EdX 6.00x Notes

Lecture 7:

- Testing and Debugging
    - Testing Methods
        - Ways of trying code on examples to determine if running correctly
    - Debugging methods
        - Ways of fixing a program that you know does not work as intended
- When should you test and debug?
    - Design your code for ease f testing and debugging
        - Break program into components that can be tested and debugged independently
        - Document constraints on modules
            - Expectations on inputs, on outputs
            - Even if code does not enforce constraints, valuable for debugging to have description
        - Document assumptions behind code design
- When are you ready to test?
    - Ensure that code will actually run
        - Remove syntax errors
        - Remove static semantic errors
        - Both of these are typically handled by Python interpreter
    - Have a set of expected results (i.e. input-output pairings) ready
- Testing
    - Goal:
        - Show that bugs exist
        - Would be great to prove code is bug free bug generally hard
            - Usually can't run all possible inputs to check
            - Formal methods sometimes help, but usually on simpler code
- Test suite
    - Want to find a collection of inputs that has high likelihood of revealing bugs, yet is efficient
        - Partition space of inputs into subsets that provide equivalent information about correctness
            - Partition divides a set into group of subsets such that each element of set is in exactly one subset
        - Construct test suite that contains one input from each element of partition.
- Partitioning
    - Use natural partitions for input space.

    - If natural partitions do not exist:
        - Random testing:
            - Probability that code is correct increases with number of trials; but should be able to use code to do better
        - Black-Box testing:
            - Use heuristics based on exploring paths through the specifications
        - Glass-Box testing:
            - Use heuristics based on exploring paths through the code.
- Black-box testing
    - Test suite designed without looking at code
        - Can be done by someone other than implementer
        - Will avoid inherent biases of implementer, exposing potential bugs more easily
        - Testing designed without knowledge of implementation, thus can be refused even if implementation changed
- Note: If you find a bug the problem can be with the code or with the spec.
- Paths through a specification:
    - Also good to consider boundary cases
        - For lists: empty list, singleton list, many elements in a list
        - For numbers: very small, very large, “typical”
- Glass-box testing
    - Use code directly to guide design of test cases
    - Glass-box test suite is path-complete if ever potential path through the code is testing at least once
        - Not always possible if loop can be exercised arbitrary times, or recursion can be arbitrarily deep
    - Even path-complete suite can miss a bug, depending on choice of examples. Check for boundary cases.
- Rules of thumb for glass-box testing
    - Exercise both branches of all if statements
    - Ensure each expect clause is executed
    - For each for loop, have tests where:
        - Loop is not entered
        - Body of loop executed exactly once
        - Body of loop executed more than once
    - For each while loop,
        - Same cases as for loops
        - Cases that catch all ways to exit loop
    - For recursive functions, test with no recursive calls, one recursive call, and more than one recursive call
- Conducting tests
    - Start with unit testing

        - Check that each module (e.g. function) works correctly
        - Checks for algorithm bugs
    - Move to integration testing
        - Check that system as a whole works correctly
        - Checks for iteration bugs (bugs that an incorrect value is being communicated to another function)
    - Cycle between these phases
- Test Drivers and Stubs
    - Drivers are code that:
        - Set up environment needed to run code
        - Invoke code on predefined sequence of inputs
        - Save results and report
    - Drivers simulate parts of program that use unit being tested
    - Stubs simulate parts of program used by unit being tested
        - Allow you to test units that depend on software not yet written
- Good testing practice
    - Start with unit testing
    - Move to integration testing
    - After code is corrected, be sure to do **regression testing**:
        - Check that program still passes all the tests it used to pass, i.e., that your code fix hasn't broken something that used to work
- Runtime bugs
    - **Overt vs. covert:**
        - **Overt** has an obvious manifestation – code crashes or runs forever
        - **Covert** has no obvious manifestation – code returns a value, which may be incorrect but hard to determine
    - **Persistent vs. intermittent:**
        - **Persistent** occurs every time code is run
        - **Intermittent** only occurs some times, even if run on same input
- Categories of bugs
    - Overt and persistent
        - Obvious to detect
        - Good programmers use **defensive programming** to try to ensure that if error is made, bug will fall into this category
    - Overt and intermittent
        - More frustrating, can be harder to debug, but if conditions that prompt bug can be reproduced, can be handled
    - Covert
        - Highly dangerous, as users may not realize answers are incorrect until code has been run for long period
- Debugging skills

- Treat as a search problem: looking for explanation for incorrect behavior
  - Study available data – both correct test cases and incorrect ones
  - Form an hypothesis consistent with the data
  - Design a run a repeatable experiment with potential to refuse the hypothesis
  - Keep record of experiments performed: use narrow range of hypotheses
- Debugging as a search
  - Want to narrow down space of possible sources of error
  - Design experiments that expose intermediate stages of computation (use print statements!), and use results to further narrow search
  - Binary search can be a powerful tool for this
    - Pick a spot about halfway through code, and devise experiment
      - Pick a spot where easy to examine intermediate values and add a print statement
- Aliasing Bug:
  - A bug that occurs because you accidentally alias an object instead of pointing to a copy of an object
- Some pragmatic hints
  - Look for the usual suspects:
    - Do I have a boundary condition case?
    - Am I passing in the wrong argument?
    - Am I reversing the order of arguments?
    - Have I forgotten to call a method?
    - Do I actually invoke it rather than just accessing it?
  - Ask why the code is doing what it is, not why it is not doing what you want
  - The bug is probably not where you think it is – eliminate locations
  - Explain the problem to someone else
  - Don't believe the documentation
  - Take a break, take a walk, come back later