See the following for giving feedback:
https://docs.google.com/document/d/1zWWfosXmbq9m6ZIy0D2ZFbxpqYLAVUSI0xjB3xUuuLU/edit?usp=sharing

Deprecated docs. There may be some useful stuff in here but you should check out the main docs online for using run_afni_tests.py:
https://afni.nimh.nih.gov/pub/dist/doc/htmldoc/boring/main_toc.html
Or
https://github.com/afni/afni/tree/master/tests

You may also want to check out the build docs

Requirements specific to testing:

Testing quickstart (using docker):

Summary of a test session:

Writing a test:

Other issues with the git-annex/datalad stuff:

Versions of testing related data:

Hackable issues:

Tests should be run using “pytest” or “python3 -m pytest”, from the “tests” directory in the AFNI repository. In general pytest -h gives a very useful guide to using pytest. Note the “custom options” added for our specific tests suite.

Requirements specific to testing:

HARD DEPS: Nibabel, git-annex, git, datalad, python 3.6, rsync, numpy, pytest, attr

SOFT DEPS: pytest-parallel (can use more threads during testing), autopep8, and black (just used for testing python style in the tests directory)

Testing quickstart (using docker):

Install docker: https://docs.docker.com/get-docker/ (unless you create and add the docker group you will have to run docker commands with sudo)

Cd into afni repo and run:

```
docker pull afni/afni_cmake_build
docker run \
   --user=root  \
   -e CHOWN_HOME="yes"  -e CHOWN_HOME_OPTS='-R'  -e CHOWN_EXTRA="/opt/"  -e
CHOWN_EXTRA_OPTS='-R' \
   -e CONTAINER_UID=$(id -u) -e CONTAINER_GID=$(id -g) \
   --rm -ti \
   -v $PWD/tests/afni_ci_test_data:/opt/afni/src/tests/afni_ci_test_data \
   afni/afni_cmake_build \
   bash -c 'bash /opt/afni/src/tests/run_tests_cmake_build.sh'
```

```
docker run --rm -ti -v $PWD:/opt/src/afni afni/afni_cmake_build
```

Inside the container execute a specific test by typing (-k flag searches for a regex, scripts is a positional argument to tell pytest where to search):

```
ARGS='scripts -k afni' ninja pytest
```

The first time you execute some of the tools may be rebuilt but subsequent runs should work without rebuilding (assuming you have not changed any of the C files)

Summary of a test session:

One or more tests are selected and run by the user (-k <pattern> will select matching tests for example). Each test is a function in a test module within the tests-tree. If everything goes well a minimal report of success is made upon completion.

Upon failure...

When failures occur, a traceback is printed to the terminal with some printed variables to help debug the cause of the error. If your weapon of choice is python, adding the --pdb flag can make debugging easier. You will be dropped into the call stack at the point of failure. Often, typing “u” once will move you one frame higher in the call stack giving you access to “cmd” the command that failed, and “proc” the object returned by the failed run containing stdout and stderr. If you prefer the shell, the failed command is printed out as part of the failure report. Copy it and begin to iterate....

Writing a test:

Add tests. Consider the importance of the program you are adding tests for (rough guide here)

A description below lays out how to define some input data and write a test function for a given tool. The basic overview is that for each test function a command is constructed using data previously defined, executed to check it runs without error, and finally the output of the successful command is checked against previous output to make sure no unexpected change has occurred. The approach taken to implement this is:

1. Input data files are specified as values of the “data_paths” dictionary defined in the test module outside of test functions themselves. The paths to the data files are a relative reference to a file within the test data directory (afni_repo/tests/afni_ci_test_data). The keys for the dictionary (eg “anat”) are used to reference the files from within the tests themselves.
2. Each test is defined in a python function. The function must start with the string “test_”. Almost always, the argument taken by each test function is “data”. The object is defined

in afni_repo/tests/conftest.py. The data object can be used in the test itself to access the predefined data paths (eg “data.anat”) and some other useful paths defined dynamically - “module_data_dir”, “outdir”, “logdir”, “comparison_dir” - and the test name itself. All output should be saved to “data.outdir”. The run_cmd object will take the string stored in the cmd variable (cmd by convention only), define a default environment to execute the command in, check that it succeeds, writes logs, and return the object that describes the results of the execution.

3. Assuming the command executes without failure, the output files can be compared against previous output files. The easiest way to do this is to use the helper class in the tools module: “OutputDiffer”. By default OutputDiffer iterates through all the files in the output directory (including the stdout and stderr logs) and compares them against pre-existing output (described in detail later). The comparison performed depends on filetype and will no doubt evolve over time as the various use-cases that we need to contend with arise. Currently it distinguishes between 1d files, multi-d neuroimaging files, log files, user-defined text files. Files not fitting into these categories currently are compared byte by byte. It is a good idea to check a few pre-existing tests to get a sense of the various inputs for this.

Comparing with previous output:

When running tests one has 3 choices for comparing test output:

1. the data stored in the default sample output in the test data repository
2. the data stored as part of a test run in output_of_tests
3. the data stored as a result of running a test session with the --create_sample_output flag

By default the comparison directory used is afni_ci_test_data/sample_test_output. This should contain the required data for all tests so that anyone can easily run them on any system.

A directory for comparison can be any relevant previous test output directory. These are created in tests/output_of_tests after each test session. The user may specify an alternative directory

with the pytest custom option “--diff_with_outdir”. Provided each current test has a corresponding directory in the comparison output tree the test should proceed as normal, otherwise a FileNotFoundError is raised.

In addition to using previous test output directories the comparison directory can also be a sample output directory created explicitly for this purpose. The sample output directory might be identical to the raw output but it could also be a subsample of the full output of the command being tested. This occurs when only a subset of files are explicitly compared.

The “--create_sample_output” flag will execute a test without performing any comparison with any pre-existing data (tests should otherwise pass for this to work though). Instead, when this flag is used, all files that would have been compared are copied into a sample outputs directory.

If one desires to permanently save such a sample output of a test run to the default sample output directory, run the tests with the “--save_sample_output” flag (not implemented) . Specifically this command saves all new output/changed data to the test data repository. If some sample output data for a test already exists, only files that “differ”, as defined by the assert tools used for the tests themselves, will be copied over and committed to the repository.

For adding data to the datalad repository follow the instructions at:

https://github.com/afni/afni_ci_test_data

~~Currently adding this data is not streamlined. There’s lots of issues that occur while attempting to add it. Some points in no particular order to help the process (a lot of this has been resolved by solving race conditions in data download):~~

- ~~One should always generate the sample output test data from the dev container for most consistent results (using ---source-mode host) From what I can tell chowning the sample_output directory to the uid of the host and then committing the change files from outside the container seems to be the best approach.~~
    - ~~The config should be something along the lines of (this occasionally gets wiped):~~

~~url = https://afni.nimh.nih.gov/pub/dist/data/afni_ci_test_data/.git~~

~~pushurl = afni:/fraid/pub/dist/data/afni_ci_test_data/.git~~

~~fetch = +refs/heads/*:refs/remotes/afni_ci_test_data/*~~

~~annex-bare = false~~

~~annex-uuid = c1ce38d5-c2ef-48c6-a1f2-e207215d0717~~

~~Other issues with the git-annex/datalad stuff:~~

- ~~Something like the following is necessary to reindex files after additions: " ~/@make.directory.index -nested -dirs /fraid/pub/dist/data"~~
- ~~Sometimes the quick and easy fix is to run datalad update~~
- ~~Check commits logs in the afni_ci_test_data repo, these provide an excellent guide to commands that were used to add data to the repository.~~
- ~~Sometimes when the remote is not accessible the annex remote will be disabled. This can be undone by modifying .git/config or using git-annex enableremote afni_ci_test_data~~
- ~~Datalad does some website indexes which work nicely for the datalad template website. There are some details on the afni server that made this challenging to keep working though.~~
- ~~When pushing data you can't use the default url provided in .git/config. Instead you must supply a push url. Using ssh that would be something like afni.nimh.nih.gov:/fraid/pub/dist/data/afni_ci_test_data~~
- ~~One needs to pay attention if the annex remote used as a common data source on the afni server is bare or not. If bare set annex_bare value to true in the config for afni_ci_test_data. And the https url will not end in /.git (This should not be touched as it is setup fine now. But useful to know if another datalad repository were considered for something)~~
- ~~In order to pass tests online you need to be careful that you have not only commited your changes in afni_ci_test_data but in turn have made a commit in the main code repo to note that the submodule has changed.~~
- ~~Sometime you want to tweak the history. This is something you would consider casually typically. It starts to get very difficult with datalad/git-annex. I'm not sure whether it's for the purposes of reproducibility that this is so difficult but it often goes pear-shaped. To~~

~~recover from this mess: rewrite the git history and force push it, run git annex forget --force, delete remove synced/… branches, force push master and git annex branches, and finally run git-annex sync. Datalad publish should have little to do at this point. I don't know if all of this is needed. But this is sufficient at least. EDIT: do not do this no matter how confusing you think a couple of erroneous pushes would be!~~

- ~~A little gotcha is that in the container at one point (I think I have fixed it now) the user.name and user.email were set to something not so useful. This will likely happen if you run tests without configuring git first. Check .git/config in the test data repo~~
- ~~(EDIT: Cleaner solution for the following was to make it difficult to use the container for development without switching the id of the files to the user on the host machine. This prevents all of the permissions issues happening). Careful of permissions. It seems setting read and execute permissions has fixed difficult problems sometimes. One way this happens is it seems some ownership changes occur when running tests inside the container. Chowning the sample -output files and the .git repo (to the id of the host user) seems to be the solution for now until a cleaner one is figured out~~
- ~~Sometimes the git annex version is set to 8. Not sure why. If you change it back to 7 in the .git/config all is well (this fix is also run at the start of each test run)~~

Versions of testing related data:

It should be noted that the test data repository is a datalad repository. A datalad repository is simply a git repository that uses git-annex to manage large binary files with a little extra metadata. Apart from making the various required operations easier, datalad itself is not essential (although currently all operations are performed with datalad). Just like any git repository, the data used for testing is versioned to prevent test-data becoming out of sync with the tests themselves. A full record of the data is stored in the github repository while the large files themselves are stored on the afni server. Data is only downloaded as required to run any given test module.

Hackable issues:

- Add tests. Consider the importance of the program you are adding tests for (rough guide here)
- Work towards improving text diffing. Currently paths, host, and user are normalized. What about blank lines? Afni_proc on mac with current sample output has a small and irrelevant diff.
- Add warning when save_sample_output is not run from within a container
- Get coverage.py working. Works with no source file specification but otherwise checks if the "package" is imported. Add a badge to the github repo.
- Write tests that verify that various files that are different raise assertion errors when they are different.
- Documentation: Add some useful example pytest/datalad/annex commands in the tests directory
- Make testing robust to checking out different commits. The current plan is to store the data directory as a submodule of the afni repo. This means that checking out a previous commit should checkout a different version of the submodule. Write a test for this process to confirm it works and the correct file version is resolved/downloaded as required.
- Currently circleci builds and tests for version updates and pull requests. Should still do this for pull requests. In addition, for version updates, the full test suite should be run on circle ci.
- Using the function " compute_expected_euclidean_distance_for_affine" in tools.py modify the 3dAllineate test so that the "tolerance" on the comparison is the expected value of displacement (rather than the current approach of defining tolerances for numbers within the matrix itself).

- Enable save_sample_output for run_cmd: would require pulling update_sample_output from class. Trixiness? Don't want to fail tests because of failed syncing. Write tests for

the --save_sample_output flag to pytest. If no files are compared it should not create the directory and report that. Some of the test infrastructure logic needs changing here. Currently the directories are made and required even if no output is generated.

Currently needs to be implemented:

- ~~py2/3 parameterization (for circleci at least), have a think about best way to cope with output. Uniquely name for each interpreter or have output_directories that get compared to a unified sample directory. In this case would need to be careful about output log number (EDIT: not going to do this. There may be meaningless differences between python versions that we don't want to keep track of). For now this is implemented by defining "python_interpreter" as an argument to the test function and passing this as a named argument to the OutputDiffer instance.~~
- ~~Missing directories etc should fail either once at setup or within the differ class, not at different places in conftest.py~~
- ~~Consider reorganizing the logic of slow, veryslow, and other test selection flags.~~
- ~~Figure out how to hard reset and sync with remotes... Do not hard resest… syncing with remotes is now described~~
- ~~Figure out how to get openneuro datasets... Can download with aws s3 sync --no-sign-request s3://openneuro.org/dataset dataset but not with datalad get~~