Deliverable #5

Team Silver Bullets

For this deliverable, our team injected five different faults into the code, one for each of the five methods we tested.

add(x,y)

Pre-faults

1	add(x,y)	6 , 2	TestCase1.py	8	8	Pass
2	add(x,y)	1 , "testString"	TestCase2.py	TypeError	TypeError	Pass
3	add(x,y)	-1 , -1	TestCase3.py	-2	-2	Pass
4	add(x,y)	0 , 7	TestCase4.py	7	7	Pass
5	add(x,y)	.99999999999999999999 , .00000000000000000001	TestCase5.py	1.0	1.0	Pass

Post-faults

1	add(x,y)	6 , 2	TestCase1.py	8.0	8	Fail
2	add(x,y)	1 , "testString"	TestCase2.py	TypeError	TypeError	Pass
3	add(x,y)	-1 , -1	TestCase3.py	2.0	-2	Fail
4	add(x,y)	0 , 7	TestCase4.py	7.0	7	Fail
5	add(x,y)	.99999999999999999999 , .00000000000000000001	TestCase5.py	1.0	1.0	Pass

For the add(x,y) method, we added a fault to return the absolute value of the addition results. This not only made the expected negative results positive, it also ended up converting the value to contain a floating point! This was unforeseen, and caused other tests to fail that we expected to pass.

Changes :

```
#fault inserted: added abs method to return value
return abs(x + y)
```

sub(x,y)

Pre-faults

6	sub(x,y)	-1 , 3	TestCase6.py	-4	-4	Pass
7	sub(x,y)	-5 , -10	TestCase7.py	5	5	Pass
8	sub(x,y)	.1911 , .9998089	TestCase8.py	-0.8087089	-0.8087089	Pass
9	sub(x,y)	"silver" , "bullets"	TestCase9.py	TypeError	TypeError	Pass
10	sub(x,y)	"00099" , 66	TestCase10.py	TypeError	TypeError	Pass

Post-faults

6	sub(x,y)	-1 , 3	TestCase6.py	-4	-4	Pass
7	sub(x,y)	-5 , -10	TestCase7.py	5	5	Pass
8	sub(x,y)	.1911 , .9998089	TestCase8.py	-0.8087089	-0.8087089	Pass
9	sub(x,y)	"silver" , "bullets"	TestCase9.py	TypeError	TypeError	Pass
10	sub(x,y)	"00099" , 66	TestCase10.py	TypeError	TypeError	Pass

We removed the if statement that would convert values to decimals.

Changes :

```
#fault inserted: took away if statement so it would not convert values to
decimals.
#if isinstance(x, _Decimal) or isinstance(y, _Decimal):
 #    x = _d(x)
  #   y = _d(y)
return x - y
```

mul(x,y)

Pre-faults

11	mul(x,y)	0 , .55555	TestCase11.py	0.0	0.0	Pass
12	mul(x,y)	-7 , -7	TestCase12.py	49	49	Pass
13	mul(x,y)	8 , -8	TestCase13.py	-64	-64	Pass
14	mul(x,y)	.000001 , .011111111	TestCase14.py	1.1111111e-08	1.1111111e-08	Pass
15	mul(x,y)	"silver" , 4	TestCase15.py	silversilversilversilver	TypeError	Fail

Post-faults

11	mul(x,y)	0 , .55555	TestCase11.py	None	0.0	Fail
12	mul(x,y)	-7 , -7	TestCase12.py	None	49	Fail
13	mul(x,y)	8 , -8	TestCase13.py	None	-64	Fail
14	mul(x,y)	.000001 , .011111111	TestCase14.py	None	1.1111111e-08	Fail
15	mul(x,y)	"silver" , 4	TestCase15.py	None	TypeError	Fail

For this method, we indented the return statement to be included in an if statement. The if statement checks if either number is a decimal. Indenting it causes Python to return nothing when the method is called, if the if statement isn't checked.

Changes :

```
if isinstance(x, _Decimal) or isinstance(y, _Decimal):
    x = _d(x)
    y = _d(y)
    #fault inserted: indented return statement.
    return x * y
```

div(x,y)

Pre-faults

16	div(x,y)	4 , 0	TestCase16.py	ValueError	ValueError	Pass
17	div(x,y)	0 , 4	TestCase17.py	0	0	Pass
18	div(x,y)	"hhh8" , 100	TestCase18.py	TypeError	TypeError	Pass
19	div(x,y)	5.18 , 1.65	TestCase19.py	3.13939393939	3.13939393939	Pass
20	div(x,y)	1 , .999999999999	TestCase20.py	1.0	1.0	Pass

Post-faults

16	div(x,y)	4 , 0	TestCase16.py	4	ValueError	Fail
17	div(x,y)	0 , 4	TestCase17.py	ValueError	0	Fail
18	div(x,y)	"hhh8" , 100	TestCase18.py	TypeError	TypeError	Pass
19	div(x,y)	5.18 , 1.65	TestCase19.py	3.13939393939	3.13939393939	Pass
20	div(x,y)	1 , .999999999999	TestCase20.py	1.0	1.0	Pass

We changed the clause to check for is ‘y’ is a 0 to check if ‘x’ is a 0. This causes the test to return an incorrect result when attempting to divide by zero. In test 16, we expect a ValueError, however they have a clause in there that returns ‘x’ if the ‘y’ is a 0. In test 17, a ValueError is thrown, since that is the result when it checks to see if the ‘x’ value is 0.

Changes :

```
if x == 0 or x == 0.0:#fault insert changed y to x
```

mod(x,y)

Pre-faults

21	mod(x,y)	-340 , 60	TestCase21.py	20	20	Pass
22	mod(x,y)	340 , 60	TestCase22.py	40	40	Pass
23	mod(x,y)	0 , 10000	TestCase23.py	0	0	Pass
24	mod(x,y)	4 , 3.14	TestCase24.py	ValueError	ValueError	Pass
25	mod(x,y)	-100 , -23.14	TestCase25.py	ValueError	ValueError	Pass

Post-faults

21	mod(x,y)	-340 , 60	TestCase21.py	20	20	Pass
22	mod(x,y)	340 , 60	TestCase22.py	40	40	Pass
23	mod(x,y)	0 , 10000	TestCase23.py	0	0	Pass
24	mod(x,y)	4.1 , 3	TestCase24.py	1.1	.86	Fail
25	mod(x,y)	-100 , -23.14	TestCase25.py	-7.44	-7.44	Pass

For the mod function, we removed the clause that checks if ‘y’ is an integer. This allows us to mod by decimal numbers, which the program wasn’t allowing us to do before. While this is added functionality, it is still technically a fault from what we expected the program to return.

Changes:

```
#if is_int(y):
return x % y
#else:
    #raise ValueError(_('Can only calculate x modulo <integer>'))
```