**MIT – 6.00.1x: Introduction to Computer Science and Programming**

**WEEK 4**

**Lecture 7: Debugging**

Part 1: Testing and Debugging

* Testing and Debugging
  + Would be great if our code always worked properly the first time we ran it!
  + But our code isn’t perfect, so we need:
    - Testing methods
* Ways of trying code on examples to determine if it is 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 of testing and debugging.
    - Break program into components that can be tested and debugged independently.
    - Document constraints on modules and good documentation in general can ease the process.
* Expectations on inputs and on outputs.
* Even if the code does not enforce constraints, documentation can be valuable for debugging to have description.
  + - Document assumptions behind all written code.
* When Are You Ready to Test?
  + Ensure that the code will actually run to begin with.
    - Remove syntax errors.
    - Remove static semantic errors.
    - Both of these are typically handled by Python interpreter.
  + It is good to have a set of expected results (i.e. input-output pairings) ready.

Part 2: Test Suites

* Testing
  + Goal:
    - To show that bugs exist in code.
    - Would be great to prove that code is bug free, but is generally hard to do.
* Usually can’t run test cases on all possible inputs and check that there are no bugs.
* Formal mathematical methods sometimes help prove bug-free code, but usually only work when testing simpler code.
* Test Suite
  + We want to find a collection of inputs that has a high likelihood of revealing bugs, yet is efficient at doing so.
    - Partition the total 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 exactly in one subset.
  + - Construct a test suite that contains one input from each element of that partition.
    - Run the test suite.
* Example of Partition

def isBigger(x, y):

‘‘‘

assumes x and y are integers

returns True if x is less than y

returns False otherwise

’’’

* + Input space is all pairs of integers.
  + Possible partition:
    - x positive, y positive
    - x negative, y negative
    - x positive, y negative
    - x negative, y positive
    - x = 0, y = 0
    - x = 0, y != 0
    - x != 0, y = 0
* Why This Partition?
  + Lots of other choices
    - E.g., x prime, y not prime; y prime, x not prime; both prime; both not prime
    - However, this division of input space is irrelevant to the problem.
  + Spaces of inputs often have natural boundaries.
    - Integers are either positive, negative, or 0.
    - From this perspective, we should have 9 subsets.
* Split x = 0, y != 0 into x = 0, y positive and x = 0, y negative
* Partitioning
  + What if there is no natural partition to the input space?
    - Random testing – probability that code is correct increases with number of trials; but should be able to use code to do better.
    - Use heuristics based on exploring paths through the specifications – **black-box testing**.
    - Use heuristics based on exploring paths through the code – **glass-box testing**.

Part 3: Black-Box Testing

* Black-Box Testing
  + Test suite is designed without actually looking at the code itself, just the specifications of all the functions and methods used.
    - Can be done by someone other than implementer.
    - Will avoid inherent biases of implementer, exposing potential bugs more easily.
    - Testing is designed without knowledge of implementation, and thus can be reused even if implementation changes.
* Paths Through a Specification

def sqrt(x, eps):

‘‘‘

assumes x and eps are floats

x >= 0

eps > 0

return result such that

x – eps < result \* result < x + eps

’’’

* + Paths through the specification:
    - x = 0
    - x > 0
  + But clearly not enough divide entire input space.
  + Also good to consider boundary cases.
    - For lists, boundary cases are: empty list, singleton list, many element list.
    - For numbers, boundary cases include: very small, very large, and ‘typical’ numbers.
* Example
  + For our square root example, we can try these test cases:

|  |  |
| --- | --- |
| *x* | *eps* |
| 0.0 | 0.0001 |
| 25.0 | 0.0001 |
| 0.05 | 0.0001 |
| 2.0 | 0.0001 |
| 2.0 | 1.0 / (2.0 \*\* 64.0) |
| 1.0 / (2.0 \*\* 64.0) | 1.0 / (2.0 \*\* 64.0) |
| 2.0 \*\* 64.0 | 1.0 / (2.0 \*\* 64.0) |
| 1.0 / (2.0 \*\* 64.0) | 2.0 \*\* 64.0 |
| 2.0 \*\* 64.0 | 2.0 \*\* 64.0 |

* + First four test cases are typical
    - Perfect square
    - Irrational square root
    - Example less than 1
  + Last five extreme cases
    - If there is a bug, it might be in the code, or might be in the specifications (e.g. don’t try to find square root is *eps* is really tiny.

Part 4: Glass-Box Testing

* Glass-Box Testing
  + Use code directly to guide design of test cases. The name sort of suggests this, as glass-box testing suggests that I can see inside to find out what is going on in the code.
  + A glass-box test suite is **path-complete** if every potential path through the code is tested at least once.
    - Not always possible if loop can be exercised an arbitrary number of times, or with recursion, where the recursive call can be arbitrarily deep, up to a certain limit.
  + Even path-complete test suite can miss a bug, depending on the choice of inputs taken.
* Example

def abs(x):

‘‘‘

assumes x is an integer

returns x if x >= 0 and –x if x < 0

’’’

if x < –1:

return –x

else:

return x

* + Test suite of {-2, 2} will be path-complete.
    - x = –2: returns –x 🡪 returns 2
    - x = 2: returns x 🡪 returns 2
  + But will miss abs(–1), which incorrectly returns –1.
    - Testing boundary cases and typical cases would catch this {–2, –1, 2}.
  + Thus, it’s not just about testing a path-complete test, but is also about testing the boundary cases at conditions to see if they are met properly.
* Rules of Thumb for Glass-Box Testing
  + Exercise all branches of all if statements.
  + Ensure that each except clause is executed.
  + For each for loop, have tests where:
    - Loop is not entered.
    - Body of the loop is executed exactly once.
    - Body of the loop is executed more than once.
  + For each while loop, have tests where:
    - Same test cases as for loops.
    - Cases that catch all the ways to exit the loop.
  + For recursive functions, have the following tests:
    - Test with no recursive call.
    - One recursive call.
    - More than one recursive call.

Part 5: Test Drivers and Stubs

* Conducting Tests
  + Start with **unit testing**
    - Check that each module (e.g. function) works correctly. This will catch algorithm bugs, either in how the algorithm was implemented or actually designed.
  + Move to **integration testing**
    - Check that the system as a whole works correctly. This will catch interaction bugs, for example, where an incorrect value is being communicated between modules or functions, or an assumption about input is being broken in the module.
  + Cycle between these phases
* Test Drivers and Stubs
  + **Drivers** are pieces of code that:
    - Set up an environment needed to run the code.
    - Invoke code on predefined sequence of inputs.
    - Save results, and report them back.
  + Thus, a test driver is a piece of code that does the testing for us.
  + Drivers simulate parts of the program that use the unit/module being tested.
  + **Stubs** simulate parts of the program used by the unit/module being tested.
    - Allow you to test units that depend on software that has not yet been written.
* Good Testing Practice
  + Start with unit testing
  + Move to integration testing
  + Cycle back and forth between phases
  + After code is corrected, be sure to perform **regression testing**:
    - Check that the program still passes all the tests it used to pass, i.e. that your code fix hasn’t broken something that used to work.

Part 6: Debugging

* Runtime Bugs
  + **Overt vs Covert:**
    - **Overt** bug has an obvious manifestation – code crashes or continues running forever.
    - **Covert** bug has no obvious manifestation – code returns a value, which may be incorrect but is hard to determine the cause of this.
  + **Persistent vs Intermittent:**
    - **Persistent** bug occurs every time that the code is run.
    - **Intermittent** bug only occurs sometimes, even if run on the same input.
* Categories of Bugs
  + Overt and Persistent
    - Obvious to detect; usually easy to correct.
    - Good programmers use **defensive programming** to try to ensure that if an error is made, that the 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, it can be handled.
  + Covert
    - Highly dangerous, as users may not realize that the answers are incorrect until code has been run for long periods of time.

Part 7: Debugging as Search

* Debugging Skills
  + Treat finding the bug as a search problem: you are looking for an explanation for an incorrect behaviour.
    - Study available data – both correct test cases and incorrect ones.
    - Form a hypothesis consistent with the data.
    - Design and run a repeatable experiment with the potential to refute the hypothesis.
    - Keep record of experiments performed: use this to the narrow range of hypotheses.
* Debugging as Search
  + Want to narrow the space of possible sources of error.
  + Design experiments that expose intermediate stages of computation (use print statements to do this!), and use the results obtained to further narrow down the search.
  + Binary search or bisection search can be a powerful tool for this – to use information to exponentially reduce the range of places that the bug can lie in and quickly find it.
* Stepping Through the Tests
  + Suppose we run this code:
    - We try the input ‘abcba’, which succeeds.
    - We try the input ‘palinnilap’, which succeeds.
    - But we try the input ‘ab’, which incorrectly succeeds.
  + Let’s use binary search to isolate bug(s) simply using the input of ‘ab’.
  + Pick a spot about halfway into the code, and devise an experiment.
    - Pick a spot where it is easy to examine intermediate values. I’m going to put in a print statement in that spot.
* Stepping Through the Tests: Part 2
  + At this point in the code, we expect (for our test case of ‘ab’), that the result should be a list [‘a’, ‘b’].
  + We run the code, and get [‘b’].
  + Because of binary search, we know that at least one bug must be present earlier in the code.
  + So we now add a second print statement.
  + When we run with our example, the print statement returns:
    - [‘a’]
    - [‘b’]
  + This suggests that the result is not keeping all of the elements.
* Stepping Through the Tests: Part 3
  + So this now shows us that we are getting the data structure ‘result’ set up properly, but we still have a bug somewhere in the code.
    - A good reminder that there may be more than 1 bug.
    - This suggests that the second bug must lie ‘below’ the print statement; let’s take a look at the function isPal.
    - Pick a point in the middle of the code, and add a print statement again.
* Stepping Through the Tests: Part 4
  + At this point in the code, we expect (for our example of ‘ab’) that x should be [‘a’, ‘b’], but temp should be [‘b’, ‘a’], however they both have the value [‘a’, ‘b’].
  + So let’s add another print statement, just earlier in the code.
* Stepping Through the Tests: Part 5
  + And we see that temp has the same value before and after the call to reverse.
  + If we look at our code, we realize that we have committed a standard bug – we forgot to actually invoke the reverse method.
    - Need temp.reverse(); not temp.reverse
  + So let’s make that change.
* Stepping Through the Tests: Part 6
  + But now when we run on our simple example, both x and temp have been reversed.
  + We have also narrowed down this bug to a simple line. The error must lie in the reverse step.
  + And it does, because lists are mutable, so x changes if temp changes, because they point to the same object – that exact same list.
  + In fact, we have an aliasing bug – reversing temp has also caused x to be reversed.
* Stepping Through the Tests: Part 7
  + And now running this shows that before the reverse step, the two variables have the same form, but afterwards, only temp is reversed.
  + We can now go back and check that our other test cases still work properly.
* Some Pragmatic Hints
  + Look for the usual suspects.
  + Ask yourself 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 as you sift through your code trying to find the bug.
  + Explain the problem to yourself or someone else.
  + Don’t believe the documentation *you* wrote for *your* buggy code.

**Lecture 8: Assertions and Exceptions**

Part 1: Exceptions

* What is an Exception?
  + What happens when a procedure execution hits an unexpected condition?
    - Trying to access beyond the limits of a list will raise an IndexError.
* Test = [1, 2, 3]
* Test[4] 🡪 There is no 4th element
  + - Trying to convert an inappropriate type will raise a TypeError.
* Int(Test)
  + - Referencing a non-existing variable will raise a NameError.
* a
  + - Mixing data types without appropriate coercion will raise a TypeError.
* ‘a’/4
  + These are **exceptions** – exceptions to what was expected.
* What to do with Exceptions?
  + What do we do when procedure execution is stymied by an error condition?
    - Fail silently: substitute default values, continue anyways.
* Bad idea! User gets no indication that results may be incorrect.
  + - Return an “error” value.
* What value to choose? None?
* Callers must include code to check for this special case and deal with consequences 🡪 cascade of error values up the call tree.
  + - Stop execution, signal error condition to user.
* In python, raise an exception.

raise Exception(“descriptive string”)

* Dealing with Exceptions
  + Python code can provide handlers for exceptions.

try:

f = open(‘grades.txt’)

# Code to read and process grades.

except:

raise Exception(“Can’t open grades file.”)

* + Exceptions are raised by statements in the body of try statement are handled by the except statement and execution continues with the body of the except statement.
* Handling Specific Exceptions
  + Usually the handler is only meant to deal with a particular type of exception. Sometimes we need to clean up before continuing.

try:

f = open(‘grades.txt’)

# Code to read and process grades.

except IOerror,e:

print(“Can’t open grades file: ” + str(e))

sys.exit(0)

except ArithmeticError,e:

raise ValueError(“Bug in grades calculation ” + str(e))

* + The letter e that comes after the error is the actual error object that has been created, with information about the error.
* Types of Exceptions
  + Already seen common error types:
    - SyntaxError: Python can’t parse program
    - NameError: local or global name not found
    - AttributeError: attribute reference of object fails
    - TypeError: operand doesn’t have correct type
    - ValueError: operand type okay, but value is illegal
    - IOError: IO system reports malfunction (e.g. file not found)
* Other Extensions to *try*
  + else:
    - Body of this clause is executed when execution of associated try statement body completes with no exceptions.
  + finally:
    - Body of this clause is always executed after try, else and except clauses, even if they raise another error or executed a break, continue, or return statement.
    - Useful for clean-up code that should be run no matter what else happened (e.g. closing a file).
* An Example

def divide(x, y):

'''

assumes x and y are ints/floats and y != 0

raises exception for division by 0

raises exception for incorrect types

returns x / y

'''

try:

result = x / y

except ZeroDivisionError,e:

print("Division by zero! " + str(e))

except TypeError:

divide(int(x), int(y))

else:

print("Result is: " + str(result))

finally:

print("Executing finally clause.")

Part 2: Error Handling

* An Example of Exceptions
  + Here is an example of how we can use exceptions to handle unexpected situations in the code.
  + Assume we are given a class list for a subject: each entry is a list of two parts – a list of first and last names for students, and a list of grades on assignments.
  + We want to create a new subject list, with names, grades, and a weighted average of students.
* An Error if No Grades for a Student
  + If we run this on a list of students, one or more of which don’t actually have grades, we get an error.
* Let’s Flag the Error

def avg(grades, weights):

try:

return dotProduct(grades, weights)/len(grades)

except ZeroDivisionError:

print 'no grades data'

* Or We Could Change the Policy
  + Suppose we decide that a student with no grades is getting a 0 in the class.

def avg(grades, weights):

try:

return dotProduct(grades, weights)/len(grades)

except ZeroDivisionError:

print 'no grades data'

return 0.0

* We Can Handle Multiple Exceptions
  + Suppose some grades are ‘letter’ grades. We can convert them using the following function.

def convertLetterGrade(grade):

if type(grade) == int:

return grade

elif grade == 'A':

return 90.0

elif grade == 'B':

return 80.0

elif grade == 'C':

return 70.0

elif grade == 'D':

return 60.0

else:

return 50.0

* Now I Can Change the Average Function

def avg(grades, weights):

try:

return dotProduct(grades, weights)/len(grades)

except ZeroDivisionError:

print 'no grades data'

return 0.0

except TypeError:

newgrades = [convertLetterGrade(elt) for elt in grades]

return dotProduct(newgrades, weights)/len(newgrades)

Part 3: Exceptions as Control Flow

* Exceptions as Flow of Control
  + In traditional programming languages, one deals with errors by having functions return special values.
  + Any other code invoking a function has to check whether ‘error value’ was returned.
  + In Python, we can simply raise an exception when we are unable to produce a result consistent with the function’s specification.
* An Example of Exceptions

def getRatios(v1, v2):

'''

assumes v1 and v2 are lists of equal length of numbers

returns a list containing the meaning full values of v1[i] / v2[i]

'''

ratios = []

for index in range(len(v1)):

try:

ratios.append(v1[index]/float(v2[index]))

except ZeroDivisionError:

ratios.append(float('NaN')) # NaN = Not a Number

except:

raise ValueError('getRatios called with bad argument')

return ratios

* Printing Specific Examples of Code

try:

print(getRatios([1.0, 2.0, 7.0, 6.0], [1.0, 2.0, 0.0, 6.0]))

print(getRatios([],[]))

print(getRatios([1.0, 2.0], [3.0]))

except ValueError, msg:

print(msg)

* Compare to Traditional Code

def getRatios(v1, v2):

'''

assumes v1 and v2 are lists of equal length of numbers

returns a list containing the meaning full values of v1[i] / v2[i]

'''

ratios = []

if len(v1) != len(v2):

raise ValueError('getRatios called with bad argument')

for index in range(len(v1)):

if (type(v1[index]) not in (int, float)) or (type(v2[index]) not in (int, float)):

raise ValueError('getRatios called with bad argument')

if v2[index] == 0.0:

ratios.append(float('NaN')) # NaN = Not a Number

else:

ratios.append(float(v1[index] / v2[index]))

return ratios

* Compare New Code to Traditional Code
  + Easier to read, and thus easier to maintain or modify.
  + More efficient because all checks don’t have to be run every time the procedure is called.
  + Easier to think about processing on data structure abstractly, with exceptions to deal with unusual or special cases.

Part 4: Assertions

* Assertions
  + If we simply want to be sure that assumptions on state of computation are as expected, we can use an assert statement.
  + We can’t control the program’s response, but this will raise an AssertionError exception if this happens.
  + This is a good practice of defensive programming because you are catching bugs ahead of time.
* Example

def avg(grades, weights):

'''

assumes grades and weights are lists of equal length and are not empty

returns the weighted average of the assignments

'''

assert not len(grades) == 0, ‘no grades data’

newGrades = [convertLetterGrade(elt) for elt in grades]

return dotProduct(newGrades, weights) / len(grades)

* + This will raise an AssertionError if it is given an empty list for grades, but otherwise will run properly.
    - Error will print out ‘no grades data’ as part of process.
* Assertions as Defensive Programming
  + While assertions don’t allow a programmer to control the response to unexpected conditions, they are a great method for ensuring that execution halts whenever an expected condition is not met.
  + Typically used to check inputs to procedures, but can be used anywhere.
  + Can make it easier to locate the source of a bug.
* Extending Use of Assertions
  + While pre-conditions on inputs are valuable to check, we can also apply post-condition checks on outputs before proceeding to the next stage.
* Example, Extended

def avg(grades, weights):

'''

assumes grades and weights are lists of equal length and are not empty

returns the weighted average of the assignments

'''

assert not len(grades) == 0, 'no grades data'

assert len(grades) == len(weights), 'wrong number of grades'

newGrades = [convertLetterGrade(elt) for elt in grades]

result = dotProduct(newGrades, weights) / len(grades)

assert 0.0 <= result <= 100.0, 'invalid average'

return result

* Example, Extended
  + Slight loss of efficiency in code.
  + Defensive Programming:
    - By checking pre- and post-conditions on inputs and output, we avoid propagating bad values.
* Where to Use Assertions?
  + Goal is to spot bugs early, and make clear where they occurred.
    - Easier to debug when caught at first point of contact, instead of trying to trace down later.
  + Not to be used in place of testing, but as a supplement to testing.
  + Should probably rely on raising exceptions if bad data input is supplied, use assertions for:
    - Checking types of arguments or values.
    - Checking that invariants on data structures are met.
    - Checking constraints on input and return values.
    - Checking for violations of constraints on procedure (e.g. no duplicates in a list).