Unit testing, Data carpentry/Environmenta informatics

What is unit testing?

A try at a definition

- An automated set of tests of your code
- You can run them any time and should do so regularly
- They help you design your code
- You write them at the same time or before your code (during design phase)
- You typically test your functions in a separate testing document or documents/code

The purpose

- Helps design good code
- Gives some confidence in your code
- Gives some documentation to the use of your code
- Can rerun as you augment your code to detect if your changes broke something

Unit testing is technically not testing

- It is part of the design/development phase
- Not really part of the (independent) testing phase of the final product in larger software projects
- Though that distinction is less important for scientific projects
- It helps move testing and thinking about testing up to the design stage

You probably already do a form of unit testing

- Here is what many scientific programmers do:
  - Write a function
  - Load it (i.e., source it in)
  - Experiment with it in the console to see if it works
- We want to automate this, and log it, basically, so it can be repeated as the code grows

Why automate unit testing?

- When you test on the command line it is informal
- You may not remember what tests you have run and passed
- You cannot repeat it in three months
- You may end up wasting time repeating testing

When to write unit tests, and (more on) what is their purpose?

When to write them

- First design your code up to function specs
- Then write unit tests
- Then write your code
- So you write your unit tests before your code!

Why write them so soon?

- One of the main advantages of unit tests is they help at the design phase
- For unit tests, you think about what the output of your function should be for simple cases
- This helps clarify the design

What is the purpose?

- So one of the purposes is helping with design
- Helps make design more precise
- You realize as you write the tests more about what the function needs to do
- It encourages you to think about simple cases, boundary cases, sub-cases

What else is the purpose?

- Gives some confidence in your code
  - You will get more confidence as you see your unit tests catching bugs!
- Gives some documentation to the use of your code
  - In three months, will you remember how things work?
  - Well, the unit tests gives example calls to your functions!
- Can rerun as you augment your code to detect if your changes broke something
  - When you make changes later, you just rerun your unit tests!

When else to write unit tests

- Whenever you find a BUG
- This is because the bug reveals another use case of the function that you did not correctly anticipate, and it should be recorded what that case is and that it works now

Structure of a unit test

Structure

- You set up a simple case where you know what the given function should produce
- You call the function for that case
- You compare the actual output with the expected output
- Preferably in an automated way, so you are alerted to the failure noticably if it occurs

What if an automated test is not possible?

- E.g., you know the output of a function that implements some sort of random process should be normally distributed
- If an automated check is not possible, write into your testing document the kind of result one expects, qualitatively, so it will be easier later on subsequent reruns to see whether that output is obtained
- E.g., write into your test doc that a normal distribution is expected from the following function call, then make the call and display the output so you can judge easily whether the test passed

What to test, and what it means about your design

What to test - well, pretty much just functions

- You can only readily test functions
- You `source` them into a separate testing script and test them there
- We will set it up so the separate script is an `Rmd` and you knit it to do your tests

What this means about your design

- This encourages you to use functions as much as reasonably possible, which tends to increase modularization
- You chunks will mostly call functions, or call functions and do bookkeeping

What to test - tips

- The test only depends on the interface, which goes together with the fact you can write tests when you only have the interfaces nailed down
- Try to test each kind of behavior of a function in one and only one way
- Focus on “fragile” code
- Always write a test when you discover a bug

A framework for unit tests

A framework for unit tests

- Imagine your main `.Rmd` file is just going to call functions you have written and "do bookkeeping" (i.e., organize results)
- Design things so this is possible
- Put your functions in separate scripts
- They are sourced and used by your main `.Rmd`
- They are sourced and tested by `Tests.Rmd`
- To perform your tests (or perform them again), just re-knit `Tests.Rmd`

When to reknit your tests document

- Regularly
- Perhaps every time you commit
- Or at least every time you commit non-trivial code changes

In-class exercise

The goal, generally

- See `GeneticData.txt`, open it up and have a look
- This is just a (made up) string of A, T, G, C's
- The task: find all microsatellites, here defined as
  - sequences of 2-6 bases (for our purposes)…
  - that repeat 30 or more times in immediate succession
- So if you see ATGATGATGATG…ATG (32 times) when ATG is the sequence and it repeats 32 times

The goal, specifically

- Find all microsatellites in `GeneticData.txt`
- For each one, work out start position and number of repeats involved
- Do this in our framework of unit testing
- Note: I have never directly analyzed genetic data in my life, so this simulates new research in the sense that the researcher starts out not knowing what he/she is doing

To follow along, clone this repo

- **https://github.com/reumandc/Exercise_PrincUnitTesting_Biol801Evrn4**
- First make sure your command line is located in a sensible spot to put this on your hard drive

Step 1 - starting point

- I have solved this problem in this git repo, but we want to see how I did it step by step
- I have made commits and created tags at various points so we can see the state of the repo at this points
- For the original state where only the data has been added and no code has been written, type `git checkout OriginalData`
- This goes back to the state then
- I created a “tag” called `OriginalData` that labels the repo state as it was then, and the `checkout` command goes back to what it was then
- You can open the `GeneticData.txt` with a text editor and check it out visually if you want

git checkout and tags

- You can also see tags on github
- Go the repository page: **https://github.com/reumandc/Exercise_PrincUnitTesting_Biol801Evrn4**
- Find the button that says “Branch: master”
- Under the dropdown menu attached to that button, click “tags”
- This gives a list of tags
- Click the one you want
- The files listed will change to whatever state they were in at the point in t revision history that tag is tagging
- You can go back to “branch: master” to get back to the latest version

Step 2 - file structure

- Next step is to add the file structure for our analysis and unit testing
- One file for the analysis (call it `GeneticExample.Rmd`) and another for testing (call it `GeneticExample_tests.Rmd`)
- To see the state of the repo after I got that set up, type `git checkout FileStructure`
- We will create functions in separate `.R` files which will be `source`'d and used on the data in `GeneticExample.Rmd` and `source`'d and tested in `GeneticExample_tests.Rmd`

Step 3 - top-most design

- One must now do (sometimes extensive) thinking about how to arrange the code (design step)
- It may help to start by doing this in English
- type `git checkout TopMostDesign` to see the results of that process - you can see how I was thinking about separating tasks and creating different functions to perform those tasks in a top-down way

Step 4 - next-level design

- Further breaking down the functions established in top-most design into functions they will, in turn, call
- type `git checkout NextLevelDesign` to see this
- I decided another function would be necessary

Step 5 - function specs

- Having performed some high-level design thinking, as can now easily write function specs
- type `git checkout FunctionSpecs` to see those
- New `.R` files should appear
- Recall that under our framework for unit testing (and modularity) functions are in separate `.R` files so they can be sourced by both `GeneticExample.Rmd` and by `GeneticExample_tests.Rmd`

Step 6 - UNIT TESTING

- Now is when we do unit testing! It will help us think very carefully about whether our specs are good ones
- May lead to design modifications
- It would be OK to write the tests for one function and then write that function, and then carry out the tests, and then repeat for the other functions one at a time, instead of writing all the unit tests for all the functions before writing any of the functions, as I have done
- If you do it that way, you may want to start with the "lowest-level" functions since they are the ones you can call (and therefore test) without having written the other functions yet
- To see the unit tests, type `git checkout UnitTests`

Step 7 - write the functions

- In the process of writing the functions, I made some realizations
- For instance, the function `dec_to_baseb` was needed to support `getseqs`, and so unit tests were also added for this new function
- I discovered the unit test of `find_microsats` was actually itself slightly wrong, so I fixed
- This kind of back and forth between unit tests and function writing is to be expected - it's part of the reason for unit testing!
- To see the functions at this stage, type `git checkout FunctionsImplemented`

Step 8 - now apply the functions to the original problem

- Done.
- Type `git checkout AppliedToData`