Administration

- Assignment 3 is out.
- We've got the assignment 2 autograder and are working on grading it.
Testing.

- Testing is key to developing good software.
- Tests should be easy to access, and easy to reuse.
- They should be separate from the code you are testing.
  - I.e. when you test code, you should test it based on the docstrings, not based on the actual code that you're writing.
  - This means that you can test what the user will see.
Testing.

- Two main aspects of testing:
  - Figuring out what you're going to test.
  - The tools used to run tests.
- This first aspect is largely language independent.
  - Not entirely, because some languages might allow different types of input.
- The second less so.
  - There's a reasonably common paradigm for how to test code, but some languages have built-in tools for it.
Test cases.

- What do we test?
- We can't test all inputs.
- So we need to choose a subset that is representative.
  - We can have 'typical inputs'.
  - We can test things where we might suspect programmer error.
  - We can test 'boundary conditions' that we suspect might have been overlooked.
Test cases.

• It is useful to think 'adversarially' when picking test cases.

  • That is, try to picture yourself as an adversary trying to break a program.

  • But do so without cheating, so if the docstring specifies some kind of input, limit yourself to those inputs.

  • But within those inputs try and choose as bad inputs as you can.
Test cases.

- We want all of our test cases to be independent.
- That is, we want to be certain of the source of a failure.
- So having lots of test cases that build on each other is not a great idea.
- Note that this is different from having test cases that test functions that build on other functions.
What do we test?

- Ideally one tests each function individually.
  - This is called unit testing.
- Once all the smaller functions have been tested, then you test the larger functions that call the smaller ones.
- When you make any changes, you want to run all the tests again.
  - This is called regression testing.
When do we test?

- It is best if you test a function right after writing it.
  - It is easiest to fix things at this point.
- Often it is useful to come up with test cases before you actually write any code.
  - This means that you think of the structure of the program and what each function does before you write the code.
  - This means that you can really write the tests in a black-box fashion, because you don't know what the code will be yet.
When do we test?

- Professional coders often write test cases before writing code.
- Thinking about tests cases while designing is also a useful design tool, because it can inform your design.
- Makes for more robust code.
Testing Summary.

- Want individual Unit tests.
  - These should be independent of each other.
  - There should be some generic ones, and some chosen 'adversarially'.
- Want to design tests before writing code.
  - Makes for more robust code and better style tests.
- Want to rerun tests when we change code.
- How do we do all this?
Break, the first.
Testing in Python.

• So we have a lot of constraints in Python testing.
• And it's hard to satisfy all of them.
• Thus far we've been testing in shell, and it's a lot of work to do regression testing that way.
• We could store all of our old tests in a file, but then we have to write specific code for opening files and dealing with them.
• Luckily python has a module called Nose that helps us with a lot of these things.
Testing with Nose.

• The context for testing with Nose is that we have a module named \texttt{mod}.
• We want to test some or all of the functions in it.
• To do this we create a module called \texttt{test__mod}.
• In this module we \texttt{import nose} and we \texttt{import mod}.
• For each function \texttt{func} we want to test, we have a \texttt{test__func()} function.
Testing with Nose.

- We have:
  ```python
  if __name__ == '__main__':
      nose.runmodule()
  ```
- In the body of `test__func()` we have assert statements.
  - `assert (boolean condition)` will do nothing if the condition is true, but will throw an error if it's false.
  - So `test__func()` has a bunch of statements like:
    ```python
    assert func(input) == (expected_output)
    ```
- Nose runs these and produces output.
Nose Output.

- The first line of output tells us the result of the tests.
  - a dot means pass, an F means fail, an E means an error.
  - So, a failure is incorrect output, an error is an exception of some kind.
  - Each failure or error produces information about that failure or error.
  - The last bit tells us the number of tests passes, the number of tests failed, and the number of errors.
Nose Output

- The information about the errors so far is just the error information that python gives back to us.
- If we fail a test we can an 'AssertionError'.
- If we want to add some information to this, we can put in a string after a comma in the assert statement.

assert (condition), "Some String."
Testing Summary.

- Want individual Unit tests.
  - These should be independent of each other.
  - There should be some generic ones, and some chosen 'adversarially'.
- Want to design tests before writing code.
  - Makes for more robust code and better style tests.
- Want to rerun tests when we change code.
- How does Nose do this?
Nose and Testing.

● Unit Tests.
  • Each test in nose is its own function, so we can write a function for each unit test we want.

● Designing Tests Early.
  • All we need to write test in nose is the specification for the function.
  • The tests treat functions as a black box.

● Regression Testing.
  • Nose makes it quite easy to run all the tests we have whenever we want.
So you have an error.

- If you find an error, you need to debug it, a process that is often painful.
- There are a few ways to mitigate this pain.
  - Test early! Test Often.
  - Find the first point that the code differs from what you think it would be.
  - Run through the code in your head to make sure that if everything goes the way you think, the code will work.
  - Read the error information, and use it to see if the code is correct at the point of the error.
Break, the second.
Assignment 2 Solution.

- Conceptually is in several parts.
- The functions to and from algebraic notation are their own parts.
- `game_summary` and `strip_tag_info` are their own part.
- `strip_tag_info` is the function that `game_summary` calls to do its work.
Assignment 2 solution.

- For the rest, the problem of recording the moves and boards state are intertwined. But it's two big of a problem to do on it's own, so we have to break it down into component chunks.
- Chunk one: parsing the input into little moves.
  - Done by get_move_lst and get_move_str.
Assignment 2 solution

- For each individual move text, we need to extract a bunch of possible bit of information from this.
  - Done with a bunch of functions: check, mate, get_fin_sq, get_piece_type, etc.
- For each move though, this isn't enough. We also need to keep track of the board, and update it accordingly.
  - For this we have update_board(), get_init_move, move_piece.
Assignment 2 solution

• Other things:
  • We keep the board as a nested list, in the format that we're supposed to return it.
  • When trying to find where a piece started, rather than looking at all possible place a piece could move to; we look at all possible pieces that could move to a place and try and find which one could do so legally.
  • This code relies very heavily on the correctness of the file.
Assignment 3 Comments and Questions.