Why you should test your software

and how you can do it

MRI Together

Thursday 8th December 2022

Paul McCarthy
paul.mccarthy@ndcn.ox.ac.uk
pauldmccarthy@gmail.com

wellcome
centre
integrative
neuroimaging

Overview

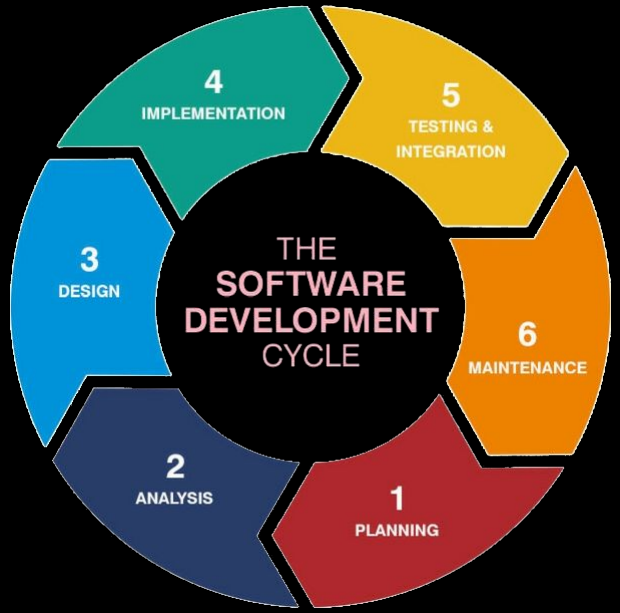
- Why testing is necessary
- Reducing the number of errors that you make when writing software
- Testing a Python application
- Automated testing using GitHub Actions

Science in 2022

Research results == software outputs?

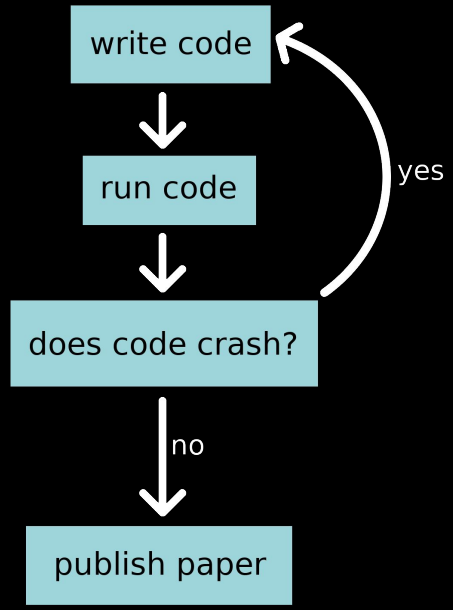
Software development practices

In industry



https://www.synotive.com/blog/software-development-client-questionnaire

In research



Compiled vs interpreted languages

Compiled/strongly-typed
(e.g. C/C++/Java)

Interpreted/dynamically typed
(e.g. MATLAB/Python)

Compiled vs interpreted languages

Compiled/strongly-typed
(e.g. C/C++/Java)

Interpreted/dynamically typed
(e.g. MATLAB/Python)

1. Write code
2. Compile code
3. Run code

Compiled vs interpreted languages

Compiled/strongly-typed
(e.g. C/C++/Java)

1. Write code
2. Compile code
3. Run code

Interpreted/dynamically typed
(e.g. MATLAB/Python)

Write code 1.

Run code 2.

Compiled vs interpreted languages

Compiled/strongly-typed
(e.g. C/C++/Java)

1. Write code
2. Compile code
3. Run code

The compiler analyses every single line of code, and catches lots of errors for you.

Interpreted/dynamically typed
(e.g. MATLAB/Python)

Write code 1.

Run code 2.

Compiled vs interpreted languages

Compiled/strongly-typed
(e.g. C/C++/Java)

1. Write code
2. Compile code
3. Run code

The compiler analyses every single line of code, and catches lots of errors for you.

Interpreted/dynamically typed
(e.g. MATLAB/Python)

Write code 1.

Run code 2.

The interpreter only analyses the code that gets run, so errors may still be present in infrequently used branches/sections of code.

Compiled vs interpreted languages

Compiled/strongly-typed
(e.g. C/C++/Java)

1. Write code
2. Compile code
3. Run code

The compiler analyses every single line of code, and catches lots of errors for you.

We can depend on the compiler to catch many errors for us.

(but it won't catch everything!)

Interpreted/dynamically typed
(e.g. MATLAB/Python)

Write code 1.

Run code 2.

The interpreter only analyses the code that gets run, so errors may still be present in infrequently used branches/sections of code.

Compiled vs interpreted languages

Compiled/strongly-typed
(e.g. C/C++/Java)

We can depend on the compiler to catch many errors for us.

(but it won't catch everything!)

1. Write code
2. Compile code
3. Run code

The compiler analyses every single line of code, and catches lots of errors for you.

Interpreted/dynamically typed
(e.g. MATLAB/Python)

We have no compiler, so need to be much more careful during development, and rely on other tools and techniques to catch errors.

1. Write code
2. Run code

The interpreter only analyses the code that gets run, so errors may still be present in infrequently used branches/sections of code.

It doesn't matter how smart you are – you ***will*** make mistakes

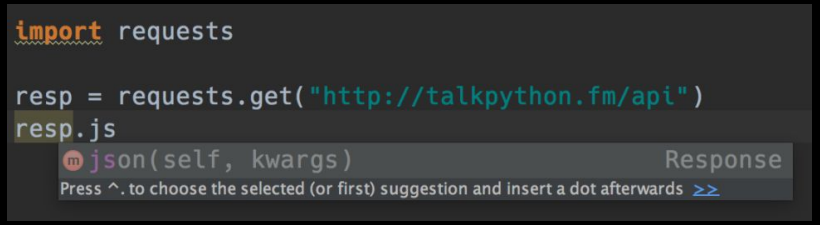
1. **Use a good editor/IDE***
2. **Organise your code sensibly**
3. **Write code to test your code**
4. **Use CI† to run your tests for you**

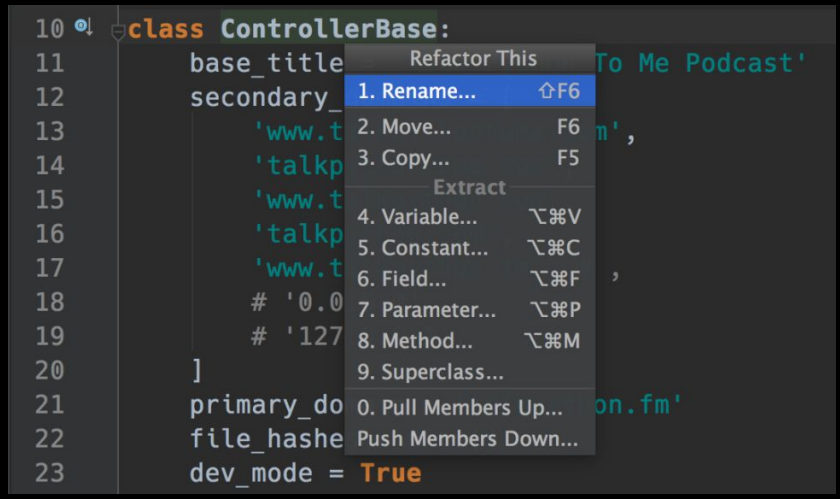
*IDE: Integrated Development Environment
†CI: Continuous Integration

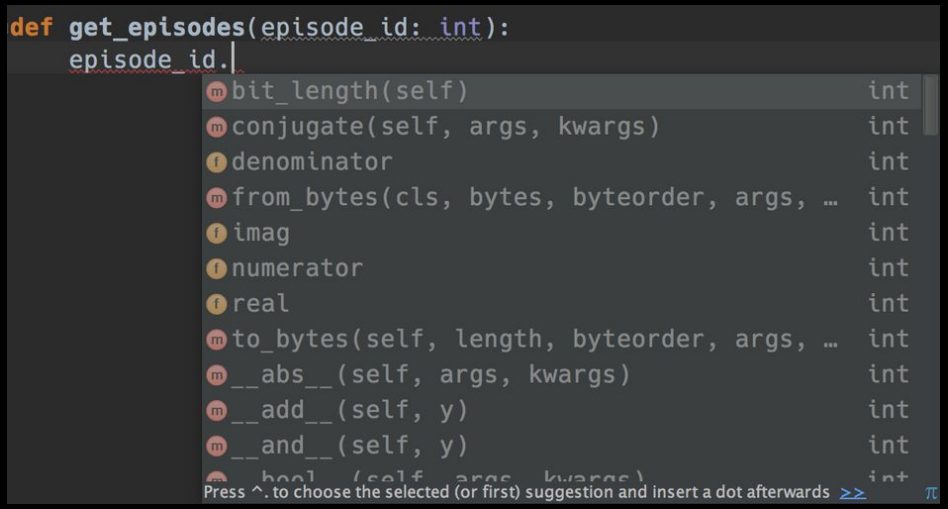
1. Use a good editor/IDE

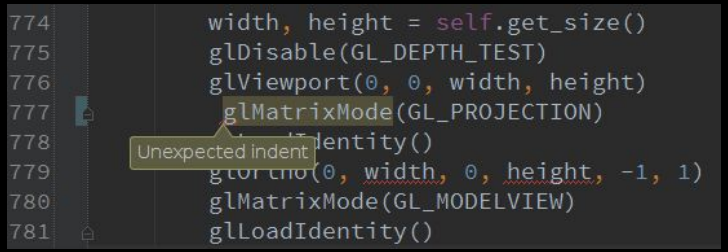
- If you're using MATLAB, feel free to zone out for the next couple of minutes
- If you're using Notepad, TextEdit, Gedit, vanilla Emacs, etc, then you're doing it wrong
- You should be using an editor which provides (at the very least):
  - Syntax highlighting
  - Intelligent error detection
- Optional (but recommended) extras which will make life easier
  - Automatic code completion
  - Refactoring (e.g. rename a function in every place it is called)
  - (important for larger projects) Code navigation/file management
  - Version control integration

1. Use a good editor/IDE









https://blog.michaelckennedy.net/2015/11/19/9-reasons-you-should-be-using-pycharm/

1. Use a good editor/IDE

- PyCharm (everything included)
- JupyterLab (MATLAB-like, browser-based)
- Spyder (MATLAB-like)
- VS Code (needs some setup)
- Atom (needs some setup)
- Sublime Text (non-free, needs some setup)
- Emacs (only for the masochistic)

2. Organise your code sensibly

```
#!/usr/bin/env python
import numpy as np

# load data and model
data = np.loadtxt('input.txt')
model = np.loadtxt('design.txt')

# perform OLS regression
fit = np.linalg.inv(model.T @ model) @ model.T @ data
error = data - (model @ fit)

# save parameter estimates and residuals
np.savetxt('pes.txt', fit)
np.savetxt('residuals.txt', error)
```

2. Organise your code sensibly

```
#!/usr/bin/env python
import numpy as np

# load data and model
data = np.loadtxt('input.txt')
model = np.loadtxt('design.txt')

# perform OLS regression
fit = np.linalg.inv(model.T @ model) @ model.T @ data
error = data - (model @ fit)

# save parameter estimates and residuals
np.savetxt('pes.txt', fit)
np.savetxt('residuals.txt', error)
```

```
#!/usr/bin/env python
import numpy as np
import sys

def ols(data, model):
    """Perform ordinary-least-squares regression. """
    fit   = np.linalg.inv(model.T @ model) @ model.T @ data
    error = data - (model @ fit)
    return fit, error

def main(args=None):
    """Fit a linear model to some data. """

    if args is None:
        args = sys.argv[1:]

    datafile   = args[0]
    designfile = args[1]
    fitfile    = args[2]
    errfile    = args[3]

    # load data and model
    data  = np.loadtxt(datafile)
    model = np.loadtxt(designfile)

    # perform OLS regression
    fit, error = ols(data, model)

    # save parameter estimates and residuals
    np.savetxt(fitfile, fit)
    np.savetxt(errfile, error)

if __name__ == '__main__':
    sys.exit(main())
```

2. Organise your code sensibly

```
#!/usr/bin/env python
import numpy as np

# load data and model
data = np.loadtxt('input.txt')
model = np.loadtxt('design.txt')

# perform OLS regression
fit = np.linalg.inv(model.T @ model) @ model.T @ data
error = data - (model @ fit)

# save parameter estimates and residuals
np.savetxt('pes.txt', fit)
np.savetxt('residuals.txt', error)
```

Harder to test

```
#!/usr/bin/env python
import numpy as np
import sys

def ols(data, model):
    """Perform ordinary-least-squares regression. """
    fit   = np.linalg.inv(model.T @ model) @ model.T @ data
    error = data - (model @ fit)
    return fit, error

def main(args=None):
    """Fit a linear model to some data. """

    if args is None:
        args = sys.argv[1:]

    datafile   = args[0]
    designfile = args[1]
    fitfile    = args[2]
    errfile    = args[3]

    # load data and model
    data  = np.loadtxt(datafile)
    model = np.loadtxt(designfile)

    # perform OLS regression
    fit, error = ols(data, model)

    # save parameter estimates and residuals
    np.savetxt(fitfile, fit)
    np.savetxt(errfile, error)

if __name__ == '__main__':
    sys.exit(main())
```

Easier to test

2. Organise your code sensibly

General advice

Wherever possible, strive to write functions which are:

- Small
- Simple
- Without side-effects

They will be easier to understand, easier to re-use, and easier to test.

2. Organise your code sensibly

General advice

Try to separate procedural operations (e.g. loading/saving data) from purely functional/numeric operations.

Doing so will make the critical parts of your code easier to test.

2. Organise your code sensibly

General advice

Use functions, classes, modules, and packages to arrange your project in a sensible manner.

2. Organise your code sensibly

General advice

There is no single answer to the question of how you should organise your code.

If it is easy to understand, navigate, and test (and it does its job), then it is a good design.

2. Organise your code sensibly

General advice

Don’t be afraid to refactor and rearrange your code as it evolves.

If you have a test suite*, then you can use it to ensure that you are not breaking things when you rearrange them.

**Integration tests*, introduced later, are handy here.

2. Organise your code sensibly

General advice

Don’t be afraid to refactor and rearrange your code as it evolves.

If you use version control, then you can experiment freely with your design, and easily revert back to a known good version if needed.

3. Write code to test your code

```
test_model = np.array([[0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1],
                       [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]]).T
test_data  = (test_model[:, 0] * 5) + (test_model[:, 1] * 20)
```

```
def ols(data, model):
    """Perform ordinary-least-squares regression. """
    fit   = np.linalg.inv(model.T @ model) @ model.T @ data
    error = data - (model @ fit)
    return fit, error

def main(args=None):
    """Fit a linear model to some data. """

    if args is None:
        args = sys.argv[1:]

    datafile   = args[0]
    designfile = args[1]
    fitfile    = args[2]
    errfile    = args[3]

    # load data and model
    data  = np.loadtxt(datafile)
    model = np.loadtxt(designfile)

    # perform OLS regression
    fit, error = ols(data, model)

    # save parameter estimates and residuals
    np.savetxt(fitfile, fit)
    np.savetxt(errfile, error)
```

3. Write code to test your code

```
test_model = np.array([[0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1],
                       [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]]).T
test_data  = (test_model[:, 0] * 5) + (test_model[:, 1] * 20)


def test_ols():
    fit, err = ols(test_data, test_model)
    assert np.isclose(fit, [5, 20]).all()
    assert np.isclose(err, 0)      .all()
```

```
def ols(data, model):
    """Perform ordinary-least-squares regression. """
    fit   = np.linalg.inv(model.T @ model) @ model.T @ data
    error = data - (model @ fit)
    return fit, error

def main(args=None):
    """Fit a linear model to some data. """

    if args is None:
        args = sys.argv[1:]

    datafile   = args[0]
    designfile = args[1]
    fitfile    = args[2]
    errfile    = args[3]

    # load data and model
    data  = np.loadtxt(datafile)
    model = np.loadtxt(designfile)

    # perform OLS regression
    fit, error = ols(data, model)

    # save parameter estimates and residuals
    np.savetxt(fitfile, fit)
    np.savetxt(errfile, error)
```

3. Write code to test your code

```
test_model = np.array([[0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1],
                       [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]]).T
test_data  = (test_model[:, 0] * 5) + (test_model[:, 1] * 20)


def test_ols():
    fit, err = ols(test_data, test_model)
    assert np.isclose(fit, [5, 20]).all()
    assert np.isclose(err, 0)      .all()


def test_main():
    np.savetxt('data.txt', test_data)
    np.savetxt('model.txt', test_model)

    main('data.txt', 'model.txt', 'pes.txt', 'residuals.txt')

    fit = np.loadtxt('pes.txt')
    err = np.loadtxt('residuals.txt')

    assert np.isclose(fit, [5, 20]).all()
    assert np.isclose(err, 0)      .all()
```

```
def ols(data, model):
    """Perform ordinary-least-squares regression. """
    fit   = np.linalg.inv(model.T @ model) @ model.T @ data
    error = data - (model @ fit)
    return fit, error

def main(args=None):
    """Fit a linear model to some data. """

    if args is None:
        args = sys.argv[1:]

    datafile   = args[0]
    designfile = args[1]
    fitfile    = args[2]
    errfile    = args[3]

    # load data and model
    data  = np.loadtxt(datafile)
    model = np.loadtxt(designfile)

    # perform OLS regression
    fit, error = ols(data, model)

    # save parameter estimates and residuals
    np.savetxt(fitfile, fit)
    np.savetxt(errfile, error)
```

3. Write code to test your code

Dependency injection is easy using `unittest.mock` (or similar)

```
def load_parameters(cfgfile):
    with open(cfgfile) as f:
        p1 = int(f.readline())
        p2 = int(f.readline())
        p3 = int(f.readline())
    return p1, p2, p3


def do_analysis(data):
    p1, p2, p3 = load_parameters('analysis_config.txt')
    return data * p1 * p2 / p3
```

https://docs.python.org/3/library/unittest.mock.html

3. Write code to test your code

Dependency injection is easy using `unittest.mock` (or similar)

```
        def load_parameters(cfgfile):
            with open(cfgfile) as f:
                p1 = int(f.readline())
                p2 = int(f.readline())
                p3 = int(f.readline())
            return p1, p2, p3


        def do_analysis(data):
            p1, p2, p3 = load_parameters('analysis_config.txt')
            return data * p1 * p2 / p3
```

```
from unittest import mock

def test_analysis():
    with mock.patch('mymodule.load_parameters', return_value=(10, 10, 10)):
        assert mymodule.do_analysis(10) == 100
```

https://docs.python.org/3/library/unittest.mock.html

3. Write code to test your code

and run that code regularly

`pytest` https://docs.pytest.org/en/latest/
searches through your code for functions beginning with `test_` and runs them for you.

`coverage` https://github.com/nedbat/coveragepy
tells you which lines of code were executed during testing

```
(brc) ➜  my_analysis pytest
==================== test session starts ====================
collected 2 items

tests/test_main.py::test_ols PASSED                   [ 50%]
tests/test_main.py::test_main PASSED                  [100%]

----------- coverage: platform linux, python 3.6.9-final-0 -
----------
Name                     Stmts   Miss  Cover
------------------------------------------
analysis/__init__.py         0      0   100%
analysis/main.py            18      5    72%
------------------------------------------
TOTAL                       18      5    72%


==================== 2 passed in 0.13s =====================
(brc) ➜  my_analysis
```

3. Write code to test your code

General advice

Write and run tests at the same time as you are writing the code.

Doing so will help you to better understand and trust your code.

3. Write code to test your code

General advice

Write tests for individual functions - these are known as *unit tests*.

Write tests for the entire program - these are known as *integration tests*.

3. Write code to test your code

General advice

Whenever you fix a bug in your code, write a test for it, so you'll know if it ever pops up again.

This is known as a *regression test*.

4. Use CI to run your tests for you

Continuous Integration (CI): Run code on somebody elses' computer using magic

1. Push some changes to your remote repository (e.g. GitHub)
2. GitHub notifies the CI provider about the changes
3. The CI provider starts up a Docker container or a VM running somewhere in the cloud
4. A script (which you write), running in the container/VM, checks out and builds your project, and runs your tests

4. Use CI to run your tests for you

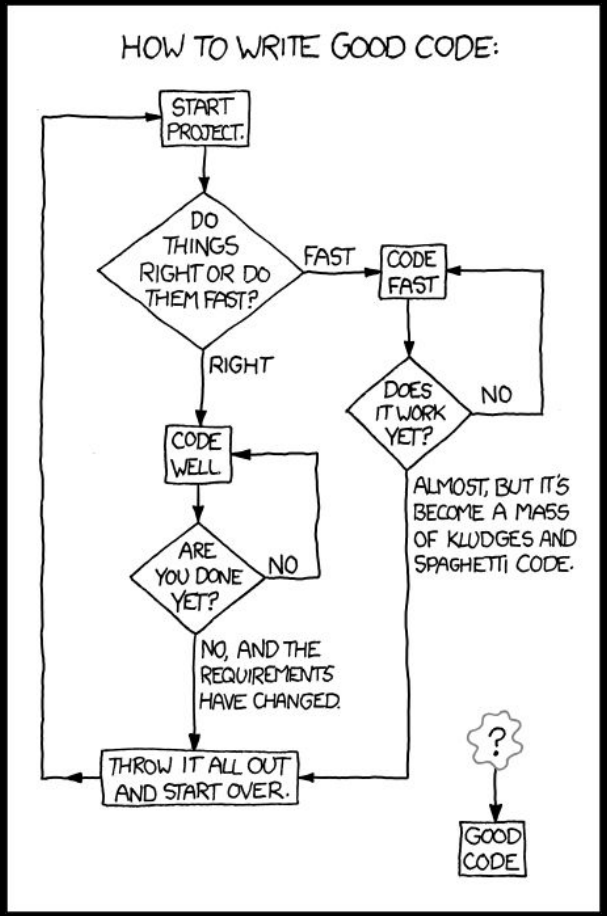
- Several CI providers work seamlessly with GitHub:
  - GitHub Actions (https://docs.github.com/en/actions)
  - CircleCI (https://circleci.com/)
  - Travis (https://travis-ci.org/)
  - Azure (https://azure.microsoft.com/en-us/services/devops/pipelines/)
  - Appveyor (https://www.appveyor.com/)
- Often free for open source projects
- If using GitLab, you can install your own CI provider (running on your own hardware)

4. Use CI to run your tests for you

Example project:

https://github.com/pauldmccarthy/mritogether-2022-software-testing

Thanks for listening!



https://xkcd.com/844/