Fuzzing Book

Tools and Techniques for Generating Software Tests

by Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler

Introduction to Software Testing

Simple Testing

Computes the square root of x, using the Newton-Raphson method

In [1]:
```
def my_sqrt(x):
    approx = None
    guess = x / 2
    while approx != guess:
        approx = guess
        guess = (approx + x / approx) / 2
    return approx
```

Running a Function

Test it with a few values. For x = 4 and x = 2.
It produces the correct value:

```
In [2]: my_sqrt(4)

Out[2]: 2.0

In [3]: my_sqrt(2)

Out[3]: 1.414213562373095
```

Debugging a Function

Insert `print()`

In [4]:

```
def my_sqrt_with_log(x):
    approx = None
    guess = x / 2
    while approx != guess:
        print("approx =", approx)  # <-- New
        approx = guess
        guess = (approx + x / approx) / 2
    return approx
```

Debugging a Function

In [5]:
```
my_sqrt_with_log(9)
```

```
approx = None
approx = 4.5
approx = 3.25
approx = 3.0096153846153846
approx = 3.000015360039322
approx = 3.0000000000393214
```

Out[5]:
```
3.0
```

Checking a Function

Are the above values of `my_sqrt(2)` actually correct?

```
In [6]: my_sqrt(2) * my_sqrt(2)

Out[6]: 1.9999999999999996
```

Automating Test Execution

We have tested the above program manually.
Very flexible, but inefficient:

Automating Test Execution

We have tested the above program manually.
Very flexible, but inefficient:

1. Manually, you can only check a very limited number of executions and their results.

Automating Test Execution

We have tested the above program manually.
Very flexible, but inefficient:

1. Manually, you can only check a very limited number of executions and their results.

2. After any change to the program, you have to repeat the testing process.

Automating Test Execution

How about automate tests?

One simple way of doing so is to let the computer first do the computation, and then have it check the results.

Automating Test Execution

Example: $\sqrt{4} = 2$

In [7]:

```
result = my_sqrt(4)
expected_result = 2.0
if result == expected_result:
    print("Test passed")
else:
    print("Test failed")
```

```
Test passed
```

Issues

- We need *five lines of code* for a single test
- We do not care for rounding errors
- We only check a single input (and a single result)

Assertion

- If the condition evaluates to false, though, assert raises an exception.

In [8]:
```
assert my_sqrt(4) == 2
```

< Nothing happens >

Rounding Errors

- Ensure that the absolute difference between them stays below a certain threshold value, typically denoted as ϵ or epsilon.

In [9]:
```
EPSILON = 1e-8
```

In [10]:
```
assert abs(my_sqrt(4) - 2) < EPSILON
```

Check Multiple Inputs

- Introduce a special function for above purpose, and now do more tests for concrete values:

In [11]:
```
def assertEquals(x, y, epsilon=1e-8):
    assert abs(x - y) < epsilon
```

In [12]:
```
assertEquals(my_sqrt(4), 2)
assertEquals(my_sqrt(9), 3)
assertEquals(my_sqrt(100), 10)
```

Generating Tests

- Test $\sqrt{x} \times \sqrt{x} = x$

In [13]:
```
assertEquals(my_sqrt(2) * my_sqrt(2), 2)
assertEquals(my_sqrt(3) * my_sqrt(3), 3)
assertEquals(my_sqrt(42.11) * my_sqrt(42.11), 42.11)
```

Generating Tests

- Test $\sqrt{x} \times \sqrt{x} = x$

In [13]:
```
assertEquals(my_sqrt(2) * my_sqrt(2), 2)
assertEquals(my_sqrt(3) * my_sqrt(3), 3)
assertEquals(my_sqrt(42.11) * my_sqrt(42.11),
42.11)
```

In [14]:
```
for n in range(1, 1000):
    assertEquals(my_sqrt(n) * my_sqrt(n), n)
```

Run-Time Verification

- Integrate the check right into the implementation

In [20]:
```
def my_sqrt_checked(x):
    root = my_sqrt(x)
    assertEquals(root * root, x)
    return root
```

Run-Time Verification

- Integrate the check right into the implementation

```
In [20]: def my_sqrt_checked(x):
             root = my_sqrt(x)
             assertEquals(root * root, x)
             return root
```

```
In [21]: my_sqrt_checked(2.0)
```

```
Out[21]: 1.414213562373095
```

Automatic Run-time Checks

- Assume two things, though:
  - One has to be able to *formulate* such run-time checks.
  - One has to be able to *afford* such run-time checks.

System Input vs Function Input

- Input that comes from *third parties*.

```
In [22]: def sqrt_program(arg):
             x = int(arg)
             print('The root of', x, 'is', my_sqrt(x))
```

System Input vs Function Input

- Input that comes from *third parties*.

In [22]:
```
def sqrt_program(arg):
    x = int(arg)
    print('The root of', x, 'is', my_sqrt(x))
```

In [23]:
```
sqrt_program("4")
```

```
The root of 4 is 2.0
```

What's the problem?

- Try invoking sqrt_program("-1")

What's the problem?

- Try invoking sqrt_program("-1")

```
*** It enters an infinite loop ***
```

What's the problem?

- We use a special with ExpectTimeOut(1) construct to interrupt execution after one second.

In [24]:
```
from ExpectError import ExpectTimeout
```

In [25]:
```
with ExpectTimeout(1):
    sqrt_program("-1")
```

```
Traceback (most recent call last):
...
TimeoutError (expected)
```

Check External Input

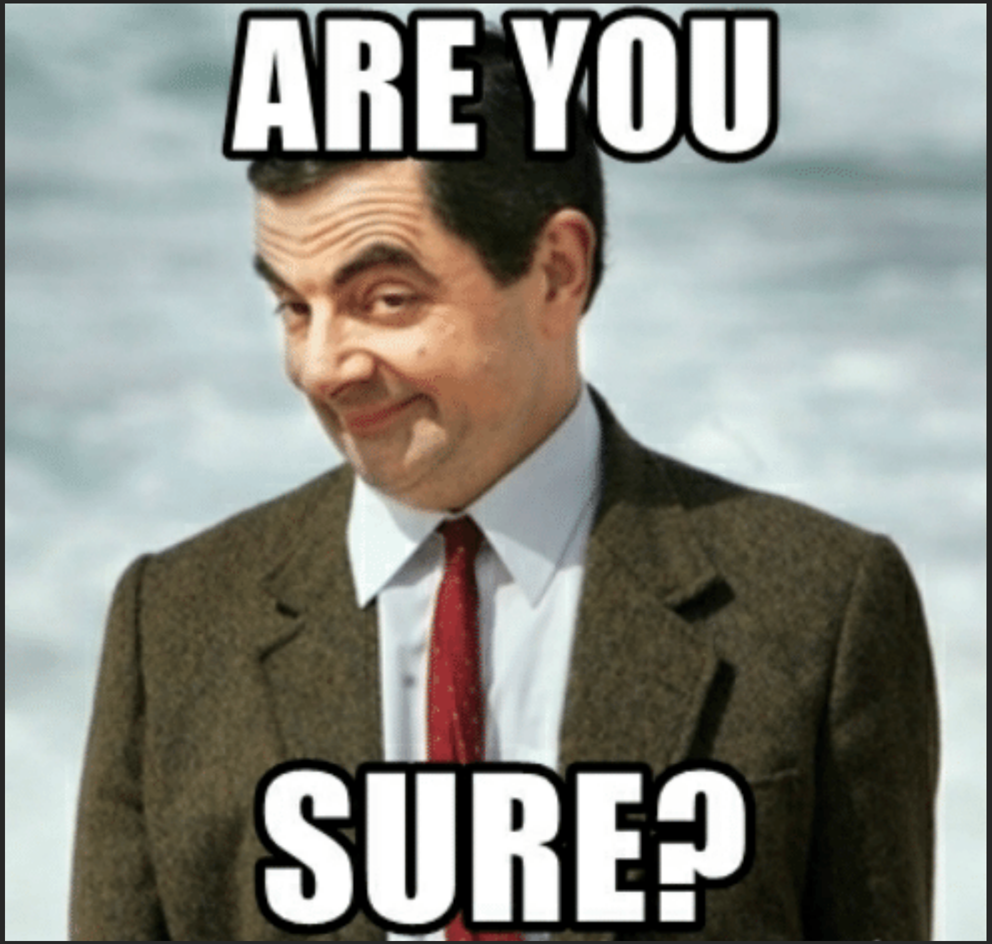
```
In [26]: def sqrt_program(arg):
             x = int(arg)
             if x < 0:
                 print("Illegal Input")
             else:
                 print('The root of', x, 'is', my_sqrt(x))
```

Check External Input

In [26]:
```
def sqrt_program(arg):
    x = int(arg)
    if x < 0:
        print("Illegal Input")
    else:
        print('The root of', x, 'is', my_sqrt(x))
```

In [27]:
```
sqrt_program("-1")
```

```
Illegal Input
```


ARE YOU
SURE?

Another Problem

- What if sqrt_program() is not invoked with a number?

Another Problem

- What if sqrt_program() is not invoked with a number?
  - checks for bad inputs.

In [30]:
```
def sqrt_program(arg):
    try:
        x = float(arg)
    except ValueError:
        print("Illegal Input")
    else:
        if x < 0:
            ...
```

In [31]:
```
sqrt_program("4")
```
```
The root of 4.0 is 2.0
```

In [32]:
```
sqrt_program("-1")
```
```
Illegal Input
```

In [33]:
```
sqrt_program("xyz")
```
```
Illegal Input
```

The Limits of Testing

- *finite* set of inputs.
  There may always be untested inputs for which the function may still fail.

The Limits of Testing

- *finite* set of inputs.
  There may always be untested inputs for which the function may still fail.

The Limits of Testing

- *finite* set of inputs.
  There may always be untested inputs for which the function may still fail.
- Consider input with `"0"`.
  Computing $\sqrt{0}$ results in a division by zero.

Fix It

- What if sqrt_program() is not invoked with a number?
  - checks for bad inputs.

In [35]:
```python
def my_sqrt_fixed(x):
    assert 0 <= x
    if x == 0:
        return 0
    return my_sqrt(x)
```

Can we guarantee that all future executions will be correct?

- No guarantee that future executions may not lead to a failing check.
- We can only guarantee that if it produces a result, the result will be correct.

Lessons Learned

- The aim of testing is to execute a program such that we find bugs.
- Test execution, test generation, and checking test results can be automated.
- Testing is *incomplete*; it provides no 100% guarantee that the code is free of errors.