organizing code, data, results

- maybe useful: https://scipy-lectures.org/intro/language/reusing_code.html
- much of programming and data analysis revolves around organizing your code, data and results
- open questions:
  - what kind of naming scheme to use for files and folders?
  - where to store code, data and results relative to each other? All together in one place, or separately?
  - if separately, how to track which code was run on which data, producing which results?
  - will your results be reproducible if you re-run your analysis on the same data in a year?
- some general tips:
  - use long descriptive file/folder names
    - more info is generally better, but at some point file names might get too long
    - avoid spaces, use `_` instead - makes command line use easier
  - consider including date (& maybe time) stamps in file/folder names, to better track down history
  - when naming results files/folders, include some reference to the source data from which the results were generated, as well as the name of the analysis that was run on that data to generate that result
  - above all: be consistent! This will let you programmatically manipulate all of your data
- we've been mostly executing short code fragments in IPython/Jupyter
  - usually a few lines of code at a time, separated by output, or embedded figures
  - Jupyter notebooks are well suited for exploratory analysis, and for storing all your code, data and results in one document
  - one very simple way of organizing things would be to have one Python script or Jupyter notebook per experiment, each generating results for that experiment only
  - e.g. storing data, code and results together for an experiment with both behaviour and imaging for some subjects, and only behaviour for other subjects:

```
experiments/
    subj01/
        subj01_exp01_2017_01_08_behaviour.xlsx
        subj01_exp01_2017_01_08_images/
        subj01_exp01_2017_01_08_analysis.ipynb
        subj01_exp02_2017_01_09_behaviour.xlsx
        subj01_exp02_2017_01_09_images/
        subj01_exp02_2017_01_09_analysis.ipynb
    subj02/
        subj02_exp07_2017_02_10_behaviour.xlsx
        subj02_exp07_2017_02_10_analysis.ipynb
        subj02_exp08_2017_02_11_behaviour.xlsx
        subj02_exp08_2017_02_11_analysis.ipynb
    ...
```

- over time, as you collect new data, you probably want to run the same code on the new data
- you could simply copy and paste your previous .ipynb to your new data folders, maybe change a line or two in the .ipynb to load the new data from a differently named file, and then run the whole notebook

- but...

    - what if you want to collect analysis results **across** all your experiments?
    - or, what if you find a bug in your code, or what if you want to add an extra analysis step, or format your figures differently, and then re-run your code on **all** or your data, and regenerate **all** of your results?
    - or, what if your data is really huge, and has to be stored in a shared location, like on a server, so that multiple users can access it?
    - your lab might also require centralized data storage to ensure all data is regularly backed up and protected against data loss
    - in any of these cases, it might be inappropriate, even messy, to mix code and results in with your centralized data
    - better to store your data, code and results in separate locations, and write Python scripts/modules (`.py` files) or Jupyter notebooks (`.ipynb` files) that work on all of your data, instead of just one particular experiment:

```
data/
    subj01/
        subj01_exp01_2017_01_08.xlsx
        subj01_exp01_2017_01_08_images/
        subj01_exp02_2017_01_09.xlsx
        subj01_exp02_2017_01_09_images/
    subj02/
        subj02_exp07_2017_02_10.xlsx
        subj02_exp08_2017_02_11.xlsx

code/
    common.py
    analyze_behaviour.py
    analyze_imaging.py

results/
    subj01/
        subj01_exp01_2017_01_08_behaviour.pdf
        subj01_exp01_2017_01_08_imaging.pdf
        subj01_exp02_2017_01_09_behaviour.pdf
        subj01_exp02_2017_01_09_imaging.pdf
    subj02/
        subj02_exp07_2017_02_10_behaviour.pdf
        subj02_exp08_2017_02_11_behaviour.pdf
    all_subjects_behaviour.pdf
    all_subjects_imaging.pdf
    ...
```

    - keeping code/data/results separate might also reduce the number of files per folder, making file listings easier to examine at once
    - another benefit is that your original raw data should never be modified - storing it separately from your code and results will help ensure that you don't modify (or delete!) it by accident
        - another safeguard is to only give yourself read access to the raw data when doing analysis - details of that depend on your OS and filesystem
    - code folder might consist of both scripts that you execute, and modules that you import
        - scripts are something you *run*, either directly at the command line (`python my_script.py`) or from within IPython/Jupyter (`run -i my_script.py`)
        - modules are something you *import*, typically define functions that you might use throughout many scripts. script might start with `from common import plot_reaction_times` and then use that function somewhere in the script
        - modules are great because you don't need to write the same function multiple times, just once, then import it into each script where you need it
        - this encourages you to not repeat code
        - less code repetition is very very desirable:
            - less code to read, understand and debug
            - bugs are detected faster, because all your scripts rely on the same common functions in your module(s) - one error will likely affect all of your scripts, instead of just one or two scripts
- Python lets you control what happens if a `.py` file is being run as a script vs. if it's being imported as a module
    - at the end of the file, you can write: `if __name__ == '__main__':`
    - this says "if I'm being run as a script, do the following:"
    - if the `.py` file is being imported then the `__name__ == '__main__'` is `False`, and the code that follows isn't executed
    - if the `.py` file is being run as a script, then the `__name__ == '__main__'` is `True`, and the code that follows is executed
- script and module file names are typically lowercase, no spaces, use underscores if needed
- writing easily readable code:
    - some style guides:
        - http://pep8.org
        - https://docs.python-guide.org/writing/style/
- Wilson2014 - best practices for scientific computing
    - http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745

doctests as light unit testing

- doctests are function docstrings that have some tests embedded in them that can be automatically run as part of a test suite

- tests are just a copy and paste of what the function input and output are at the plain Python prompt
- besides being functional, they also serve as examples to the user of how the function is expected to work
- See https://docs.python.org/3/library/doctest.html
- in a file named e.g. `example.py`, define:

  ```python
  def squared(x):
      """Return the square of x.

      >>> squared(2)
      4
      >>> squared(3)
      9
      >>> squared(4)
      15
      """
      return x**2
  ```

  Then run `python -m doctest -v example.py` from the command line to run all the doctests in `example.py`, and print a verbose report about each one. Or within IPython/Jupyter notebook, type `run -m doctest -v example.py`.

version control with Git

- a version control system (VCS) is a way of keeping track of changes to a set of files (project) over time
- Git is the most popular VCS today
  - many big open source projects use Git, including Linux and Python
  - GitHub is a popular commercial website that uses Git to help you collaborate with others
- Git stores a project's history of changes in a "repository", which is typically created as a hidden `.git` folder in the same folder as the project
- Git works best with plain text files, i.e. files you can view in a plain text editor
- Git compares old revisions of files with new revisions, and generates a line-by-line "diff" that shows the difference between the two revisions
  - this way you can see exactly what was changed on what lines from one revision to the next
- every time you make a change to your code, you can do a "commit" which commits those changes permanently to the repository, along with a timestamp, a comment about the commit, and author information
- Git is a command line program, with many subcommands:
- `git config`

    - used to tell Git your name, so that changes that you make to a project show up under your name

    ```
    git config --global user.name "Martin Spacek"
    git config --global user.email git@mspacek.mm.st
    ```

    - check your config with `git config --list`
    - you can always get help on a subcommand with `git <subcommand> --help` or `git help <command>`

- `git init`

    - initializes a git repository in the current directory
    - this creates a hidden `.git` folder, where git will store all versions of all files that you "commit" to the repository
    - you can move or copy this `.git` folder somewhere else, and you'll have a full copy of not only the latest versions of files you've committed to the repo, but also the full version history of all of those files

do Exercises 1 and 2

- `git status`

    - tells you the current status of files in your project folder
        - are they being "tracked" by git? i.e., have they been added to the repo yet?
        - if so, has any file changed since it was last "committed" to the repo?
    - gives you hints about commands you can use to change the state of a file
    - every file you use with git can be in one of 4 different states:
        - untracked: git knows nothing about this file's history
        - unmodified: tracked, hasn't changed since last commit
        - modified: tracked, but has changed since last commit
        - staged: changes are ready to be committed to the git repo and entered into the change log
    - see `git_file_states.png`
    - git sees only what's in the project folder (and all its child folders)

- `git add`

    - let's say you create a file named `filename.txt` in your project folder
    - `git add filename.txt` tells git to start noticing that file by "staging" it, i.e. marking it as ready to be committed

- `git diff`

    - if there are any "tracked" files that have been modified, `git diff` will show you how they've changed, on a line-by-line basis
    - diff and long lines don't get along, so best to keep lines below 100 chars max

- `git commit`

    - takes all changes you staged with `git add` and "commits" them permanently into the repository
    - every commit requires a commit message
    - `git commit -m "Commit message"`
    - commit messages should be short, but specific
        - use the diff to help you describe the changes you made and write your commit message
        - convention is to capitalize first word, start with a verb (Add, Fix, Change, etc.), end without a period
        - e.g. `Add filename.txt` or `Fix roundoff error`
    - commit early and often, in small logical chunks
        - chunks are not necessarily files, but subsets of files, maybe multiple subsets across multiple files

- `git log`
    - each commit has an long hexadecimal ID, like `f7e6501a935cf7214389372af84d51e3c6e9988e`
    - this is actually a hash, or fingerprint, of the changes that went into that commit
    - you can refer to a commit by just the first few characters of its ID, i.e. `f7e6501`
    - `git log f7e6501` displays only that one commit

do Exercises 3, 4 and 5

- `git checkout`
    - revert uncommited changes with `git checkout -- filename` or `git checkout --force filename`
    - "check out" the state of your files at a particular time in history by specifying the commit ID:
        - `git checkout f7e6501`
        - notice the `HEAD detached` message in `git status`
        - to go back to the latest version, use `git checkout master`

- `.gitignore` file lets you specify patterns to match which kinds of files/folders git should ignore, so that they never even show up as untracked
    - e.g. `*.temp` - one pattern per line

do Exercises 6 and 7

- you can version control any files, not just code or plain text
    - images, PDFs, excel files, word docs... - these are binary files, can't be viewed in text editor
    - because these binary files don't have the concept of line numbers, you can't do a diff on them

- `git clone`
    - copies a remote repository, such as one on GitHub, to your local file system
    - `git clone git@github.com:SciPyCourse2021/notes.git` clones the class notes to your current working directory

  - if you want, you can then use the local repository as your own, i.e. make changes and commit them. You can only "push" those changes back to the server (github) if you have the right permissions

- `git pull`
  - updates your local copy of a repository from the remote one
  - if you've made your own local changes and commits, conflicts might arise

- `git push`
  - pushes your local commits up to the remote one
  - if someone else has made commits and pushed them to the server since the last time you pulled, then you'll first have to pull again and resolve any conflicts locally, then try pushing again

- GUIs for Git:
  - see https://git-scm.com/download/gui/linux for an extensive list
  - for all OSes, these two (should) come with Git, but they're a bit rough:
    - `git gui` - for staging and committing
    - `gitk` - for browsing the history (graphical version of `git log`)
  - Linux/Mac/Windows:
    - `git-cola` - https://git-cola.github.io/ - Martin's current favourite
  - Windows:
    - TortoiseGit - https://tortoisegit.org
      - modifies Windows Explorer icons, right click menu
    - GitHub desktop - https://desktop.github.com/
  - Mac
    - GitHub desktop - https://desktop.github.com/

- psychological reasons for VC:
  - each commit is a nice little reward, gives closure on some feature or fix
  - hard to remember what you've actually done, when you're coding, or editing files in general. VC is a way of accounting for the time you've spent sitting on the computer, and a way of sharing that progress with others

- cheat sheet site: http://rogerdudler.github.io/git-guide/