CS562 Applied Software Engineering

Final Report

Zhicheng Fu

What I have accomplished

In this project, I tested a python library named "Equation", which we can get from github. I tested it by using TSTL. Firstly, I read the instruction of the Equation library to understand what this library can do. And I learnt that this library can interpret input expression. The input can be a simply formula or complex function.

Firstly I generate some parameters, also I generate real number. A real number combining an imaginary part 'j' can indicates an imaginary number. Adding a real number and an imaginary number we can get complex number. The next step is generate expressions by connecting numbers or parameters with different operators, such as '+', '-', 'sin', 'cos'. I use assert function to judge if the output of Expression function is an empty string. I use this method because if there is an error, then the process should stop with a prompt of Type Error. Otherwise the equation should be interpreted. So the library function should never return an empty string. Then I print the output of the Expression function to see what it interprets. Also I evaluate the equation from the Expression function to check if it is evaluated well.

After a lot of testing, I found two bugs. Also I did coverage analysis by TSTL.

Bugs Description

First Bug

I found two kinds of bugs when I am testing.

The first bug is that when I type some unexpected string in the expression, the Expression function will return an empty string. A very important thing is that the Expression function should never return a NONE or "" unless the input is empty string. If the equation is correct, it should return an interpreted function. If the equation is wrong, it should return a prompt of TypeError. But when I pass some unexpected characters in the expression, like '#', '@', the Expression function will return an empty string. It should be given a Type Error.

If the author of the library want to use empty string to indicate this Type Error. It is bad idea. The disadvantage of this problem is that when I type something unexpected characters in the expression, it won’t return error prompt and it will be very hard for me to find out why there is no meaningful output.

bugs1.tstl shows this bug.

Second Bug

The second bug is about complex number. According to the introduction of Equation library, it can deal with complex number when evaluating the expression. Here is an example from the instruction of the library.

```
>>> from Equation import Expression

>>> fn = Expression("sin(x+y^2)",["y","x"])

>>> fn

sin((x + (y ^ (2+0j))))

>>> print fn

\sin\left(\left(x + y^{(2+0j)}\right)\right)

>>> fn(3,4)

(0.42016703682664092+0j)
```

But when I try to test it with imaginary number in the expression. It doesn’t work correctly. Here is the bug that I meet.

```
>>> func = Expression("x+2j",["x"])

>>> func

(x + 2)

>>> func(1)

3
```

We can see that the imaginary number part ‘j’ is not interpreted. The Expression function just ignores it.

Since this way of presenting imaginary part number doesn’t work, I tried another way:

```
>>> func = Expression("x+2*j",["x"])

>>> func

(x + (2 * j))

>>> func(1)
```

I use ‘*’ to connect the number and the imaginary part symbol ‘j’. Then it will give a Type Error saying that j is not defined. That means ‘j’ is interpreted as a parameter but not the imaginary part of a complex number.

In a conclusion, there is no way for Equation library to understand the imaginary part of a complex number.

The interesting thing is that when I evaluate the equation by passing some value to parameters, I can pass complex number with imaginary part. And the library can interpret that. The bug I discussed above only happens when I type expression and let the library understand it.

The first bug which is about returning empty string is bug about how to handle the incorrect input. This bug makes the library not very stable. The second bug which indicates that the library cannot understand the imaginary number in the input. That is about how to deal with the correct input. All in all, the Equation library I tested can neither handle and deal with incorrect input nor interpret some correct input expression. I would say this library is not a very good library. It need a lot of work on it.

bugs2_1.tstl and bugs2_2.tstl shows this bug.

How does the tester work?

I use the TSTL to test the python library. It help me to find several bugs. And it help me to understand the process of testing. Also TSTL is easy to learn so I don’t need to remember a lot of grammar before I begin to write TSTL file. TSTL can run very fast to find bugs.

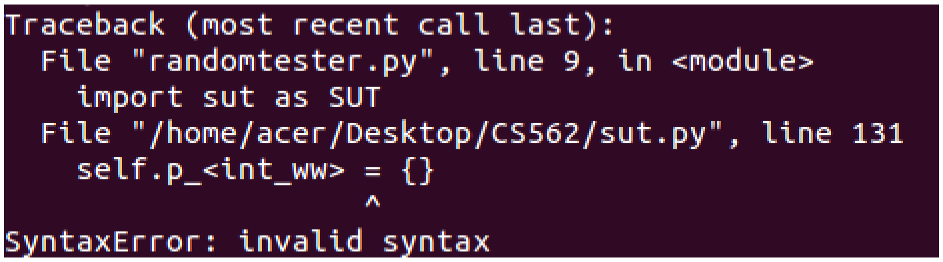
How is the library I test on?

Since I found two bugs in the Equation library, the first one is about when there is something unexpected character, it doesn’t give a prompt. The second one is more serious. This Equation library can only interpret the complex number when evaluating the expression but cannot understand when we input expression as string.

Something not perfect or something wrong with TSTL

TSTL is a very good tester. But also here is something not perfect with TSTL.

Firstly, TSTL cannot declare variables with '_' in the names. For example, if I use <int_ww> as a variable in the pool, when I run random tester file, it will give an error. The '_' is very useful when we are naming variables. It would be good if TSTL can accept "_" for names in the future. The error message is posted below when a name of variable including '_'.



```
Traceback (most recent call last):
  File "randomtester.py", line 9, in <module>
    import sut as SUT
  File "/home/acer/Desktop/CS562/sut.py", line 131
    self.p_<int_ww> = {}
                    ^
SyntaxError: invalid syntax
```

Secondly, in one of my version of TSTL testing code, because there are a lot of variables and the actions are complex, when I run tstl with tstl file, it runs very slow.

Something very well about TSTL

TSTL is a very easy to learn testing tool. I don't need to learn a lot before I start to use it. The grammar of TSTL is easy to understand. And I only need to write several lines of code in TSTL, like less than one hundred lines, to achieve the result of several hundred lines of pure python code. So TSTL decreases the amount of work of testing.

What's more, TSTL has very good prompt for different kind of error. Because TSTL is a tester for python, which means that it tests other python library. So when I run the TSTL, I may met the error that I made in the TSTL file. Also I will meet the error or bugs in the system under testing. If the prompt for different kinds of error is not clear. Then it will be hard for me to know whether the TSTL code is not correct or the library I am testing on is not correct.

Also TSTL can generate guards for testing. That is very good tool and design for users. Since reducing the structure of condition statement, the guards will make the structure of TSTL code very simple.

Coverage Discussion

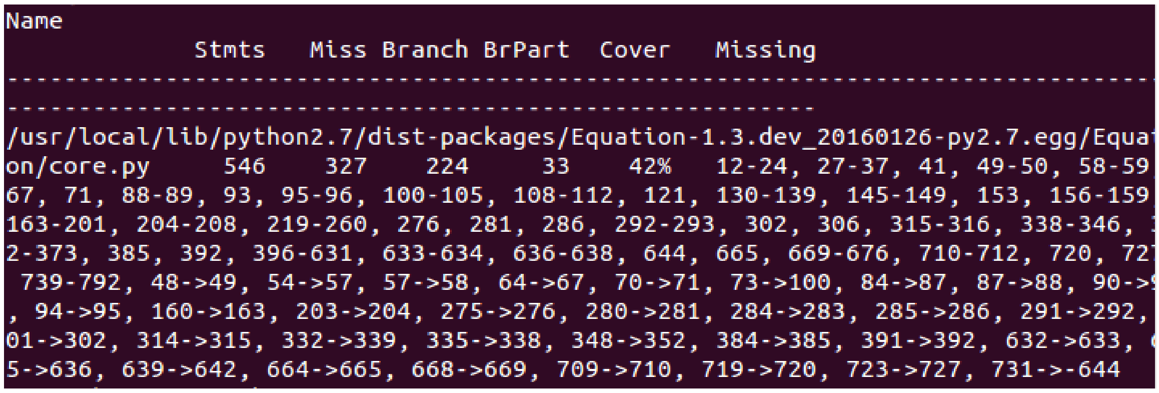
I have tested the coverage of my testing by using the coverage tool in TSTL. File cover.tstl is used for general testing and coverage analysis.

In order to use coverage analysis in TSTL, I type the following commands:

tstl cover.tstl

python randomtester.py --timeout=15 --depth=30

It will use coverage analysis in TSTL to analysis the testing. The coverage percent is 41.948.



```
Name
             Stmts   Miss Branch BrPart  Cover   Missing
-------------------------------------------------------------------------------
--------------------------------------------------------
/usr/local/lib/python2.7/dist-packages/Equation-1.3.dev_20160126-py2.7.egg/Equa
on/core.py     546    327    224     33    42%   12-24, 27-37, 41, 49-50, 58-59
67, 71, 88-89, 93, 95-96, 100-105, 108-112, 121, 130-139, 145-149, 153, 156-159
163-201, 204-208, 219-260, 276, 281, 286, 292-293, 302, 306, 315-316, 338-346, 
2-373, 385, 392, 396-631, 633-634, 636-638, 644, 665, 669-676, 710-712, 720, 72
 739-792, 48->49, 54->57, 57->58, 64->67, 70->71, 73->100, 84->87, 87->88, 90->
, 94->95, 160->163, 203->204, 275->276, 280->281, 284->283, 285->286, 291->292,
01->302, 314->315, 332->339, 335->338, 348->352, 384->385, 391->392, 632->633, 
5->636, 639->642, 664->665, 668->669, 709->710, 719->720, 723->727, 731->-644
```