

CS562 FINAL REPORT

xujing@oregonstate.edu



JINGYUAN XU

2016.3.14

EECS

INTRODUCTION / BACKGROUND

The library which I want to test named Examine, and you can download from this link: <https://github.com/Jazzer360/python-examine>. This library is the structure parser that allows users to easily see the resulting structure of python object. When we deal with a dataset, we need to make sure the dataset is in the right data type, data structure, and understand the relationship between the metadata and the dataset, in order to get the right result. Thus, I will use TSTL to test this library.

USEAGE OF THIS LIBRARY

we call the analysis function and output the data relation with a dataset. Such like:

input:

```
examine.Structure({'key1': {'subkey1': 'subvalue'}})
```

output:

```
dict
```

```
    key1 - dict
```

```
        subkey1 - str
```

Then I will use different kind of test cases to test the structure like the example.

For example, if the input dataset like this: [{'key1': {'subkey1': 'subvalue'}}], what kind of structure it will output. The correct format output should like this:

```
[dict]
```

```
    key1 - dict
```

```
        subkey1 - str
```

if we want to test the dataset has the parallel data structure like this: {'key1': 'val1'}, {'key2': 'val2'}

The correct format output should like this:

```
dict
```

```
    key1- dict
```

```
    subkey1 - str
```

.

LIST OF TEST CASES

Below are the test cases list in my test file. Each of them can be used in testing different situations while users use this library.

1. non structure
2. simple list
3. none list
4. mixed type list
5. simple dict
6. dictionaries with lists
7. blank list in dict
8. blank list
9. blank dict
10. defer list type if empty
11. empty list merged with non list
12. nested list
13. merging none list with int list
14. nested dict
15. add empty list to list of dicts
16. none in list
17. none as val of key in list of dicts
18. tuple
19. deep list nest
20. list of tuples
21. empty tuple merging

ACCOMPLISH

1. Follow the library usage to create tstl functions in testing this library.

```
GNU nano 2.0.6 File: examine.  
  
pool: <nestedlist> 3  
pool: <incosistentdepth> 3  
pool: <nesteddict> 3  
pool: <addeptylistdicts> 3  
pool: <noneinlist> 3  
pool: <nonekeydicts> 3  
pool: <tuple> 3  
pool: <deeplistnest> 3  
pool: <listuples> 3  
pool: <emptytuplemerging> 3  
  
actions:  
#test no structure  
pool: <strtest> 3  
<strtest> :=[]  
<strtest> +={{'key1': 'val1'}}  
<strtest> +=['coding', 'snowboarding']  
print(<strtest>,<strtest>,<strtest>,1,<strtest>,2>)  
  
# test simple list sturcture  
<strname> := ['coding', 'snowboarding']  
<list1> :={{'key1': 'val1'}, {'key2': 'val2'}}  
print examine.Structure(<list1>);examine.Structure(<list1>,1>)  
■
```

2. I implement the test cases list which I showed above in my TSTL file.
3. I do random test with random combining two different test cases together and observe the output result

```
#test mixed type list structure  
<datalist> := ['<list1>','<strname>']  
<exams> := ['<list1>']  
~<exams>.append(<datalist>)  
<strname> := ['coding', 'snowboarding']  
  
<exam1>:= examine.Structure(<strname>)  
<exam2>:= examine.Structure(<list1>)  
<exam3>:= examine.Structure(<exams>)  
  
check(<exam3>.__add__(<exam3>))  
check(<exam1>.__add__(<exam3>))
```

4. After testing this test cases, I test the code coverage of my tstl file.

TESTER WORKS/BUGS

I use randomtester.py in TSTL to do my test. After I run tstl command, it will build a “sut.py” file, then randomtester.py will executes this file to show the result. If the TSTL file has error in it, then the SUT.py will contains error (usually sytanx error), then randomtester can’t do testing unless generate a right TSTL file again.

Follow the design logic in the library I choose, examine.py can analysis the datature and show to users in a clear way. Because the main class in this library is Structure(), and it contains some important methods like: __add__(), copy(), type_string(), __str__(), these methods can all be tested by calling Structure class in the test cases. Thus my TSTL code will also follow this way to work. First I need to implement call examine() for test cases, make them work in library functions, then I use __add__() to combine different structures already store in examine(). Another way to combine two different test case together is use “append” in TSTL code. For example, I have a “list 1” have one data structure, then I have “strname” has another data structure. If I want to combine together, I can create a empty list “exams”, and put “list 1” in it. Then I can put another data structure “strname” in this list with “append”. If I want to test two combine list structure, I can print out the list “exams” instead of these two data structures:

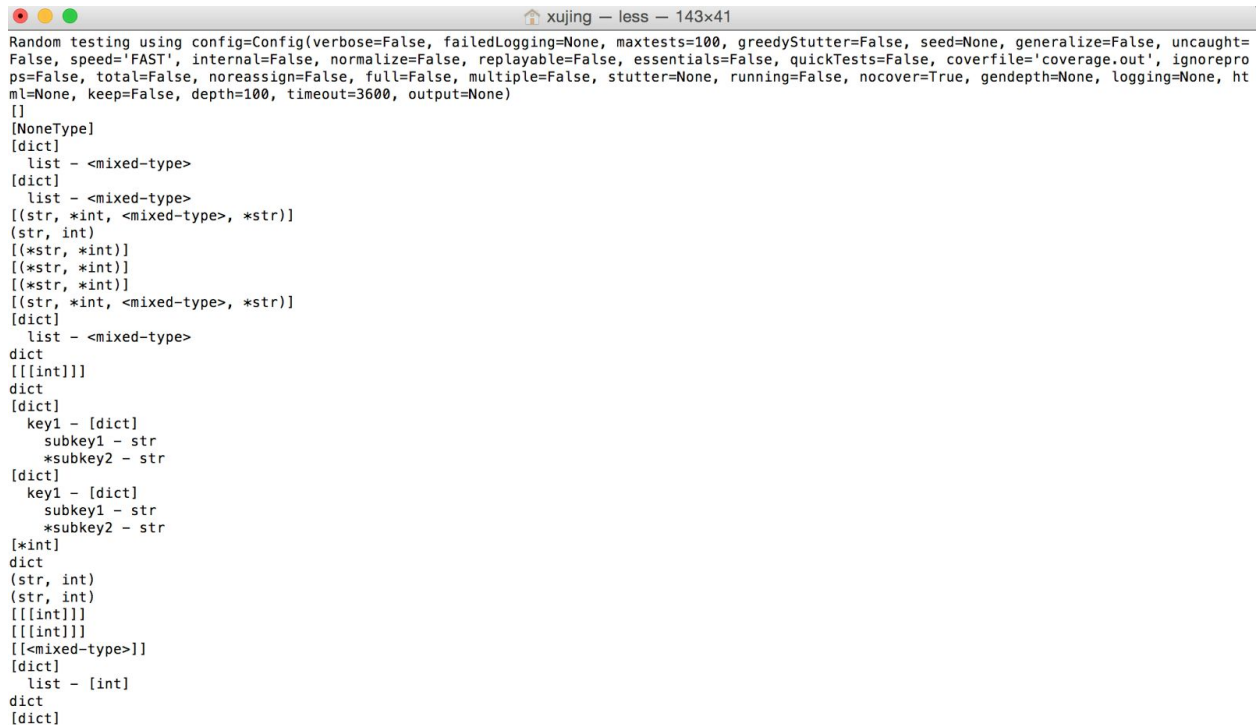
Code:

```
#test mixed type list structure
<datalist> := ['<list1>','<strname>']
<exams> := ['<list1>']
~<exams>.append(<datalist>)
```

This is a good library, the author did a lot of editing while he develop this library. Also he’s official document shows many test cases in python, which try to cover all the situations users may face in the process of using the library. So very unlucky I didn’t find out bugs in this library.

CONS AND PROS WITH TSTL

TSTL is very useful for random testing. Here is a screenshot of testing result:



```
Random testing using config=Config(verbose=False, failedLogging=None, maxtests=100, greedyStutter=False, seed=None, generalize=False, uncaught=False, speed='FAST', internal=False, normalize=False, replayable=False, essentials=False, quickTests=False, coverfile='coverage.out', ignorepropos=False, total=False, noreassign=False, full=False, multiple=False, stutter=None, running=False, nocover=True, gendepth=None, logging=None, htm=None, keep=False, depth=100, timeout=3600, output=None)
[]
[NoneType]
[dict]
  list - <mixed-type>
[dict]
  list - <mixed-type>
[(str, *int, <mixed-type>, *str)]
(str, int)
[(str, *int)]
[(str, *int)]
[(str, *int)]
[(str, *int)]
[(str, *int, <mixed-type>, *str)]
[dict]
  list - <mixed-type>
dict
[[[int]]]
dict
[dict]
  key1 - [dict]
    subkey1 - str
    *subkey2 - str
[dict]
  key1 - [dict]
    subkey1 - str
    *subkey2 - str
[*int]
dict
(str, int)
(str, int)
[[[int]]]
[[[int]]]
[[[int]]]
[[[<mixed-type>]]]
[dict]
  list - [int]
dict
[dict]
```

Also, if I type wrong test cases which not follow the syntax with `tstl`, it will also give me warning and lead me to the problem point. For example, in my test cases, if I didn't call `examine()` to analysis the structure first, then it is wrong in logic way. Such that if I run TSTL, it will give me a syntax error. Also, when I'm a beginner of TSTL, sometimes I forget to use `Pool` as an initialization, then I also will get a syntax error.

However some problems I faced when I use TSTL are:

- Coverage warnings.

If I didn't use `--nocover` this command in `tstl` command, then my test cases may combine lots of no cover warnings: *Coverage.py warning: No data was collected*. But lucky this problem may already fixed with the new version of TSTL for this python version.

- TSTL now works on python 2, how about python 3?
- The tutorials of TSTL:

I hope to see more guidance in a handbook of TSTL, instead of avlnew.tstl. That file is awesome, with lots of comments, but if TSTL has an easy version handbook instead of a TSTL file, that's very nice.

COVERAGE SUMMARY

When I run coverage command, I got this output file to show the stats of my code coverage:

```
10-249-19-51:exam root# coverage report
Name                                                    Stmts   Miss  Cover
-----
/Library/Python/2.7/site-packages/examine/__init__.py      1        0   100%
/Library/Python/2.7/site-packages/examine/examine.py     145        5    97%
/private/var/root/tstl/generators/randomtester.py       582     442    24%
/private/var/root/tstl/generators/sut.py               4246     660    84%
-----
TOTAL                                                    4974    1107    78%
```

Which seems good for my code coverage, with this details of my tstl file running analysis:

```
0.40921998024 TOTAL RUNTIME
100 EXECUTED
10000 TOTAL TEST OPERATIONS
0.240902662277 TIME SPENT EXECUTING TEST OPERATIONS
0.104068279266 TIME SPENT EVALUATING GUARDS AND CHOOSING ACTIONS
0.0060818195343 TIME SPENT CHECKING PROPERTIES
0.246984481812 TOTAL TIME SPENT RUNNING SUT
0.0287914276123 TIME SPENT RESTARTING
0.0 TIME SPENT REDUCING TEST CASES
(END)
```

CONCLUSION

Using TSTL is a nice experience for me. It shows the result of code coverage, random test. TSTL also has other test functions, and I hope I can use them in the next term. This test tool give me a nice begining in writing test case, rather than JUNIT which I used before, this one has the similar method while I used it to write test cases, but gives more interesting outputs. The new knowledge for me is code coverage, and by these command, I can see the analysis with this test system, which add a new experience in

my testing history.