PyMOTW-3

trace — Follow Program Flow

Purpose: Monitor which statements and functions are executed as a program runs to produce coverage and callgraph information.

The trace module is useful for understanding the way a program runs. It watches the statements executed, produces coverage reports, and helps investigate the relationships between functions that call each other.

Example Program

This program will be used in the examples in the rest of the section. It imports another module called recurse and then runs a function from it.

```
# trace example/main.py
from recurse import recurse
def main():
    print('This is the main program.')
    recurse(2)
   name == ' main ':
   main()
```

The recurse() function invokes itself until the level argument reaches 0.

```
# trace example/recurse.py
def recurse(level):
    print('recurse({})'.format(level))
    if level:
        recurse(level - 1)
def not called():
    print('This function is never called.')
```

Tracing Execution

It is easy use trace directly from the command line. The statements being executed as the program runs are printed when the --trace option is given. This example also ignores the location of the Python standard library to avoid tracing into importlib and other modules that might be more interesting in another example, but that clutter up the output in this simple example.

```
$ python3 -m trace --ignore-dir=.../lib/python3.7 \
--trace trace example/main.py
 --- modulename: main, funcname: <module>
main.py(7): """
main.py(10): from recurse import recurse
--- modulename: recurse, funcname: <module>
recurse.py(7): """
recurse.py(11): def recurse(level):
recurse.py(17): def not_called():
main.py(13): def main():
main.py(18): if __name__ == '__main__':
main.py(19):
                 main()
--- modulename: main, funcname: main
main.pv(14):
                 print('This is the main program.')
```

```
p. -... , ...- -- -.. ... ... p. -3. .... ,
This is the main program.
main.py(15):
              recurse(2)
--- modulename: recurse, funcname: recurse
                    print('recurse({})'.format(level))
recurse.py(12):
recurse(2)
recurse.py(13):
                    if level:
recurse.py(14):
                        recurse(level - 1)
--- modulename: recurse, funcname: recurse
                    print('recurse({})'.format(level))
recurse.py(12):
recurse(1)
                    if level:
recurse.py(13):
recurse.py(14):
                        recurse(level - 1)
--- modulename: recurse, funcname: recurse
                    print('recurse({})'.format(level))
recurse.py(12):
recurse(0)
                    if level:
recurse.py(13):
```

The first part of the output shows the setup operations performed by trace. The rest of the output shows the entry into each function, including the module where the function is located, and then the lines of the source file as they are executed. recurse() is entered three times, as expected based on the way it is called in main().

Code Coverage

Running trace from the command line with the --count option will produce code coverage report information, detailing which lines are run and which are skipped. Since a complex program is usually made up of multiple files, a separate coverage report is produced for each. By default the coverage report files are written to the same directory as the module, named after the module but with a .cover extension instead of .py.

```
$ python3 -m trace --count trace_example/main.py
This is the main program.
recurse(2)
recurse(1)
recurse(0)
```

Two output files are produced, trace_example/main.cover:

and trace example/recurse.cover:

```
# trace_example/recurse.cover

1: def recurse(level):
    3:    print('recurse({})'.format(level))
    3:    if level:
    2:    recurse(level - 1)

1: def not_called():
        print('This function is never called.')
```

Note

Although the line def recurse(level): has a count of 1, that does not mean the function was only run once. It means the function definition was only executed once. The same applies to def not called():, because the function

definition is evaluated even though the function itself is never called.

It is also possible to run the program several times, perhaps with different options, to save the coverage data and produce a combined report. The first time trace is run with an output file, it reports an error when it tries to load any existing data to merge with the new results before creating the file.

```
$ python3 -m trace --coverdir coverdir1 --count \
--file coverdir1/coverage report.dat trace example/main.py
This is the main program.
recurse(2)
recurse(1)
recurse(0)
Skipping counts file 'coverdir1/coverage report.dat': [Errno 2]
No such file or directory: 'coverdir1/coverage_report.dat'
$ python3 -m trace --coverdir coverdir1 --count \
--file coverdir1/coverage report.dat trace example/main.py
This is the main program.
recurse(2)
recurse(1)
recurse(0)
$ python3 -m trace --coverdir coverdir1 --count \
--file coverdir1/coverage report.dat trace example/main.py
This is the main program.
recurse(2)
recurse(1)
recurse(0)
$ ls coverdir1
coverage report.dat
main.cover
recurse.cover
```

To produce reports once the coverage information is recorded to the .cover files, use the --report option.

```
$ python3 -m trace --coverdir coverdir1 --report --summary \
--missing --file coverdir1/coverage_report.dat \
trace_example/main.py

lines cov% module (path)
   7 100% trace_example.main (trace_example/main.py)
   7 85% trace_example.recurse
(trace_example/recurse.py)
```

Since the program ran three times, the coverage report shows values three times higher than the first report. The --summary option adds the percent covered information to the output. The recurse module is only 87% covered. Looking at the cover file for recurse shows that the body of not _called is indeed never run, indicated by the >>>>> prefix.

```
# coverdir1/trace_example.recurse.cover

3: def recurse(level):
9:    print('recurse({})'.format(level))
9:    if level:
6:     recurse(level - 1)

3: def not_called():
>>>>>    print('This function is never called.')
```

Calling Relationships

In addition to coverage information, trace will collect and report on the relationships between functions that call each other.

For a simple list of the functions called, use --listfuncs.

```
$ python3 -m trace --listfuncs trace_example/main.py | \
grep -v importlib

This is the main program.
recurse(2)
recurse(1)
recurse(0)

functions called:
filename: trace_example/main.py, modulename: main, funcname:
<module>
filename: trace_example/main.py, modulename: main, funcname:
main
filename: trace_example/recurse.py, modulename: recurse,
funcname: <module>
filename: trace_example/recurse.py, modulename: recurse,
funcname: recurse
```

For more details about who is doing the calling, use --trackcalls.

```
$ python3 -m trace --listfuncs --trackcalls \
trace example/main.py | grep -v importlib
This is the main program.
recurse(2)
recurse(1)
recurse(0)
calling relationships:
*** .../lib/python3.7/trace.py ***
  --> trace example/main.py
    trace.Trace.runctx -> main.<module>
  --> trace_example/recurse.py
*** trace example/main.py ***
    main.<module> -> main.main
  --> trace_example/recurse.py
    main.main -> recurse.recurse
*** trace example/recurse.py ***
    recurse.recurse -> recurse.recurse
```

Note

Neither --listfuncs nor --trackcalls honors the --ignore-dirs or --ignore-mods arguments, so part of the output from this example is stripped using grep instead.

Programming Interface

For more control over the trace interface, it can be invoked from within a program using a Trace object. Trace supports setting up fixtures and other dependencies before running a single function or executing a Python command to be traced.

```
# trace_run.py
import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=False, trace=True)
tracer.run('recurse(2)')
```

Since the example only traces into the recurse() function, no information from main.py is included in the output.

```
$ python3 trace run.py
 --- modulename: trace run, funcname: <module>
<string>(1): --- modulename: recurse, funcname: recurse
recurse.py(12):
                    print('recurse({})'.format(level))
recurse(2)
                    if level:
recurse.py(13):
recurse.py(14):
                        recurse(level - 1)
--- modulename: recurse, funchame: recurse
recurse.py(12):
                   print('recurse({})'.format(level))
recurse(1)
                    if level:
recurse.py(13):
                        recurse(level - 1)
recurse.py(14):
--- modulename: recurse, funchame: recurse
                    print('recurse({})'.format(level))
recurse.py(12):
recurse(0)
                    if level:
recurse.py(13):
```

That same output can be produced with the runfunc() method, too.

```
# trace_runfunc.py
import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=False, trace=True)
tracer.runfunc(recurse, 2)
```

runfunc() accepts arbitrary positional and keyword arguments, which are passed to the function when it is called by the tracer.

```
$ python3 trace runfunc.py
 --- modulename: recurse, funcname: recurse
recurse.py(12):
                    print('recurse({})'.format(level))
recurse(2)
                    if level:
recurse.py(13):
                        recurse(level - 1)
recurse.py(14):
 --- modulename: recurse, funcname: recurse
                    print('recurse({})'.format(level))
recurse.py(12):
recurse(1)
recurse.py(13):
                    if level:
recurse.py(14):
                        recurse(level - 1)
--- modulename: recurse, funcname: recurse
                    print('recurse({})'.format(level))
recurse.py(12):
recurse(0)
                    if level:
recurse.py(13):
```

Saving Result Data

Counts and coverage information can be recorded as well, just as with the command line interface. The data must be saved explicitly, using the CoverageResults instance from the Trace object.

```
# trace_CoverageResults.py
import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=True, trace=False)
tracer.runfunc(recurse, 2)

results = tracer.results()
results.write_results(coverdir='coverdir2')
```

This example saves the coverage results to the directory coverdir2.

```
$ python3 trace_CoverageResults.py
```

```
recurse(2)
recurse(1)
recurse(0)

$ find coverdir2

coverdir2
coverdir2/trace_example.recurse.cover
```

The output file contains

```
#!/usr/bin/env python
      # encoding: utf-8
      # Copyright (c) 2008 Doug Hellmann All rights reserved.
      #
       0.00
>>>>>
      #end pymotw header
>>>>> def recurse(level):
    3: print('recurse({})'.format(level))
   3:
          if level:
              recurse(level - 1)
    2:
>>>>> def not called():
          print('This function is never called.')
>>>>>
```

To save the counts data for generating reports, use the infile and outfile arguments to Trace.

Pass a filename to infile to read previously stored data, and a filename to outfile to write new results after tracing. If infile and outfile are the same, it has the effect of updating the file with cumulative data.

```
$ python3 trace_report.py

recurse(2)
recurse(1)
recurse(0)
lines cov% module (path)
    7 42% trace_example.recurse
(.../trace_example/recurse.py)
```

Options

The constructor for Trace takes several optional parameters to control runtime behavior.

count

Boolean. Turns on line number counting. Defaults to True. countfuncs

Boolean. Turns on list of functions called during the run. Defaults to False.

countcallers

Boolean. Turns on tracking for callers and callees. Defaults to False.

ignoremods

Sequence. List of modules or packages to ignore when tracking coverage. Defaults to an empty tuple.

ignoredirs

Sequence. List of directories containing modules or packages to be ignored. Defaults to an empty tuple.

infile

Name of the file containing cached count values. Defaults to None.

outfile

Name of the file to use for storing cached count files. Defaults to None, and data is not stored.

See also

- Standard library documentation for trace
- <u>Tracing a Program As It Runs</u> The sys module includes facilities for adding a custom tracing function to the interpreter at run-time.
- <u>coverage.py</u> Ned Batchelder's coverage module.
- figleaf Titus Brown's coverage application.

Q<u>unittest — Automated Testing Framework</u>

<u>traceback — Exceptions and Stack Traces</u>

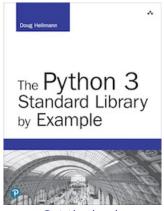
Quick Links

Example Program
Tracing Execution
Code Coverage
Calling Relationships
Programming Interface
Saving Result Data
Options

This page was last updated 2018-12-09.

Navigation

unittest — Automated Testing Framework
traceback — Exceptions and Stack Traces

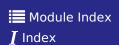


Get the book

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site





© Copyright 2019, Doug Hellmann



Other Writing



