Instrumental Documentation

Release 0.5.1

Matthew J Desmarais

February 08, 2016

Contents

Contents:

CHAPTER 1

Introduction

1.1 What is instrumental?

instrumental is a structural coverage reporting tool. instrumental can tell you which parts of your program have been executed after it has been run. You can use this information to verify that your tests have executed your code and to determine which code has not been tested.

1.2 What's the problem?

Testing is hard. Writing test cases is the easy part (and it isn't always that easy). The hard part is determining which tests to write and which inputs to choose when you write those tests.

Let's say that your code contains an ''and'' decision that comprises three conditions as inputs. Maybe your decision looks like this:

```
(a == 4) and (b > 2) and c
```

We'll assume that the three conditions (a == 4, b > 2, and c (is not False)) are significant. This should be a safe assumption to make; if those conditions don't significantly affect the result of your program then they probably shouldn't be in the code in the first place.

Since these are significant conditions, it is important to test them thoroughly. This means ensuring that there are tests that depend on each of the conditions to force the decision result to a predictable value. The way to do this is, when testing a particular condition, to hold the other conditions that can affect the result of the decision to particular values.

These ideas should be starting to feel familiar. This is what we do when we write unit tests; we isolate individual units so that we can verify their behaviour without having to worry about any other code confounding the results of our testing. So think of this as testing units within units. It isn't possible to break the and decision into smaller functional units, but the other conditions can be effectively removed from the equation by holding them constant.

So then to test our first condition in our decision, a == 4, we'll need to write at least two cases. We need one case in which a is 4 and one in which it isn't. Further, we need to neutralize the other conditions. In the case of an and decision we do this by holding the other input conditions to True. This allows us to prove that if the result of the decision is True, then it must have been a taking a value of 4 that caused it. We can similarly prove that if the result of the decision is False then it must have been a taking a value other than 4 that was the cause. The following table illustrates input selections that do just this:

a	b	c
4	3	True
5	3	True

Inputs that fully test this decision might look like this:

a	b	c
4	3	True
5	3	True
4	2	True
4	3	False

If we display this as the result of evaluating the individual conditions, we get a table that looks like this:

True	True	True
False	True	True
True	False	True
True	True	False

You can see that we need a case where all conditions are True, and then we 'walk' the False across the conditions to get cases where the significant condition forces the parent decision False.

This is a simple case, but you can see that the kind of analysis involved here is non-trivial. When you extrapolate this to a realistic program, the analysis quickly becomes prohibitively time-consuming and tedious. This is where instrumental can help.

1.3 What does instrumental do?

instrumental executes your program for you. Upon completion, it can produce a report indicating whether or not the significant cases for a decision were executed.

1.4 What doesn't instrumental do?

instrumental doesn't currently provide branch coverage. We are unlikely to provide branch coverage any time soon since we provide decision coverage, a superset of branch coverage.

Using instrumental

2.1 Running instrumental

instrumental provides a command that will run your Python code in an environment in which it can measure code execution characteristics. Using it looks like this:

```
$ instrumental <path to your python script>
```

When you run your code this way, it should run and exit as normal. instrumental will have gathered coverage information, but because you haven't asked for a report it won't tell you anything about it.

2.2 Statement coverage

Try this:

```
$ instrumental -S -t <packagename> <path to your python script>
```

The '-S' flag indicates that you want to see a statement coverage report and the '-t' flag tells instrumental that you want the package *packagename* instrumented and included in the coverage report.

While developing instrumental, I often run it against the pyramid tests (since they keep their coverage levels high). That looks something like this:

```
$ instrumental -S -t pyramid -i pyramid.tests `which nosetests`
```

There I'm telling instrumental to run nosetests targetting the 'pyramid' package, ignoring the 'pyramid.tests' package, and producing a statement coverage report.The results look something like this:

```
=======================================
Instrumental Statement Coverage Summary
=======================================

Name                          Stmts   Miss  Cover   Missing
-----------------------------------------------------------
pyramid                           5      0   100%
pyramid.asset                    30      0   100%
pyramid.authentication          325      0   100%
...
pyramid.view                    103      0   100%
pyramid.wsgi                     14      0   100%
-----------------------------------------------------------
TOTAL                          6419      0   100%
```

If there are statements that weren't executed during the instrumented run, instrumental will report their line numbers in the 'Missing' column of the statement coverage report.

2.3 Condition / decision coverage

instrumental also aims to provide more rigorous forms of code coverage. Try running instrumental like this:

```
$ instrumental -r -t pyramid -i pyramid.tests `which nosetests`
```

Invoking instrumental this way executes your code and provides you with a condition/decision coverage report when execution is complete. The output should look something like this:

```
===============================================
Instrumental Condition/Decision Coverage Report
===============================================

Decision -> pyramid.authentication:43 < logger >

T ==> X
F ==>

...

Decision -> pyramid.view:31 < (package_name is None) >

T ==> X
F ==>

LogicalOr -> pyramid.view:281 < (getattr(request, 'exception', None) or context) >

T * ==> X
F T ==>
F F ==>
```

The preceeding output is formatted like this:

```
<construct type> -> <modulename>.<line number> < <source code> >

<description of result>
```

So in the example report above, the first chunk tells us that for the decision on line 43 of pyramid.authentication, the False case was never executed. So we can say that test, evaluating "logger", never ran when "logger" evaluated not-True. Likewise, the second chunk tells us that the code at line 31 in pyramid.view was never executed when package_name was not None.

The third chunk in the example report describes the condition coverage for the logical or on line 281 of pyramid.view. The result there says that every time the or decision was executed, the expression on the left side always evaluated True as a boolean expression. So the expression on the right side of the or, 'context', was never Tested!

2.4 Unreachable Cases

There are times when you may include a literal in an expression. It may be in an assignment (as below) or in the test for an if statement. Instrumental doesn't judge. If Instrumental finds a decision that contains one or more literals it won't, by default, count against coverage totals the cases that are directly unreachable because of a literal. On the other hand, Instrumental will hold you responsible for cases that are unreachable due to the presence of literals proceeding them in the calcuation of the decision. A concrete example of this would probably be helpful here.

Note that in the following report section, the literal value '/' is used as the second condition in the decision. So the 'F F' condition combination will never be possible and is considered unreachable. In this situation, Instrumental will only expect the 'T *' and 'F T' condition combinations to be reported as covered. As long as those combinations are seen during execution, this decision is considered to be fully covered and it will not be present in the coverage report.:

```
LogicalOr -> pyramid.view:277.1 < (request.environ['PATH_INFO'] or '/') >

** One or more condition combinations may not be reachable due to the presence of a literal in the de

T * ==> X
F T ==> X
F F ==> U
```

Now consider the decision described in the next report chunk. This is the same expression as the one in the last example except that it has an additional condition at the end. In this situation, Instrumental will recognize that the 'F F T' and 'F F F' combinations are unreachable, but only the 'F F F' combination is marked as unreachable and so exempted from coverage. This is because the literal in the expression prevents the final condition, 'default_path()', from ever being evaluated. Since this represents a possible bug, Instrumental will report it as missed coverage.:

```
LogicalOr -> pyramid.view:277.1 < (request.environ['PATH_INFO'] or '/' or default_path()) >

** One or more condition combinations may not be reachable due to the presence of a literal in the de

T * * ==> X
F T * ==> X
F F T ==>
F F F ==> U
```

If you're interested in expressions that contain literals, you can always use the –report-literals option. This option tells Instrumental to count cases that are unreachable due to the presence of literals during coverage calculation.

2.5 Marking conditions as unreachable

It may be that there are condition combinations that aren't possible, but appear possible to Instrumental. In this situation you can tell Instrumental to not expect to find certain condition combinations using the 'pragma: no cond' directive. Let's look at a concrete example:

```
[1] a = func1()
[2] b = False
[3] c = func2()
[4] if a or b or c: # pragma: no cond(F T F)
[5]     func3()
[6] else:
[7]     func4()
```

In this example we can see that the 'F T F' case will not ever be possible since b will always be False. We can communicate this to instrumental by adding a comment to the end of line 4 in the form "pragma: no cond(<condition1>[,condition2, ..., conditionN])". When we do that, Instrumental will output something like the following:

```
LogicalOr -> somemodule:4.1 < (a or b or c) >

T * * ==>
F T * ==> P
```

```
F  F  T ==>
F  F  F ==>
```

In this report chunk, the ‘P’ indicates that the ‘F T F’ condition combination has been marked as impossible by a pragma. If we wanted to also say that the ‘F F T’ case we impossible, our pragma would look more like, “pragma: no cond(F T F,F F T)”. The “pragma: no cond” system also supports nested expressions. Consider the following modified code:

```
[1] a = func1()
[2] b = True
[3] c = func2()
[4] if a or (b and d) or c: # pragma: no cond[.2](F T)
[5]     func3()
[6] else:
[7]     func4()
```

Here we can see that the impossible condition will be the ‘F T’ combination in the nested ‘and’. You can indicate that the pragma applies to the nested ‘and’ by specifying a “selector” of [.2]. The .2 will match the label that Instrumental will give the expression (i.e. 4.2) and Instrumental will know which expression to apply the pragma to. You can even add the pragma on a separate line and specify a selector that contains a line number. In this case, you could add the comment, “# pragma: no cond[4.2](F T)” to line 3 and Instrumental would figure out that the pragma should be applied to the expression labeled 4.2.

2.6 Excluding expressions from instrumentation

In order to do the things it does, Instrumental takes some liberties with your code. This doesn’t always work out very well with a language as dynamic as Python. Comparison operations are a good case to loo at. Instrumental, by default, attempts to detect comparisons and modify them so that it can measure the result of their executions as either True or False. But Python allows you to replace the semantics of comparisons with your own if you’d like. So comparisons may not evaluate to True or False at all. It is for this case that Instrumental provides –ignore-comparisons. Specifying the –ignore-comparisons option on the command-line tells instrumental to not touch comparisons at all. So you’ll lose the ability to measure the execution of comparisons, but at least they won’t raise exceptions or give you other problems.

Instrumental also instruments and reports on the expressions in assertions by default. This can result in noisy missed condition reports since the expressions evaluated in the context of assertions are usually expected to be True for all cases. This is why Instrumental provides the –ignore-assertions option. Specifying –ignore-assertions on the command-line tells Instrumental to leave those assertions alone and not report on the results of evaluating them.

2.7 Gathering coverage over multiple runs

In some cases, running your tests may mean running several actualy test runs. Then you’ll want to produce a coverage result for each run and combine then all into one file that you can report on. Instrumental now support this!

You can now run instrumental with the -l (or –label) option turned on. Instrumental will give the coverage file it produces a (probably) unique filename each time it runs. Then you can use the instrumental-tools to combine the coverage files into one file that you can report on.

Here’s an example session illustrating the use of the -l option and instrumental-tools command:

```
$ instrumental -l -t yourpackage test_things.py
$ instrumental -l -t yourpackage test_more_things.py
$ ls -a1
.
```

```
    ..
    .instrumental.p1111.cov
    .instrumental.p2222.cov
     ...
    yourpackage
[4] $ instrumental-tools combine .instrumental.cov .instrumental.p*.cov
[5] $ instrumental -r
```

Commands [1] and [2] each run some tests in a different process. Command [3] checks to see what the names of the created coverage files are. We can see that the files .instrumental.p1111.cov and .instrumental.p2222.cov were created. Command [4] uses the instrumental-tools combine command to combine coverage files matching the pattern *.instrumental.p*.cov* into the file *.instrumental.cov*. Then command [5] runs instrumental and asks for a report on the coverage file. Command [5] is a little tricky since it counts on the generated coverage file being called *.instrumental.cov* because that's the default coverage file name. If we had specified a different coverage file name in command [4], like my.cov, then command [5] would look more like:

```
[5] $ instrumental -f my.cov -r
```

CHAPTER 3

Compatibility

3.1 coverage.py

There is one major code coverage tool available in the Python world, coverage.py. coverage.py is a powerful and mature tool that provides both statement and branch coverage information for almost any running Python code. It's a great tool; you should use it if you don't use instrumental and maybe even if you do.

instrumental aims to be compatible with coverage.py use patterns wherever possible. Currently this means supporting the '# pragma: no cover' tag, producing a cobertura-compatible xml coverage report, and producing a colorful html coverage report. So theoretically, instrumental should be easy to fit into a process that currently uses coverage.py.

Indices and tables

- genindex
- modindex
- search