Tracing a Program As It Runs

There are two ways to inject code to watch a program run: *tracing* and *profiling*. They are similar, but intended for different purposes and so have different constraints. The easiest, but least efficient, way to monitor a program is through a *trace hook*, which can be used to write a debugger, monitor code coverage, or achieve many other purposes.

The trace hook is modified by passing a callback function to `sys.settrace()`. The callback will receive three arguments: the stack frame from the code being run, a string naming the type of notification, and an event-specific argument value. the table below lists the seven event types for different levels of information that occur as a program is being executed.

Event Hooks for settrace()

Event	When it occurs	Argument value
call	Before a line is executed	None
line	Before a line is executed	None
return	Before a function returns	The value being returned
exception	After an exception occurs	The (exception, value, traceback) tuple
c_call	Before a C function is called	The C function object
c_return	After a C function returns	None
c_exception	After a C function throws an error	None

Tracing Function Calls

A `call` event is generated before every function call. The frame passed to the callback can be used to find out which function is being called and from where.

```
# sys_settrace_call.py
```

```
#!/usr/bin/env python3
# encoding: utf-8

import sys


def trace_calls(frame, event, arg):
    if event != 'call':
        return
    co = frame.f_code
    func_name = co.co_name
    if func_name == 'write':
        # Ignore write() calls from printing
        return
    func_line_no = frame.f_lineno
    func_filename = co.co_filename
    if not func_filename.endswith('sys_settrace_call.py'):
        # Ignore calls not in this module
        return
    caller = frame.f_back
    caller_line_no = caller.f_lineno
    caller_filename = caller.f_code.co_filename
    print('* Call to', func_name)
    print('*  on line {} of {}'.format(
        func_line_no, func_filename))
    print('*  from line {} of {}'.format(
        caller_line_no, caller_filename))
    return


def b():
```

```
def b():
    print('inside b()\n')


def a():
    print('inside a()\n')
    b()


sys.settrace(trace_calls)
a()
```

This example ignores calls to `write()`, as used by `print` to write to `sys.stdout`.

```
$ python3 sys_settrace_call.py

* Call to a
*  on line 35 of sys_settrace_call.py
*  from line 41 of sys_settrace_call.py
inside a()

* Call to b
*  on line 31 of sys_settrace_call.py
*  from line 37 of sys_settrace_call.py
inside b()
```

Tracing Inside Functions

The trace hook can return a new hook to be used inside the new scope (the *local* trace function). It is possible, for instance, to control tracing to only run line-by-line within certain modules or functions.

```
# sys_settrace_line.py
```

```
#!/usr/bin/env python3
# encoding: utf-8

import functools
import sys


def trace_lines(frame, event, arg):
    if event != 'line':
        return
    co = frame.f_code
    func_name = co.co_name
    line_no = frame.f_lineno
    print('*  {} line {}'.format(func_name, line_no))


def trace_calls(frame, event, arg, to_be_traced):
    if event != 'call':
        return
    co = frame.f_code
    func_name = co.co_name
    if func_name == 'write':
        # Ignore write() calls from printing
        return
    line_no = frame.f_lineno
    filename = co.co_filename
    if not filename.endswith('sys_settrace_line.py'):
        # Ignore calls not in this module
        return
    print('* Call to {} on line {} of {}'.format(
        func_name, line_no, filename))
    if func_name in to_be_traced:
        # Trace into this function
        return trace_lines
    return
```

```
def c(input):
    print('input =', input)
    print('Leaving c()')


def b(arg):
    val = arg * 5
    c(val)
    print('Leaving b()')


def a():
    b(2)
    print('Leaving a()')


tracer = functools.partial(trace_calls, to_be_traced=['b'])
sys.settrace(tracer)
a()
```

In this example, the list of functions is kept in the variable :py``to_be_traced``, so when `trace_calls()` runs it can return `trace_lines()` to enable tracing inside of `b()`.

```
$ python3 sys_settrace_line.py

* Call to a on line 49 of sys_settrace_line.py
* Call to b on line 43 of sys_settrace_line.py
*  b line 44
*  b line 45
* Call to c on line 38 of sys_settrace_line.py
input = 10
Leaving c()
*  b line 46
Leaving b()
Leaving a()
```

Watching the Stack

Another useful way to use the hooks is to keep up with which functions are being called, and what their return values are. To monitor return values, watch for the `return` event.

```
# sys_settrace_return.py
```

```
#!/usr/bin/env python3
# encoding: utf-8

import sys


def trace_calls_and_returns(frame, event, arg):
    co = frame.f_code
    func_name = co.co_name
    if func_name == 'write':
        # Ignore write() calls from printing
        return
    line_no = frame.f_lineno
    filename = co.co_filename
    if not filename.endswith('sys_settrace_return.py'):
        # Ignore calls not in this module
        return
    if event == 'call':
        print('* Call to {} on line {} of {}'.format(
            func_name, line_no, filename))
        return trace_calls_and_returns
    elif event == 'return':
        print('* {} => {}'.format(func_name, arg))
    return

```

```
26
27 def b():
28     print('inside b()')
29     return 'response_from_b '
30
31
32 def a():
33     print('inside a()')
34     val = b()
35     return val * 2
36
37
38 sys.settrace(trace_calls_and_returns)
39 a()
```

The local trace function is used for watching return events, so `trace_calls_and_returns()` needs to return a reference to itself when a function is called, so the return value can be monitored.

```
$ python3 sys_settrace_return.py

* Call to a on line 32 of sys_settrace_return.py
inside a()
* Call to b on line 27 of sys_settrace_return.py
inside b()
* b => response_from_b
* a => response_from_b response_from_b
```

Exception Propagation

Exceptions can be monitored by looking for the `exception` event in a local trace function. When an exception occurs, the trace hook is called with a tuple containing the type of exception, the exception object, and a traceback object.

```
# sys_settrace_exception.py
```

```
1  #!/usr/bin/env python3
2  # encoding: utf-8
3
4  import sys
5
6
7  def trace_exceptions(frame, event, arg):
8      if event != 'exception':
9          return
10     co = frame.f_code
11     func_name = co.co_name
12     line_no = frame.f_lineno
13     exc_type, exc_value, exc_traceback = arg
14     print(('* Tracing exception:\n'
15            '* {} "{}"\n'
16            '* on line {} of {}\n').format(
17                exc_type.__name__, exc_value, line_no,
18                func_name))
19
20
21 def trace_calls(frame, event, arg):
22     if event != 'call':
23         return
24     co = frame.f_code
25     func_name = co.co_name
26     if func_name in TRACE_INTO:
27         return trace_exceptions
28
29
30 def c():
31     raise RuntimeError('generating exception in c()')
32
33
34 def b():
```

```
    c()
    print('Leaving b()')


def a():
    b()
    print('Leaving a()')


TRACE_INTO = ['a', 'b', 'c']

sys.settrace(trace_calls)
try:
    a()
except Exception as e:
    print('Exception handler:', e)
```

Take care to limit where the local function is applied because some of the internals of formatting error messages generate, and ignore, their own exceptions. Every exception is seen by the trace hook, whether the caller catches and ignores it or not.

```
$ python3 sys_settrace_exception.py

* Tracing exception:
* RuntimeError "generating exception in c()"
* on line 31 of c

* Tracing exception:
* RuntimeError "generating exception in c()"
* on line 35 of b

* Tracing exception:
* RuntimeError "generating exception in c()"
* on line 40 of a

Exception handler: generating exception in c()
```

See also

- `profile` – The `profile` module documentation shows how to use a ready-made profiler.
- `trace` – The `trace` module implements several code analysis features.
- Types and Members – The descriptions of frame and code objects and their attributes.
- Tracing python code – Another `settrace()` tutorial.
- Wicked hack: Python bytecode tracing – Ned Batchelder's experiments with tracing with more granularity than source line level.
- smiley – Python Application Tracer

Modules and Imports

os — Portable access to operating system specific features

Quick Links

Tracing Function Calls
Tracing Inside Functions
Watching the Stack
Exception Propagation

This page was last updated 2018-03-18.

Navigation

Modules and Imports
os — Portable access to operating system specific features

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

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example