

Tracing in Python

The basics

- Python can call a function for every line executed

```
def mytracer(frame, event, args):  
    ...
```

```
import sys  
sys.settrace(mytracer)
```

- There are 4 kinds of events: `call`, `line`, `return`, `exception`

(don't believe the bs in docs about `c_call`, `c_return` and `c_exception` - those are only used for profiling via the similar `sys.setprofile`)

- The frame object contains information about the code being run (module, filename location and call stack)
- The `arg` will have the return or exception value (if event is `return` or `exception`)

A basic tracer

An almost useful tracer:

```
def dumbtrace(frame, event, args):
    sys.stdout.write("%015s:%-3s %09s %s" % (
        frame.f_code.co_filename,
        frame.f_lineno,
        event,
        linecache.getline(frame.f_code.co_filename, frame.f_lineno)
    ))
    return dumbtrace # "step in"

sys.settrace(dumbtrace)
try:
    foobar(1, 2)
finally: # why uninstall?
    sys.settrace(None)
```


A basic tracer

Gets you this:

examples.py:3	call	def foobar(a, b):
examples.py:4	line	c = a + b
examples.py:5	line	try:
examples.py:6	line	raise Exception(c)
examples.py:6	exception	raise Exception(c)
examples.py:7	line	except Exception as exc:
examples.py:8	line	raise RuntimeError(exc)
examples.py:8	exception	raise RuntimeError(exc)
examples.py:10	line	c = None
examples.py:10	return	c = None

For some code:

```
def foobar(a, b):
    c = a + b
    try:
        raise Exception(c)
    except Exception as exc:
        raise RuntimeError(exc)
    finally:
        c = None
```

What other stuff can you do?

Besides printing code, you could implement:

- debuggers (like pdb)
- call graphs
- line profilers
- special purpose logging
- testing tools (coverage)
- crazy stuff

Crazy stuff?

Did you know decorators were added in Python 2.4?

On 2.3 [DecoratorTools](#) supports "decorator syntax" with a tracer:

```
from peak.util.decorators import decorate

class Foo(object):
    decorate(classmethod)
    def something(cls, etc):
        """This is a classmethod"""
```

How?

Traces the local scope, diffs locals and patches when the function gets added into them.

What about profilers

A profiler pretty much the same, only that:

- it isn't called for **line** events
- return value doesn't matter (it's always "stepped in")

Dealing with the slowness

There is support to implement a tracer in C:

```
PyEval_SetTrace(my_fast_tracer, optional_context_object)
```

What `sys.settrace` actually do:

```
PyEval_SetTrace(trace_trampoline, args); // args is the function
```

The trampoline starts like this

```
trace_trampoline(PyObject *self, PyFrameObject *frame,  
                  int what, PyObject *arg) {  
    PyObject *callback;  
    if (what == PyTrace_CALL)  
        callback = self;  
    else  
        callback = frame->f_trace;  
    if (callback == NULL)  
        return 0;  
    ...  
}
```


The frame object

The most interesting stuff:

- `frame.f_locals`
- `frame.f_globals`
- `frame.f_code.co_name` - function name
- `frame.f_lineno`
- `frame.f_globals['__name__']` - module name (could be missing tho)

A better way?

Contrived when adding filters:

```
def dumbtrace(frame, event, args):
    filename = frame.f_code.co_filename # missing handling for .pyc/o etc
    if filename == 'examples.py':
        sys.stdout.write("%015s:%-3s %09s %s" % (
            filename,
            frame.f_lineno,
            event,
            linecache.getline(frame.f_code.co_filename, frame.f_lineno)
        ))
    return dumbtrace # "step in"

sys.settrace(dumbtrace)
try:
    foobar(1, 2)
finally: # why uninstall?
    sys.settrace(None)
```

An every day tracer

<http://python-hunter.rtfd.org/>

```
import hunter

with hunter.trace(filename='examples.py'):
    foobar(1, 2)
```

Would result in (sans colors):

```
examples.py:34 line      foobar(1, 2)
examples.py:3 call      => foobar(a=1, b=2)
examples.py:4 line      c = a + b
examples.py:5 line      try:
examples.py:6 line      raise Exception(c)
examples.py:6 exception ! foobar: (<class 'Exception'>, Exception(3,), ...)
examples.py:7 line      except Exception as exc:
examples.py:8 line      raise RuntimeError(exc)
examples.py:8 exception ! foobar: (<class 'RuntimeError'>, RuntimeError(Exception(3,)), ...)
examples.py:10 line      c = None
examples.py:10 return    <= foobar: None
examples.py:34 exception ! <module>: (<class 'RuntimeError'>, RuntimeError(Exception(3,)), ...)
```


What can you filter on

- **arg** - A value that depends on kind
- **calls** - A counter for total number of calls up to this Event
- **depth** - Tracing depth (increases on calls, decreases on returns)
- **filename** - A string with absolute path to file.
- **fullsource** - A string with a line or more (decorators are included if it's a class/function definition).
- **function** - A string with function name.
- **globals** - A dict with global variables.
- **kind** - The kind of the event ('call', 'line', 'return' or 'exception').

What can you filter on

- `lineno` - An integer with line number in file.
- `locals` - A dict with local variables.
- `module` - A string with module name (eg – "foo.bar").
- `source` - A string with the sourcecode (fast but may be incomplete).
- `stdlib` - A boolean flag. True if frame is in stdlib.
- `threadid` - Current thread ident. If current thread is main thread then it returns None.
- `threadname` - Current thread name.

Filtering operators

```
hunter.trace(filename='examples.py')
```

```
hunter.trace(filename_endswith='examples.py')
```

```
hunter.trace(filename_ew='examples.py')
```

```
hunter.trace(filename_regex=r'[/]examples\.py')
```

```
hunter.trace(filename_rx=r'[/]examples\.py')
```

```
hunter.trace(filename_contains='examples')
```

```
hunter.trace(filename_has='examples')
```

```
hunter.trace(filename_in=['examples.py'])
```

```
hunter.trace(depth_lt=10)
```

```
hunter.trace(depth_lte=10)
```

```
hunter.trace(depth_gt=1)
```

```
hunter.trace(depth_gte=1)
```


Environment activation

Bash:

```
PYTHONHUNTER='module="os.path"' python examples.py
```

Batch:

```
set PYTHONHUNTER=module='os.path'  
python examples.py
```

Strace-like functionality

It has it but it's Linux only (probably works on BSD/OSX):

```
hunter-trace [-h] -p PID [-t TIMEOUT] [--gdb] [-s SIGNAL]
              [OPTIONS [OPTIONS ...]]
```

Arguments:

- gdb Use GDB to activate tracing. WARNING: it may deadlock the process!

- s *SIGNAL*, --signal *SIGNAL*

Send the given *SIGNAL* to the process before connecting.

The `OPTIONS` are `hunter.trace()` arguments.

Strace-like functionality

How it works:

- GDB: automates it and drops this bomb in the process:

```
Py_AddPendingCall(  
    PyRun_SimpleString,  
    "from hunter import remote; remote.activate(...)")
```

- [Manhole](#): requires user to install it:

```
from hunter import remote  
remote.install()
```

An idea for Windows support:

- One way would be using <https://github.com/fabioz/PyDev.Debugger/> for GDB-like attach-inject.

Strace-like functionality

How it works:

- GDB: automates it and drops this bomb in the process:

```
Py_AddPendingCall(  
    PyRun_SimpleString,  
    "from hunter import remote; remote.activate(...)")
```

- [Manhole](#): requires user to install it:

```
from hunter import remote  
remote.install()
```

An idea for Windows support:

- One way would be using <https://github.com/fabioz/PyDev.Debugger/> for GDB-like attach-inject.