, which are also included with Anaconda Python.

create a directory for docs as well as for source

It's important to have a dedicated source directory for a project (customarily called **src**, but up to your discretion). You should also create a separate directory for documentation.

run sphinx-quickstart
This automated script asks a series of questions, most of which should be accepted with the suggested defaults, except for two: say 'yes' to separate source and build directories; and say 'yes' to 'autodoc' (which automatically geneates documentation)

edit conf.py
This file generated by sphinx-quickstart should be informed of the location of your source with a basic python statement:

```
sys.path.insert(0, '../src')   # relative path to your docs
```

run sphinx-apidoc
This script finds all of the modules and classes within your code and creates special .rst files for each

make html
This command generates documentation.

I found the process of generating documentation to be less than smooth, but with practice it became easier. A few hours following Giselle Zeno's tutorial (http://gisellezeno.com/tutorials/sphinx-for-python-documentation.html) should be sufficient to become comfortable with Sphinx.

I built some Sphinix docs against a test lib - they look very pretty (../data/docs/build/html/py-modindex.html) and suspiciously similar to documentation for Flask (http://flask.pocoo.org/docs/0.10/).

pdb: the Python Debugger

Print statements and **raw_input()** are useful tools in debugging. But sometimes a dedicated *debugger* can give us a lot more power in reviewing our programs *as they are running*.

A **pdb.set_trace()** statement placed in your code will cause program execution to pause (similar to **raw_input()**), but allowing you to interact with the script's variables while the script is paused.

```
import pdb

def dosomething(aa):
    print "now I'm inside dosomething()
    aa = aa + 5
    return aa

print 'hello, debugger'

for counter in range(5):
    countsum = 0
    countsum = countsum + counter
    pdb.set_trace()

pdb.set_trace()
newval = dosomething(5)
print "now I'm moving on"
```

At the point that python reaches the **pdb.set_trace()** statement, it will pause and you now have the opportunity to examine and set variables, call functions, etc.

There are several control options when working within the paused shell, but here are the ones I use most often:

```
n(ext):  execute next statement
s(tep):  execute next statement (and enter a function if one is encountered)
c(ontinue):  resume program execution until the next stopping point
```

iPython

The iPython shell is a "smart shell" for running Python programs. It's a fantastic tool not only for running code snippets (in the same way as the Python interactive shell) but can be used to run scripts directly through the shell and cause the script to drop back to the shell for testing and analysis (similar to **pdb** but with enhanced capabilities).

command/statement history

command history	[up arrow to cycle through]
show all history (prior iPython commands)	%history
show prior commands (last 3 commands)	%history 1-3
upload prior commands to github's **gist** repo	%pastebin 1-3

environment and object inspection

show currently active variables	%who
attribute tab completion	type an object name and 'dot', then [Tab]
show info about object (docstring, function signature, file location)	[object]?
show source, file, function signature	[object]??
show object docstring (for functions)	%pdoc [object]
search for a variable	%psearch [name]

run python scripts and code

run a python script, stop without exiting with variables available	%run [python script]
run a program in debug mode	%debug [scriptname]
open a temporary editor to enter code	%edit

external commands

built-in shell commands	%ls, %cd, %pwd
set aliases to shell commands	%alias [shortcut] [command-line command]

The Zen of Python

One of the advantages of Python is its emphasis on design elegance. The non-backwards-compatible **Python 3** is a clear indication that doing things "the right way" (however it may be defined) is more important than popularity.

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one -- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

PEP8: the Python design standard

A **PEP** or **Python Enhancement Proposal** is a series of numbered discussion documents that propose new features to be added to Python. A significant discussion over whether or not an addition is useful, meaningful, worthwhile, and "Pythonic" accompanies each proposal. Proposals are accepted or rejected by common consensus (with special weight given to the opinion of **Guido van Rossum**, the proclaimed **Benevolent Dictator for Life**).

PEP8 (https://www.python.org/dev/peps/pep-0008/) is often cited in online discussions as the standard guide for Python style and design. It treats both specific style decisions (such as whether to include a space before an operator; the case and format of variable names; whether to include one blank line or two after a function definition) as well as design considerations (how to design for inheritance, when/how to write return statements, etc.).

While the proposal itself proclaims "A Foolish Consistency is the Hobgoblin of Little Minds" it is generally accepted as the standard to follow.

"*A universal convention supplies all of maintainability, clarity, consistency, and a foundation for good programming habits too. What it doesn't do is insist that you follow it against your will. That's Python!*"

â€”Tim Peters on comp.lang.python, 2001-06-16

important conventions in **PEP8**

4-space indents
maximum line length: 79 characters
wrapping long lines with parentheses
2 blank lines before and after functions
imports on separate lines, 'absolute' imports recommended
single space on either sides of operators

naming style is not recommended, but general convention is lowercase_with_underscores

Adhering to the 79-character limit

The original CRT terminal screens had an 80-character width; this is how the 79-character limit was born (wrapped lines are difficult to read).

Many text editors and IDEs can be set to display the 80-character limit onscreen and/or warn the user when the limit exceeded.

Even though modern screens allow for pretty much any length (since you can size the text to any size), the 79-character limit is still considered an important discipline, since it will allow multiple files to be displayed side-by-side.

When attempting to adhere to this limit, we must learn how to wrap our lines. Wrapping long lines is usually done with parentheses, for example:

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

I have found that my preference for explicit variable names makes respecting this limit to make life more difficult.

Some developers hold an opinion (acknowledged by PEP8) that the 79-character limit is antiquated (since it was based in the old terminal width) and that another limit, up to 100 characters, should be adopted. PEP8 acknowledges that with agreement among developers, it should be permissible to extend the limit. You can see some of what is said regarding the debate in this spirited discussion (http://stackoverflow.com/questions/4841226/how-do-i-keep-python-code-under-80-chars-without-making-it-ugly).