Week 12

Announcements

- Basic and Advanced 9 due Monday, November 22
- Basic and Advanced 10 due Wednesday, December 1
- Basic 7 due Wednesday, December 8
- Basic and Advanced 11 due Wednesday, December 8
- Please fill out teaching evaluations when they're out!
  - I take these very seriously and they help the class evolve for future semesters
- Last main content lecture!
  - Friday, Dec 3 will be the last lecture day, featuring our IAs
  - No lecture Friday, Dec 10

Announcements

- GitLab assignment rerun
  - On December 20 all GitLab assignments (basic and advanced) will be rerun from a clean state
  - All current penalties will be cleared, but **can be reapplied if your submission still has issues**
  - You can use this as an opportunity to do older assignments that you were not able to complete

Lecture 12: Testing

Overview

- What is and why testing?
- Kinds of testing
- Unit testing
- Test-driven development

Foreword

- We won't go into the finer details about software testing
  - It's very deep and evolving topic
- If you want to know more, try taking EECS 481: Software Engineering

What is and why testing?

- Wikipedia: "Software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test"
  - Broad definition: includes checking for correctness, quality of service, etc.
  - We'll focus on the correctness *checking*
  - *Checking* to see if the right outputs are produced for the given inputs
- Testing **does not necessarily** guarantee or prove correctness
- Testing helps give confidence that the implementation follows specifications and **helps uncover bugs/defects**
  - Failing tests tells us something's broken
  - Passing tests tells us our code should work as far as tests go
  - It's still up to us to design good tests

Some kinds of testing

Hierarchy

- **Unit testing**: testing a **unit**: individual component of code e.g. function, class, etc.
- **Integration testing**: testing the interactions between components/subsystems
  - The line between integration and unit testing gets hazy when a class depends on another class...
- **System testing**: testing your final application

Other terms

- **Regression testing**: testing to see if anything old breaks from new changes
- **White-box testing**: testing that is aware of internals of the component being tested
- **Black-box testing**: testing that is blind to the internals of the component being tested

Unit testing

- Testing of individual **units**: individual component of code such as a function or class
- Write test cases that follow along with the specification
- By keeping the scope small, we can more easily locate bugs when a test fails
- Test cases provide inputs and check outputs for the particular unit
- Test cases should be independent of each other: they should not keep state between tests
- Test cases tend to have a typical structure:
  - **Setup**: sets up the "unit under test" (UUT) and its inputs
  - **Execution**: runs the UUT
  - **Validation**: checks to see if the outputs/behavior of the UUT is correct
  - **Cleanup**: restore the test system to a clean state

Unit testing

```
import unittest
class Foo:
  def __init__(self, name):
    self.name = name

  def bar(self, num):
    return self.name + str(num)

class TestFoo(unittest.TestCase):
  def test_bar(self):
    uut = Foo('test')
    out = uut.bar(42)
    self.assertEqual('test 42', out)

if __name__ == '__main__':
  unittest.main()
```

Does `Foo.bar()` work?

Unit testing frameworks

- Most languages have some sort of framework to test in
- Provides an environment to generate a special executable to run tests
- Many are based off of the xUnit paradigm influenced by Kent Beck's SUnit
  - "S" for "Smalltalk"
- Examples
  - Java: JUnit
  - Python: `unittest`
  - C/C++: Google Test

Test-driven development

- Development process where you **turn specifications for new features into tests** *before* you code

1. Add tests for new feature
2. Run tests, new tests should fail
3. Write the minimum code to pass the new tests
4. Run tests, they should pass
5. Refactor the code while passing tests
6. Repeat for new features

- In simple terms: add tests, write code to pass tests, make your code nicer, repeat

Test-driven development

- This process allows you to have some confidence that your code works
  - By minimizing your implementation you allow fewer avenues for things to go wrong
- More tests != better testing
- This process can tunnel-vision on small, simple tests; can fail to see bigger picture
- More tests = more maintenance
- Tests can take time to write: development may seem slower
  - Countered by time saved when debugging

Live TDD + unit testing demo feat. Python

`unittest`

Feel free to follow along!

Reverse polish notation calculator

- Infix notation: `(5 - 3) * (1 + 2)`
  - Binary operators in between operands
- Polish notation (PN): `* - 5 3 + 1 2`
  - Also known as "prefix notation"
  - Binary operators **before** ("pre") operands
  - Abstract syntax trees ;)
- **Reverse polish notation (RPN)**: `5 3 - 1 2 + *`
  - Also known as "postfix notation"
  - Binary operators **after** ("post") operands

Reverse polish notation calculator

- RPN lends itself to being implemented as a "stack machine"
  - Numbers get pushed onto the stack
  - Operators pop numbers off the stack and push the result
- Example: `5 3 - 1 2 + *`
  - `push 5`
  - `push 3`
  - `-`: `pop 3`, `pop 5`, perform `5-3=2`, `push 2`
  - `push 1`
  - `push 2`
  - `+`: `pop 2`, `pop 1`, perform `1+2=3`, `push 3`
  - `*`: `pop 3`, `pop 2`, perform `2*3=6`, `push 6`

Before we start

- Starter files

```
https://www.eecs.umich.edu/courses/eecs201/
        |-- files/examples/tdd/rpn.py
        |-- files/examples/tdd/test_rpn.py
```

- Python 3 unittest documentation
- Let's create a Makefile to run the tests

```
test:
    python3 -m unittest test_rpn
```

Spec

- Let's start simple then add more features to illustrate TDD
- Implement a read-evaluate-print-loop (REPL) to get input from user (Done)
- Implement a Calculator class that encapsulates the stack
  - Numbers are all floating point
  - `size()` function to return size of the stack
  - `result()` function to return top of stack
  - `input()` function to pass in commands, returns top of stack

Spec

Features

1. Handles numbers by pushing them on the stack
2. Handles addition
3. Handles unsupported operator
4. Handles not enough operands case
5. Handles subtraction
6. Handles multiplication
7. Handles division

Refactoring

- Maybe we can go back and clean things up a bit
- Are we repeating ourselves? How can we make this nicer?

Conclusion

- Testing isn't a panacea: tests are only as intelligent as their designers
- Unit tests can tell you something is wrong and what unit is failing
- TDD is a solid methodology, just beware of shortcomings
- This lecture's assignment will look at C/C++ and Google Test :)
  - You may find unit testing to be helpful in your 281 or 370 endeavors ;)

Additional resources

- EECS 481 slides about testing
  - One of the sources I drew upon: goes deeper than I did here
  - If that sort of stuff interests you, I encourage that you take the class
- Kent Beck's original work on Smalltalk testing
- Test Driven Development: By Example, Kent Beck
  - Free O'Reilly access provided by UMich (kudos to Arav)

Questions?