sugarlabs

Scott White, Logan Smith, and Brielen Beamon

Step 1: Building the project

After researching FOSS projects, we decided to test Sugar Labs, an educational software associated with the One Laptop Per Child movement. We were able to build and run Sugar Labs only by excluding a certain module. Unfortunately, excluding this module broke all applications requiring an internet connection.

Step 2: Making a Plan

Since only offline modules were available to us for testing, we focused our efforts on the calculator module. Specifically, we decided to test the power, addition, cosine, square root, and division functions. We defined the structure of our test case files, a sample of which is on the left. We also defined a structure for test results output, a sample of which is underneath the sample test case.

Step 3: Building the Framework

Our testing framework is composed of five python files. The first, runAllTests.py, calls parseTestCase.py on each test case file to extract information about each case. It then calls runTestCase.py on each function defined in each case with the specified input and makes sure the output matches the expected output. myException.py defines a custom exception for poorly written test cases. Finally, progressBar.py shows progress graphically on the command line.

Sample Test Case:

```
Test Case ID: 1

Requirement being tested:
  Correct and accurate exponentiation

Component being tested:
  Sugar labs calculator - functions.py
  Path to file:
TestAutomation/project/sugar-calculate/function
s.py

Method being tested: pow(a, b)

Test input(s) including command-line arguments:
2, 10

Expected outcome(s): 1024
```

Step 4: Writing Test Cases

Next, we had to write test cases for the framework to run. This part was fairly simple, but we did run into an issue with error messages. In order to find out what message we would get when we divided by zero, for example, we had to actually divide by zero to see what runtime error we would get. Once we had all of the error messages we needed, it was a simple task to create all 25 test cases.

Step 5: Fault Injection

Our final task was to compromise the source code and then observe the effects on our automated testing framework. To this end, we inserted a fault into each function we tested.

Testing Results

Test Case ID	Tested Method	Input	Expected Output	Actual Outcome	Pass/Fail
1	pow(a, b)	2, 10	1024	1024	Pass
2	pow(a, b)	4.0, 2.5	32.0	32	Pass
3	pow(a, b)	16, 0.5	4.0	4	Pass
4	pow(a, b)	16, -0.5	0.25	0.25	Pass
5	pow(a, b)	0, 5	0	0	Pass

Sample Test Results

Conclusion

While working on this project, we've learned to use git, write tests, and break software in every way we can think of. These skills are applicable to every project we do in the future and what separates a good programmer from a great one.

4 TestAutomation/project/sugar-calculate/rational.py

```
@@ -31,8 +31,8 @@ def __init__(self, n=None, d=None):
31 31          self.n = 0
32 32          self.d = 0
33 33
34    -        #if n is not None:
35    -        #    self.set(n, d)
   34 +        if n is None:
   35 +            self.set(n, d)
36 36
37 37      def set(self, n, d=None):
38 38          if d is not None:
```

Sample Fault Injection