BETTER SCIENCE CODE

Eric Denovellis

Presentation: **https://edeno.github.io/Better-Science-Code**

Repository: **https://github.com/edeno/Better-Science-Code**

Why should you care about producing good code

Why should you care about producing good code

REASON 1. Doing good science!

Don’t want to have to retract papers because the code had bugs

Why should you care about producing good code

Following good coding practices reduces the chance of making mistakes.

> *“As the complexity of a software program increases, the likelihood of undiscovered bugs quickly reaches certainty” – Poldrack et al. 2017*

We are writing *complex code*

Why should you care about producing good code

REASON 2. Want to remember what the code does months later

> *“The single biggest reason you should write nice code is so that your future self can understand it.” – Greg Wilson*

> *“All code has at least one collaborator and that is future you.” – Hadley Wickham*

Why should you care about producing good code

REASON 3. Want to be able to share it with other people

REASON 4. Avoid introducing new errors

Why should you care about producing good code

REASON 5. Can serve as a resume for future employers

How to write good code???

Exercise in managing complexity:

- break problems down into smaller components
- eliminate unnecessary dependencies
- keep track of what you did (be organized)

Goal: Want to form good habits

Don’t be overwhelmed *and not do any of these things*

Don’t beat yourself up *if you don’t do all these things all the time*

How to write good code???

STEP 1. Decompose programs into small, well-defined functions

```python
import numpy as np

def bad_function():
    X = np.load('/tmp/123.npy', mmap_mode='r')
    y, x1, x2 = X[:, 0], X[:, 1], X[:, 2]
    z1 = (x1 - x1.mean()) / x1.std()
    Q1, R1 = np.linalg.qr(z1, mode='reduced')
    b1 = np.linalg.solve(R1, np.dot(Q1.T, y1))
    z2 = (x2 - x2.mean()) / x2.std()
    Q2, R2 = np.linalg.qr(z1, mode='reduced')
    b2 = np.linalg.solve(R2, np.dot(Q2.T, y2))
    b = b1 - b2
    np.save('ans.npy', b)
```

```python
import numpy as np

def better_function():
    y, x1, x2 = load_data('/tmp/123.npy')
    b1 = linear_regression(zscore(x1), y)
    b2 = linear_regression(zscore(x2), y)
    b = b1 - b2
    np.save('ans.npy', b)

def load_data(data_name):
    X = np.load(data_name, mmap_mode='r')
    return X[:, 0], X[:, 1], X[:, 2]

def zscore(x):
    return (x - x.mean()) / x.std()

def linear_regression(design_matrix, response):
    Q, R = np.linalg.qr(design_matrix, mode='reduced')
    return np.linalg.solve(R, np.dot(Q.T, response))
```

How to write good code???

Try to keep functions to less than 60 lines (small)

How to write good code???

Try to keep what the function does as simple as possible (well-defined)

How to write good code???

Be ruthless about eliminating duplication of code.

Small, well-defined, without duplicates

```
import numpy as np

def bad_function():
    X = np.load('/tmp/123.npy', mmap_mode='r')
    y, x1, x2 = X[:, 0], X[:, 1], X[:, 2]
    z1 = (x1 - x1.mean()) / x1.std()
    Q1, R1 = np.linalg.qr(z1, mode='reduced')
    b1 = np.linalg.solve(R1, np.dot(Q1.T, y1))
    z2 = (x2 - x2.mean()) / x2.std()
    Q2, R2 = np.linalg.qr(z1, mode='reduced')
    b2 = np.linalg.solve(R2, np.dot(Q2.T, y2))
    b = b1 - b2
    np.save('ans.npy', b)
```

Small, well-defined, without duplicates

```
import numpy as np

def better_function():
    y, x1, x2 = load_data('/tmp/123.npy')
    b1 = linear_regression(zscore(x1), y)
    b2 = linear_regression(zscore(x2), y)
    b = b1 - b2
    np.save('ans.npy', b)

def load_data(data_name):
    X = np.load(data_name, mmap_mode='r')
    return X[:, 0], X[:, 1], X[:, 2]

def zscore(x):
    return (x - x.mean()) / x.std()

def linear_regression(design_matrix, response):
    Q, R = np.linalg.qr(design_matrix, mode='reduced')
    return np.linalg.solve(R, np.dot(Q.T, response))
```

Small, well-defined functions are more *maintainable*

Small, well-defined functions are more *composable*

Small, well-defined functions are more *readable*

* if you give them good names

STEP 2. Use good variable/function names to clarify what things do

Use good variable/function names

```
import numpy as np

def bad_function():
    X = np.load('/tmp/123.npy', mmap_mode='r')
    y, x1, x2 = X[:, 0], X[:, 1], X[:, 2]
    z1 = (x1 - x1.mean()) / x1.std()
    Q1, R1 = np.linalg.qr(z1, mode='reduced')
    b1 = np.linalg.solve(R1, np.dot(Q1.T, y1))
    z2 = (x2 - x2.mean()) / x2.std()
    Q2, R2 = np.linalg.qr(z1, mode='reduced')
    b2 = np.linalg.solve(R2, np.dot(Q2.T, y2))
    b = b1 - b2
    np.save('ans.npy', b)
```

Use good variable/function names

```
import numpy as np

def better_function():
    y, x1, x2 = load_data('/tmp/123.npy')
    b1 = linear_regression(zscore(x1), y)
    b2 = linear_regression(zscore(x2), y)
    b = b1 - b2
    np.save('ans.npy', b)

def load_data(data_name):
    X = np.load(data_name, mmap_mode='r')
    return X[:, 0], X[:, 1], X[:, 2]

def zscore(x):
    return (x - x.mean()) / x.std()

def linear_regression(design_matrix, response):
    Q, R = np.linalg.qr(design_matrix, mode='reduced')
    return np.linalg.solve(R, np.dot(Q.T, response))
```

Use good variable/function names

```
import numpy as np

def better_function():
    response, design_matrix1, design_matrix2 = load_data(
        '/tmp/123.npy')
    coefficient1 = linear_regression(
        zscore(design_matrix1), response)
    coefficient2 = linear_regression(
        zscore(design_matrix2), response)
    coefficient_difference = coefficient1 - coefficient2
    np.save('ans.npy', coefficient_difference)

def load_data(data_name):
    X = np.load(data_name, mmap_mode='r')
    return X[:, 0], X[:, 1], X[:, 2]

def zscore(x):
    return (x - x.mean()) / x.std()

def linear_regression(design_matrix, response):
    Q, R = np.linalg.qr(design_matrix, mode='reduced')
    return np.linalg.solve(R, np.dot(Q.T, response))
```

You don’t need comments if the variable or function already tells you what it does (self-documenting)

Use the naming conventions of your language of choice (`snake_case` or `camelCase`) and be consistent

Avoid using abbreviations that are not commonly used

(`sw` vs. `spike_width`)

Prefer whole words

(`elec_poten` vs. `electric_potential`)

STEP 3. Document your functions

Document your functions

Easy thing: brief sentence describing the function without using the name of the function*

**this is the most important*

Document your functions

```
def zscore(x):
    return (x - x.mean()) / x.std()

def linear_regression(design_matrix, response):
    Q, R = np.linalg.qr(design_matrix, mode='reduced')
    return np.linalg.solve(R, np.dot(Q.T, response))
```

Document your functions

```
def zscore(x):
    '''Number of standard deviations from the mean'''
    return (x - x.mean()) / x.std()

def linear_regression(design_matrix, response):
    Q, R = np.linalg.qr(design_matrix, mode='reduced')
    return np.linalg.solve(R, np.dot(Q.T, response))
```

Document your functions

```
def zscore(x):
    '''Number of standard deviations from the mean'''
    return (x - x.mean()) / x.std()

def linear_regression(design_matrix, response):
    '''Calculate a linear least-squares regression for
    two sets of measurements'''
    Q, R = np.linalg.qr(design_matrix, mode='reduced')
    return np.linalg.solve(R, np.dot(Q.T, response))
```

Document your functions

- additional detail about what the function does or method it implements
- description of the parameters
- description of the outputs
- examples if you can

Document your functions

```python
def linear_regression(design_matrix, response):
    '''Calculate a linear least-squares regression for
    two sets of measurements

    Uses the QR decomposition to avoid numerical instability
    in taking the inverse.

    Parameters
    ----------
    design_matrix, response : array_like
        Two sets of measurements. Both arrays should have
        the same length.

    Returns
    -------
    coefficients : array_like
        Parameters estimated from the model.

    Examples
    --------
    >>> design_matrix = np.random.random(10)
    >>> response = np.random.random(10)
    >>> coefficients = linear_regression(design_matrix, response)

    '''
```

STEP 4. Test your code

Test your code

Make sure your code works like you think it does

Test your code

Think about how your code can fail

Small, well-defined, well-named functions are easy to test!

Test your code

```
import numpy as np

def zscore(x):
    '''Number of standard deviations from the mean'''
    return (x - x.mean()) / x.std()

def test_zscore():
    pass
```

Test your code

```python
import numpy as np

def zscore(x):
    '''Number of standard deviations from the mean'''
    return (x - x.mean()) / x.std()

def test_zscore():
    test_values = np.asarray([1, 3])
    expected_values = np.asarray([-1, 1])

    assert np.allclose(zscore(test_values), expected_values)
```

Test your code

Unit tests test a small component of your code (usually a small function) and makes sure it works like you think it works

Unit tests prevent regression of your code

If you change your code, you want to know what still works and what has broken (Regression)

Functions should be simple to test

If you find a bug, write a test.

Use unit tests to define the requirements of your code

You can use programs called test runners to run a group of unit tests automatically.

Matlab, Python, R have unit test packages

- **Matlab unit test framework**
- **Python unit test**
- **Pytest**
- **R: testthat**

Test your code

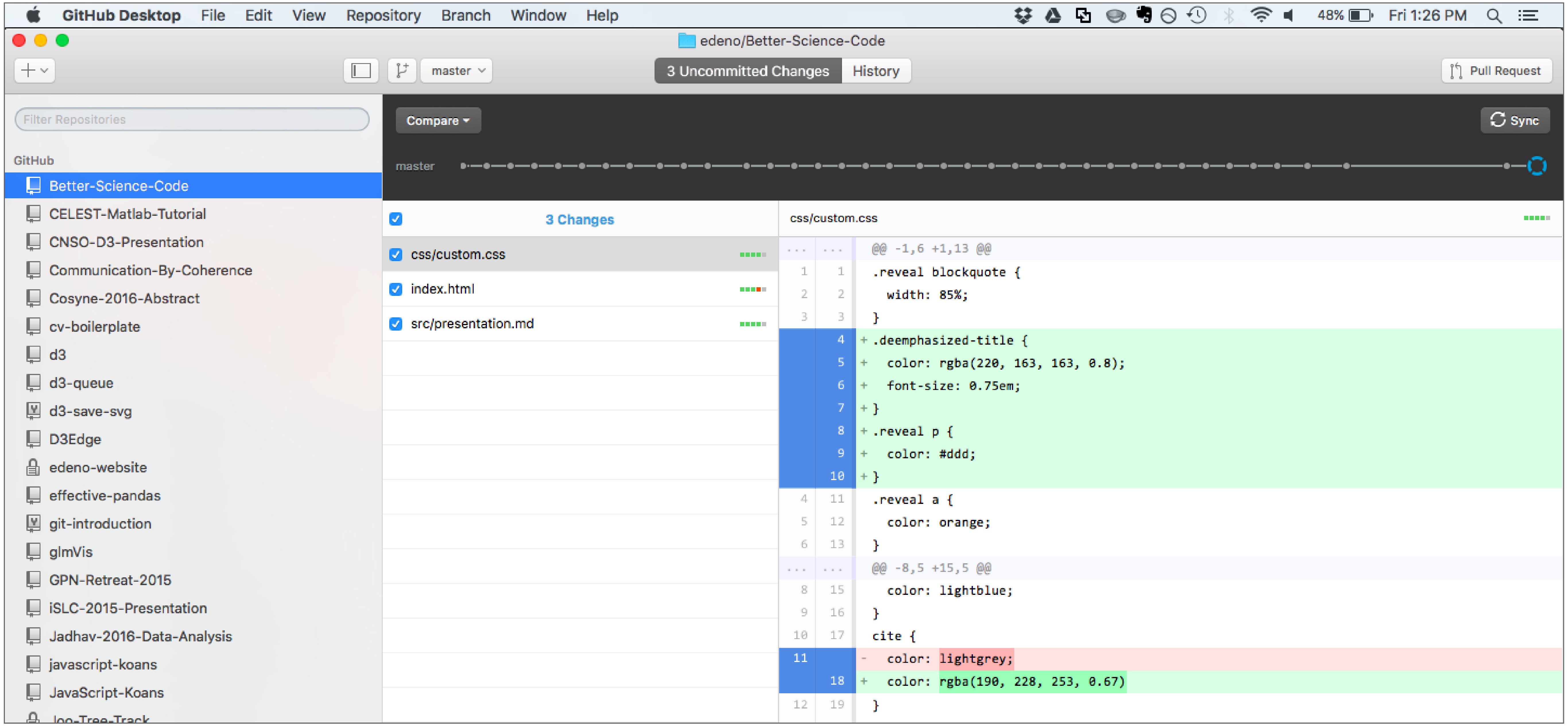
There are also libraries available that will work with your version control system to run these tests every time you commit a new piece of code (continuous integration)

STEP 5. Use version control

Use version control

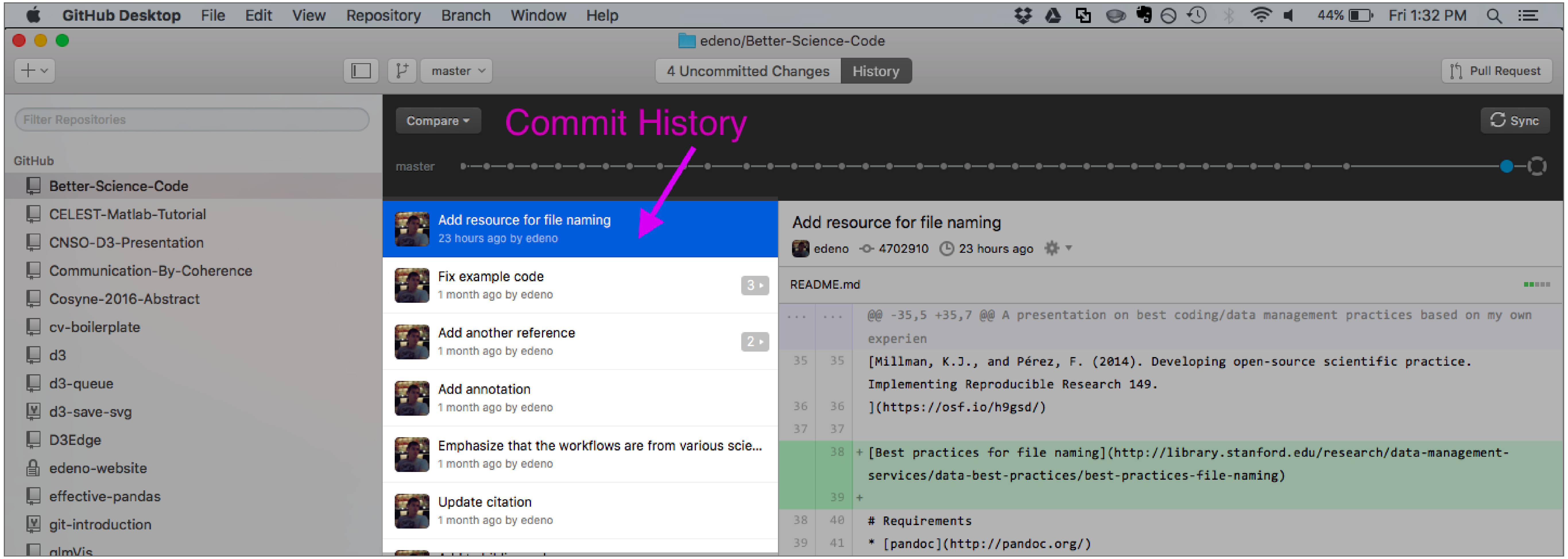
Sophisticated way to track change in your code over time

Use version control



Github Desktop

Version control stores the whole history of your project

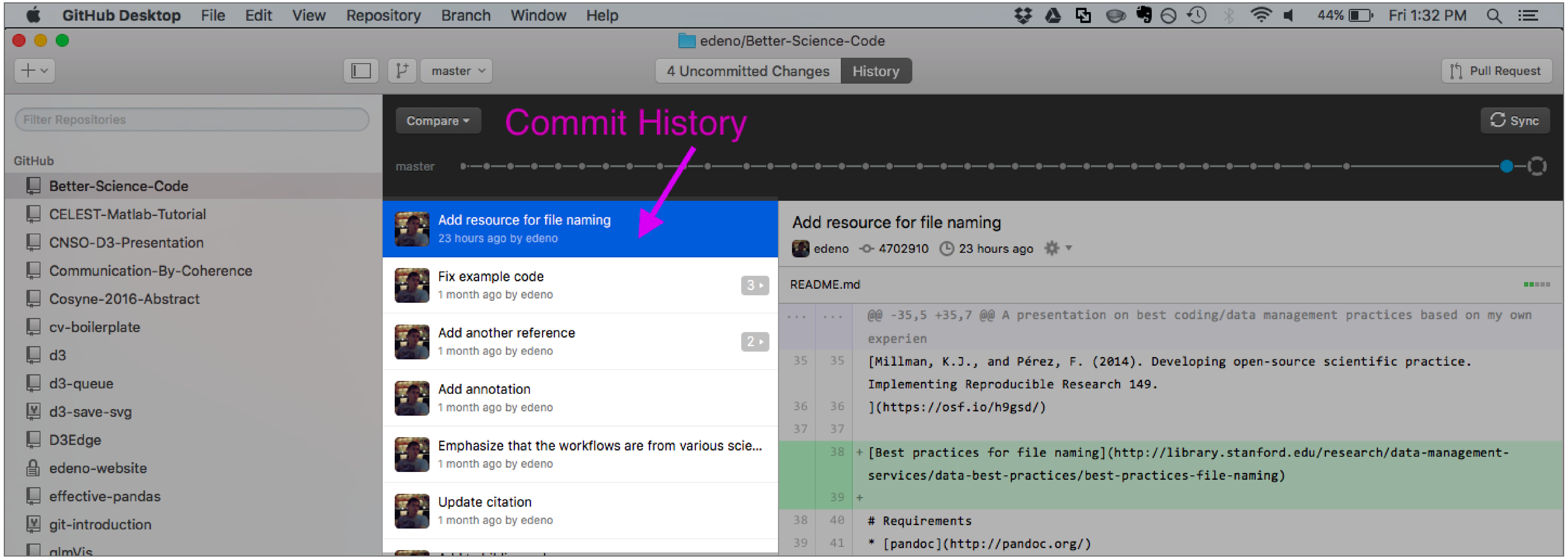

GitHub Desktop File Edit View Repository Branch Window Help
edeno/Better-Science-Code
master
4 Uncommitted Changes
History
Pull Request
Filter Repositories
Compare
Commit History
Sync
GitHub
Better-Science-Code
CELEST-Matlab-Tutorial
CNSO-D3-Presentation
Communication-By-Coherence
Cosyne-2016-Abstract
cv-boilerplate
d3
d3-queue
d3-save-svg
D3Edge
edeno-website
effective-pandas
git-introduction
Add resource for file naming
23 hours ago by edeno
Fix example code
1 month ago by edeno
Add another reference
1 month ago by edeno
Add annotation
1 month ago by edeno
Emphasize that the workflows are from various scie...
1 month ago by edeno
Update citation
1 month ago by edeno
Add resource for file naming
edeno 4702910 23 hours ago
README.md
@@ -35,5 +35,7 @@ A presentation on best coding/data management practices based on my own experien
[Millman, K.J., and Pérez, F. (2014). Developing open-source scientific practice. Implementing Reproducible Research 149.
](https://osf.io/h9gsd/)
+ [Best practices for file naming](http://library.stanford.edu/research/data-management-services/data-best-practices/best-practices-file-naming)
# Requirements
* [pandoc](http://pandoc.org/)

Use version control

Helps you back up your work

Use version control

Go back to previous versions of your code


GitHub Desktop File Edit View Repository Branch Window Help
44% Fri 1:32 PM
edeno/Better-Science-Code
master
4 Uncommitted Changes
History
Pull Request
Filter Repositories
Compare
Commit History
Sync
master
GitHub
Better-Science-Code
CELEST-Matlab-Tutorial
CNSO-D3-Presentation
Communication-By-Coherence
Cosyne-2016-Abstract
cv-boilerplate
d3
d3-queue
d3-save-svg
D3Edge
edeno-website
effective-pandas
git-introduction
Add resource for file naming
23 hours ago by edeno
Fix example code
1 month ago by edeno
Add another reference
1 month ago by edeno
Add annotation
1 month ago by edeno
Emphasize that the workflows are from various scie...
1 month ago by edeno
Update citation
1 month ago by edeno
Add resource for file naming
edeno 4702910 23 hours ago
README.md
@@ -35,5 +35,7 @@ A presentation on best coding/data management practices based on my own experien
[Millman, K.J., and Pérez, F. (2014). Developing open-source scientific practice. Implementing Reproducible Research 149.
](https://osf.io/h9gsd/)
+ [Best practices for file naming](http://library.stanford.edu/research/data-management-services/data-best-practices/best-practices-file-naming)
# Requirements
* [pandoc](http://pandoc.org/)

Use version control

Reduce code clutter and confusion

Use version control

Experiment with different versions of code (branches)

Use version control

Makes it easier to work with others

Use version control

Commit early and often

STEP 6. Refactor your code

“Whenever I have to think to understand what the code is doing, I ask myself if I can refactor the code to make that understanding more immediately apparent.” – Martin Fowler, Refactoring: Improving the Design of Existing Code

Refactor your code

Always leave the code in a better state than when you first found it.

STEP 7. Always search for well-maintained software libraries that do what you need.

Don’t rewrite functions that are already implemented as part of the core language.

Use other software libraries if they are well-maintained

How to write good code???

Exercise in managing complexity:

- break problems down into smaller components
- eliminate unnecessary dependencies
- keep track of what you did (be organized)

Summary:

1. Write small well-defined, well-named functions
2. Use good function and variable names
3. Document your functions
4. Test your code
5. Refactor your code
6. Use version control
7. Always search for well-maintained software libraries that do what you need.

break problems down into smaller components

1. Write small well-defined, well-named functions
2. Use good function and variable names
3. Document your functions
4. Test your code
5. Refactor your code
6. Use version control
7. Always search for well-maintained software libraries that do what you need.

keep track of what you did (be organized)

1. Write small well-defined, well-named functions
2. Use good function and variable names
3. Document your functions
4. Test your code
5. Refactor your code
6. Use version control
7. Always search for well-maintained software libraries that do what you need.

Conclusion: Writing good code takes work

We have a scientific obligation to ensure the correctness of our programs.

Put different projects in different folders/repositories

Use relative paths

Separate the data from the code

Processed Data should be separated from Raw Data to avoid accidentally changing the data

Tidy Data:

- Each variable forms a column.
- Each observation forms a row.
- Each type of observational unit forms a table
- flat is better than nested

If original data is not in a good form, convert it to a good form (but don’t overwrite the original data)

Don’t hand-edit data files.

All aspects of data cleaning should be in scripts

File naming:

- Don’t use spaces in file names
- Use leading zeros (001 vs. 1)