Python Programming

Chapter 4 :

Testing, Debugging, Exceptions and Assertions

Introduction

- **Testing** is the process of running a program to try and ascertain whether or not it works as intended.
- **Debugging** is the process of trying to fix a program that you already know does not work as intended.
- Good programmers design their programs in ways that make them easier to test and debug.
- The key to doing this is breaking the program up into separate components that can be implemented, tested, and debugged independently of other components.

Testing

- The most important thing to say about testing is that its purpose is to show that bugs exist, not to show that a program is bug-free.

```
def isBigger(x, y):
      """Assumes x and y are ints Returns True if x
      is less than y and False otherwise."""
```

- Running it on all pairs of integers would be, to say the least, tedious.
- The best we can do is to run it on pairs of integers that have a reasonable probability of producing the `wrong` answer if there is a bug in the program.

Testing...

- The key to testing is finding a collection of inputs, called a **test suite**, that has a high likelihood of revealing bugs, yet does not take too long to run.
- A **partition** of a set divides that set into a collection of subsets such that each element of the original set belongs to exactly one of the subsets.

Testing...

- One way to partition this set is into these seven subsets:
- x positive, y positive
- x negative, y negative
- x positive, y negative
- x negative, y positive
- $x = 0, y = 0$
- $x = 0, y \neq 0$
- $x \neq 0, y = 0$

Testing…

- For most programs, finding a good partitioning of the inputs is far easier said than done.
- Heuristics based on exploring paths through the code fall into a class called **glass-box testing**.
- Heuristics based on exploring paths through the specification fall into a class called **black-box testing**.

Black Box Testing

- black-box tests are constructed without looking at the code to be tested.
- author of a program made the implicit, but invalid, assumption that a function would never be called with a negative number.
- If the same person constructed the test suite for the program, he would likely repeat the mistake, and not test the function with a negative argument.

Black Box Testing…

- Another positive feature of black-box testing is that it is robust with respect to implementation changes.
- Since the test data is generated without knowledge of the implementation, it need not be changed when the implementation is changed.
- As we said earlier, a good way to generate black-box test data is to explore paths through a specification.

Black Box Testing…

```
def sqrt(x, epsilon):
      """Assumes x, epsilon floats
            x >= 0
            epsilon > 0
      Returns result such that
            x-epsilon <= result*result <= x+epsilon"""
```

- There seem to be only two distinct paths through this specification: one corresponding to x = 0 and one corresponding to x > 0.
- However, common sense tells us that while it is necessary to test these two cases, it is hardly sufficient.
- Boundary conditions should also be tested.

Black Box Testing…

x	epsilon
0.0	0.0001
25.0	0.0001
0.5	0.0001
2.0	0.0001
2.0	1.0/2.0**64.0
1.0/2.0**64	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

- Notice that the values for x include a perfect square, a number less than one, and a number with an irrational square root.
- If any of these tests fail, there is a bug in the program that needs to be fixed.
- The remaining rows test extremely large and small values of x and epsilon.
- If any of these tests fail, something needs to be fixed. Perhaps there is a bug in the code that needs to be fixed, or perhaps the specification needs to be changed so that it is easier to meet.

Black Box Testing…

- Another important boundary condition to think about is aliasing.

```
def copy(L1, L2):
      """Assumes L1, L2 are lists
            Mutates L2 to be a copy of L1"""
      while len(L2) > 0: #remove all elements from L2
            L2.pop() #remove last element of L2
      for e in L1: #append L1's elements to initially empty L2
            L2.append(e)
```

- It will work most of the time, but not when L1 and L2 refer to the same list.
- Any test suite that did not include a call of the form copy(L, L), would not reveal the bug.

Glass Box Testing

- Black-box testing should never be skipped, but it is rarely sufficient.
- Without looking at the internal structure of the code, it is impossible to know which test cases are likely to provide new information.

```
def isPrime(x):
      """Assumes x is a nonnegative int
            Returns True if x is prime; False otherwise"""
      if x <= 2:
            return False
      for i in range(2, x):
            if x%i == 0:
                  return False
      return True
```

Glass Box Testing…

- Glass-box test suites are usually much easier to construct than black-box test suites.
- Specifications are usually incomplete and often pretty sloppy, making it a challenge to estimate how thoroughly a black-box test suite explores the space of interesting inputs.
- A glass-box test suite is path-complete if it exercises every potential path through the program.
- This is typically impossible to achieve, because it depends upon the number of times each loop is executed and the depth of each recursion.

Glass Box Testing…

```
def abs(x):
      """Assumes x is an int
      Returns x if x>=0 and -x otherwise"""
      if x < -1:
            return -x
      else:
            return x
```

- The specification suggests that there are two possible cases, x is either negative or it isn't.
- This suggests that the set of inputs {2, -2} is sufficient to explore all paths in the specification.

Thumb rules of Glass Box Testing

- Exercise both branches of all if statements.
- Make sure that each except clause is executed.
- For each for loop, have test cases in which
  - The loop is not entered (e.g., if the loop is iterating over the elements of a list, make sure that it is tested on the empty list),
  - The body of the loop is executed exactly once, and
  - The body of the loop is executed more than once.

Thumb rules of Glass Box Testing …

- For each while loop,
  - Look at the same kinds of cases as when dealing with for loops, and
  - Include test cases corresponding to all possible ways of exiting the loop. For example, for a loop starting with
  - `while len(L) > 0 and not L[i] == e`
  - find cases where the loop exits because len(L) is greater than zero
  - and cases where it exits because L[i] == e.

Thumb rules of Glass Box Testing ...

- For recursive functions,
  - include test cases that cause the function to return with no recursive calls, exactly one recursive call, and more than one recursive call.

Conducting Tests

- Unit Testing
- Integration Testing
- Software quality assurance (SQA)
- Testers do not sit at terminals typing inputs and checking outputs. They use test drivers
  - Set up the environment needed to invoke the program (or unit) to be tested,
  - Invoke the program (or unit) to be tested with a predefined or automatically generated sequence of inputs,
  - Save the results of these invocations,
  - Check the acceptability of the results of the tests, and
  - Prepare an appropriate report.

Conducting Tests…

- Ideally, a stub should
  - Check the reasonableness of the environment and arguments supplied by the caller (calling a function with inappropriate arguments is a common error),
  - Modify arguments and global variables in a manner consistent with the specification, and
  - Return values consistent with the specification.

Debugging

- The use of the word “bug” sometimes leads people to ignore the fundamental fact that if you wrote a program and it has a “bug,” you messed up.
- If your program has multiple bugs, it is because you made multiple mistakes.

Debugging…

- Runtime bugs can be categorized along two dimensions:

1. **Overt → covert**:
   - An overt bug has an obvious manifestation, e.g., the program crashes or takes far longer (maybe forever) to run than it should.
   - A covert bug has no obvious manifestation. The program may run to conclusion with no problem—other than providing an incorrect answer.
   - Many bugs fall between the two extremes, and whether or not the bug is overt can depend upon how carefully one examines the behavior of the program.

Debugging…

2. Persistent → intermittent:

- A persistent bug occurs every time the program is run with the same inputs.
- An intermittent bug occurs only some of the time, even when the program is run on the same inputs and seemingly under the same conditions.

Debugging…

- The best kinds of bugs to have are overt and persistent.
- Developers can be under no illusion about the advisability of deploying the program.
- And if someone else is foolish enough to attempt to use it, they will quickly discover their folly.
- Perhaps the program will do something horrible before crashing, e.g., delete files, but at least the user will have reason to be worried (if not panicked).
- Good programmers try to write their programs in such a way that programming mistakes lead to bugs that are both overt and persistent.
- This is often called **defensive programming**.

Debugging…

- Programs that fail in covert ways are often highly dangerous.
- Since they are not apparently problematical, people use them and trust them to do the right thing.
- A program that makes a covert error only occasionally may or may not wreak less havoc than one that always commits such an error.
- Bugs that are both covert and intermittent are almost always the hardest to find and fix.

Learning to debug

- Debugging is a learned skill. Nobody does it well instinctively.
- The good news is that it's not hard to learn, and it is a transferable skill.
- The same skills used to debug software can be used to find out what is wrong with other complex systems, e.g., laboratory experiments or sick humans.
- For at least four decades people have been building tools called debuggers.
- Debugging starts when testing has demonstrated that the program behaves in undesirable ways.
- Debugging is the process of searching for an explanation of that behavior.
- The key to being consistently good at debugging is being systematic in conducting that search.

Learning to debug…

- Start by studying the available data. This includes the test results and the program text.
- Study all of the test results. Examine not only the tests that revealed the presence of a problem, but also those tests that seemed to work perfectly.
- Trying to understand why one test worked and another did not is often illuminating.
- Remember, as many have said, “insanity is doing the same thing, over and over again, but expecting different results.”

Designing the Experiment

- Think of debugging as a search process, and each experiment as an attempt to reduce the size of the search space.
- One way to reduce the size of the search space is to design an experiment that can be used to decide whether a specific region of code is responsible for a problem uncovered during integration testing.
- Another way to reduce the search space is to reduce the amount of test data needed to provoke a manifestation of a bug.

Designing the Experiment…

```
def isPal(x):
    """Assumes x is a list
       Returns True if the list is a palindrome; False otherwise"""
    temp = x
    temp.reverse
    if temp == x:
        return True
    else:
        return False

def silly(n):
    """Assumes n is an int > 0
       Gets n inputs from user
       Prints 'Yes' if the sequence of inputs forms a palindrome;
           'No' otherwise"""
    for i in range(n):
        result = []
        elem = raw_input('Enter element: ')
        result.append(elem)
    if isPal(result):
        print 'Yes'
    else:
        print 'No'
```

Figure 6.1 Program with bugs

Designing the Experiment…

>>> silly(2)
Enter element: a
Enter element: b

- The good news is that it fails even this simple test, so you don't have to type in a thousand strings.
- The bad news is that you have no idea why it failed.
- In this case, the code is small enough that you can probably stare at it and find the bug (or bugs).
- However, let's pretend that it is too large to do this, and start to systematically reduce the search space.

Designing the Experiment…

```
def silly(n):
    """Assumes n is an int > 0
       Gets n inputs from user
       Prints 'Yes' if the sequence of inputs forms a palindrome;
           'No' otherwise"""
    result = []
    for i in range(n):
        elem = raw_input('Enter element: ')
        result.append(elem)
    print result
    if isPal(result):
        print 'Yes'
    else:
        print 'No'
```

Designing the Experiment…

- So, let’s look at isPal.
- The line if temp == x: is about halfway through that function.
- When we run the code, we see that temp has the expected value, but x does not.
- Moving up the code, we insert a print statement after the line temp = x, and discover that both temp and x have the value ['a', 'b'].
- A quick inspection of the code reveals that in isPal we wrote temp.reverse rather than temp.reverse()—the evaluation of temp.reverse returns the built-in reverse method for lists, but does not invoke it.

Designing the Experiment…

- We run the test again, and now it seems that both temp and x have the value ['b', 'a'].
- We have now narrowed the bug to one line.
- It seems that temp.reverse() unexpectedly changed the value of x.
- An aliasing bug has bitten us: temp and x are names for the same list, both before and after the list gets reversed.
- One way to fix it is to replace the first assignment statement in isPal by temp = x[:], which causes a copy of x to be made.

Designing the Experiment…

```
def isPal(x):
    """Assumes x is a list
       Returns True if the list is a palindrome; False otherwise"""
    temp = x[:]
    temp.reverse()
    if temp == x:
        return True
    else:
        return False
```

When the Going Gets Tough

- Look for the usual suspects. E.g., have you
  - Passed arguments to a function in the wrong order,
  - Misspelled a name, e.g., typed a lowercase letter when you should have typed an uppercase one,
  - Failed to reinitialize a variable,
  - Tested that two floating point values are equal (==) instead of nearly equal (remember that floating point arithmetic is not the same as the arithmetic you learned in school),
  - Tested for value equality (e.g., compared two lists by writing the expression L1 == L2) when you meant object equality (e.g., id(L1) == id(L2)),
  - Forgotten that some built-in function has a side effect,
  - Forgotten the () that turns a reference to an object of type function into a function invocation,
  - Created an unintentional alias, or
  - Made any other mistake that is typical for you.

When the Going Gets Tough…

- Stop asking yourself why the program isn't doing what you want it to. Instead, ask yourself why it is doing what it is.
- *Keep in mind that the bug is probably not where you think it is.* If it were, you would probably have found it long ago. One practical way to go about deciding where to look is asking where the bug cannot be.
- *Try to explain the problem to somebody else.* We all develop blind spots. It is often the case that merely attempting to explain the problem to someone will lead you to see things you have missed.
- *Don't believe everything you read.* In particular, don't believe the documentation.
- *Stop debugging and start writing documentation.*
- *Walk away, and try again tomorrow*.

And When You Have Found “The” Bug

- When you think you have found a bug in your code, the temptation to start coding and testing a fix is almost irresistible.
- It is often better, however, to slow down a little.
- Remember that the goal is not to fix one bug, but to move rapidly and efficiently towards a bug-free program.
- Before making any change, try and understand the ramification of the proposed “fix.”
- Will it break something else? Does it introduce excessive complexity? Does it offer the opportunity to tidy up other parts of the code?
- Finally, if there are many unexplained errors, you might consider whether finding and fixing bugs one at a time is even the right approach.

Exception

- An “exception” is usually defined as “something that does not conform to the norm,” and is therefore somewhat rare.
- There is nothing rare about exceptions in Python.
- They are everywhere. Virtually every module in the standard Python library uses them, and Python itself will raise them in many different circumstances.

Open a Python shell and enter,

```
test = [1,2,3]
test[3]
```

and the interpreter will respond with something like

```
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    test[3]
IndexError: list index out of range
```

Exception…

- IndexError is the type of exception that Python **raises** when a program tries to access an element that is not within the bounds of an indexable type.
- The string following IndexError provides additional information about what caused the exception to occur.
- Most of the built-in exceptions of Python deal with situations in which a program has attempted to execute a statement with no appropriate semantics.

Handling Exceptions

- Up to now, we have treated exceptions as fatal events.
- When an exception is raised, the program terminates (crashes might be a more appropriate word in this case), and we go back to our code and attempt to figure out what went wrong.
- When an exception is raised that causes the program to terminate, we say that an unhandled exception has been raised.
- An exception does not need to lead to program termination.
- Exceptions, when raised, can and should be handled by the program.

Handling Exceptions…

Consider the code

```
successFailureRatio = numSuccesses/float(numFailures)
print 'The success/failure ratio is', successFailureRatio
print 'Now here'
```

Most of the time, this code will work just fine, but it will fail if `numFailures` happens to be zero. The attempt to divide by zero will cause the Python runtime system to raise a `ZeroDivisionError` exception, and the `print` statements will never be reached.

It would have been better to have written something along the lines of

```
try:
    successFailureRatio = numSuccesses/float(numFailures)
    print 'The success/failure ratio is', successFailureRatio
except ZeroDivisionError:
    print 'No failures so the success/failure ratio is undefined.'
print 'Now here'
```

Handling Exceptions…

Finger exercise: Implement a function that meets the specification below. Use a `try-except` block.

```
def sumDigits(s):
    """Assumes s is a string
       Returns the sum of the decimal digits in s
          For example, if s is 'a2b3c' it returns 5"""
```

Let's look at another example. Consider the code

```
val = int(raw_input('Enter an integer: '))
print 'The square of the number you entered is', val**2
```

- Executing the line of code will cause the Python runtime system to raise a ValueError exception, and the print statement will never be reached.

Handling Exceptions...

What the programmer should have written would look something like

```
while True:
    val = raw_input('Enter an integer: ')
    try:
        val = int(val)
        print 'The square of the number you entered is', val**2
        break #to exit the while loop
    except ValueError:
        print val, 'is not an integer'
```

Handling Exceptions...

```
def readInt():
    while True:
        val = raw_input('Enter an integer: ')
        try:
            val = int(val)
            return val
        except ValueError:
            print val, 'is not an integer'
```

Better yet, this function can be generalized to ask for any type of input,

```
def readVal(valType, requestMsg, errorMsg):
  while True:
      val = raw_input(requestMsg + ' ')
      try:
          val = valType(val)
          return val
      except ValueError:
          print val, errorMsg
```

Thank you!!!