Hunter

Release 3.4.0

Mar 25, 2021

Contents

CHAPTER 1

Overview

docs	
tests	
package	

Hunter is a flexible code tracing toolkit, not for measuring coverage, but for debugging, logging, inspection and other nefarious purposes. It has a simple Python API, a *convenient terminal API* and a *CLI tool to attach to processes*.

- Free software: BSD 2-Clause License

1.1 Installation

```
pip install hunter
```

1.2 Documentation

https://python-hunter.readthedocs.io/

1.3 Overview

Basic use involves passing various filters to the `trace` option. An example:

```
import hunter
hunter.trace(module='posixpath', action=hunter.CallPrinter)

import os
os.path.join('a', 'b')
```

That would result in:

```
>>> os.path.join('a', 'b')
         /usr/lib/python3.6/posixpath.py:75    call      => join(a='a')
         /usr/lib/python3.6/posixpath.py:80    line         a = os.fspath(a)
         /usr/lib/python3.6/posixpath.py:81    line         sep = _get_sep(a)
         /usr/lib/python3.6/posixpath.py:41    call         => _get_sep(path='a')
         /usr/lib/python3.6/posixpath.py:42    line            if isinstance(path,␣
↪bytes):
         /usr/lib/python3.6/posixpath.py:45    line               return '/'
         /usr/lib/python3.6/posixpath.py:45    return          <= _get_sep: '/'
         /usr/lib/python3.6/posixpath.py:82    line         path = a
         /usr/lib/python3.6/posixpath.py:83    line         try:
         /usr/lib/python3.6/posixpath.py:84    line         if not p:
         /usr/lib/python3.6/posixpath.py:86    line         for b in map(os.fspath,␣
↪p):
         /usr/lib/python3.6/posixpath.py:87    line            if b.startswith(sep):
         /usr/lib/python3.6/posixpath.py:89    line            elif not path or path.
↪endswith(sep):
         /usr/lib/python3.6/posixpath.py:92    line               path += sep + b
         /usr/lib/python3.6/posixpath.py:86    line            for b in map(os.fspath,␣
↪p):
         /usr/lib/python3.6/posixpath.py:96    line         return path
         /usr/lib/python3.6/posixpath.py:96    return    <= join: 'a/b'
'a/b'
```

In a terminal it would look like:

```
>>> os.path.join('a', 'b')
        /usr/lib/python3.6/posixpath.py:75    call      def join(a, *p):
        /usr/lib/python3.6/posixpath.py:80    line          a = os.fspath(a)
        /usr/lib/python3.6/posixpath.py:81    line          sep = _get_sep(a)
        /usr/lib/python3.6/posixpath.py:41    call      def _get_sep(path):
        /usr/lib/python3.6/posixpath.py:42    line          if isinstance(path, bytes):
        /usr/lib/python3.6/posixpath.py:45    line              return '/'
        /usr/lib/python3.6/posixpath.py:45    return            return '/'
                                              ...       return value: '/'
        /usr/lib/python3.6/posixpath.py:82    line          path = a
        /usr/lib/python3.6/posixpath.py:83    line          try:
        /usr/lib/python3.6/posixpath.py:84    line              if not p:
        /usr/lib/python3.6/posixpath.py:86    line              for b in map(os.fspath, p):
        /usr/lib/python3.6/posixpath.py:87    line                  if b.startswith(sep):
        /usr/lib/python3.6/posixpath.py:89    line                  elif not path or path.endswith(sep):
        /usr/lib/python3.6/posixpath.py:92    line                      path += sep + b
        /usr/lib/python3.6/posixpath.py:86    line              for b in map(os.fspath, p):
        /usr/lib/python3.6/posixpath.py:96    line          return path
        /usr/lib/python3.6/posixpath.py:96    return        return path
                                              ...       return value: 'a/b'
'a/b'
```

1.3.1 Actions

Output format can be controlled with "actions". There's an alternative `CodePrinter` action that doesn't handle nesting (it was the default action until Hunter 2.0).

If filters match then action will be run. Example:

```
import hunter
hunter.trace(module='posixpath', action=hunter.CodePrinter)

import os
os.path.join('a', 'b')
```

That would result in:

```
>>> os.path.join('a', 'b')
         /usr/lib/python3.6/posixpath.py:75    call      def join(a, *p):
         /usr/lib/python3.6/posixpath.py:80    line          a = os.fspath(a)
         /usr/lib/python3.6/posixpath.py:81    line          sep = _get_sep(a)
         /usr/lib/python3.6/posixpath.py:41    call      def _get_sep(path):
         /usr/lib/python3.6/posixpath.py:42    line          if isinstance(path,␣
↪bytes):
         /usr/lib/python3.6/posixpath.py:45    line              return '/'
         /usr/lib/python3.6/posixpath.py:45    return            return '/'
                                               ...       return value: '/'
         /usr/lib/python3.6/posixpath.py:82    line          path = a
         /usr/lib/python3.6/posixpath.py:83    line          try:
         /usr/lib/python3.6/posixpath.py:84    line              if not p:
         /usr/lib/python3.6/posixpath.py:86    line              for b in map(os.
↪fspath, p):
         /usr/lib/python3.6/posixpath.py:87    line                  if b.
↪startswith(sep):
         /usr/lib/python3.6/posixpath.py:89    line                  elif not path or␣
↪path.endswith(sep):
         /usr/lib/python3.6/posixpath.py:92    line                      path += sep␣
↪+ b
         /usr/lib/python3.6/posixpath.py:86    line              for b in map(os.
↪fspath, p):
         /usr/lib/python3.6/posixpath.py:96    line          return path
         /usr/lib/python3.6/posixpath.py:96    return        return path
                                               ...       return value: 'a/b'
'a/b'
```

- or in a terminal:

```
>>> os.path.join('a', 'b')
         /usr/lib/python3.6/posixpath.py:75    call      => join(a='a')
         /usr/lib/python3.6/posixpath.py:80    line         a = os.fspath(a)
         /usr/lib/python3.6/posixpath.py:81    line         sep = _get_sep(a)
         /usr/lib/python3.6/posixpath.py:41    call         => _get_sep(path='a')
         /usr/lib/python3.6/posixpath.py:42    line            if isinstance(path, bytes):
         /usr/lib/python3.6/posixpath.py:45    line            return '/'
         /usr/lib/python3.6/posixpath.py:45    return       <= _get_sep: '/'
         /usr/lib/python3.6/posixpath.py:82    line         path = a
         /usr/lib/python3.6/posixpath.py:83    line         try:
         /usr/lib/python3.6/posixpath.py:84    line         if not p:
         /usr/lib/python3.6/posixpath.py:86    line         for b in map(os.fspath, p):
         /usr/lib/python3.6/posixpath.py:87    line         if b.startswith(sep):
         /usr/lib/python3.6/posixpath.py:89    line         elif not path or path.endswith(sep):
         /usr/lib/python3.6/posixpath.py:92    line         path += sep + b
         /usr/lib/python3.6/posixpath.py:86    line         for b in map(os.fspath, p):
         /usr/lib/python3.6/posixpath.py:96    line         return path
         /usr/lib/python3.6/posixpath.py:96    return    <= join: 'a/b'
'a/b'
```

Another useful action is the `VarsPrinter`:

```
import hunter
# note that this kind of invocation will also use the default `CallPrinter` action
hunter.trace(hunter.Q(module='posixpath', action=hunter.VarsPrinter('path')))

import os
os.path.join('a', 'b')
```

That would result in:

```
>>> os.path.join('a', 'b')
     /usr/lib/python3.6/posixpath.py:75    call      => join(a='a')
     /usr/lib/python3.6/posixpath.py:80    line         a = os.fspath(a)
     /usr/lib/python3.6/posixpath.py:81    line         sep = _get_sep(a)
     /usr/lib/python3.6/posixpath.py:41    call      [path => 'a']
     /usr/lib/python3.6/posixpath.py:41    call         => _get_sep(path='a')
     /usr/lib/python3.6/posixpath.py:42    line      [path => 'a']
     /usr/lib/python3.6/posixpath.py:42    line            if isinstance(path, bytes):
     /usr/lib/python3.6/posixpath.py:45    line      [path => 'a']
     /usr/lib/python3.6/posixpath.py:45    line            return '/'
     /usr/lib/python3.6/posixpath.py:45    return    [path => 'a']
     /usr/lib/python3.6/posixpath.py:45    return       <= _get_sep: '/'
     /usr/lib/python3.6/posixpath.py:82    line         path = a
     /usr/lib/python3.6/posixpath.py:83    line      [path => 'a']
     /usr/lib/python3.6/posixpath.py:83    line         try:
     /usr/lib/python3.6/posixpath.py:84    line      [path => 'a']
     /usr/lib/python3.6/posixpath.py:84    line         if not p:
     /usr/lib/python3.6/posixpath.py:86    line      [path => 'a']
     /usr/lib/python3.6/posixpath.py:86    line         for b in map(os.fspath, p):
     /usr/lib/python3.6/posixpath.py:87    line      [path => 'a']
     /usr/lib/python3.6/posixpath.py:87    line         if b.startswith(sep):
     /usr/lib/python3.6/posixpath.py:89    line      [path => 'a']
     /usr/lib/python3.6/posixpath.py:89    line         elif not path or path.
↪endswith(sep):
     /usr/lib/python3.6/posixpath.py:92    line      [path => 'a']
     /usr/lib/python3.6/posixpath.py:92    line         path += sep + b
     /usr/lib/python3.6/posixpath.py:86    line      [path => 'a/b']
     /usr/lib/python3.6/posixpath.py:86    line         for b in map(os.fspath, p):
     /usr/lib/python3.6/posixpath.py:96    line      [path => 'a/b']
     /usr/lib/python3.6/posixpath.py:96    line         return path
     /usr/lib/python3.6/posixpath.py:96    return    [path => 'a/b']
     /usr/lib/python3.6/posixpath.py:96    return    <= join: 'a/b'
'a/b'
```

In a terminal it would look like:

```
>>> os.path.join('a', 'b')
        /usr/lib/python3.6/posixpath.py:75    call         => join(a='a')
        /usr/lib/python3.6/posixpath.py:80    line            a = os.fspath(a)
        /usr/lib/python3.6/posixpath.py:81    line            sep = _get_sep(a)
        /usr/lib/python3.6/posixpath.py:41    call            => _get_sep(path='a')
        /usr/lib/python3.6/posixpath.py:41    call         [path => 'a']
        /usr/lib/python3.6/posixpath.py:42    line               if isinstance(path, bytes):
        /usr/lib/python3.6/posixpath.py:42    line         [path => 'a']
        /usr/lib/python3.6/posixpath.py:45    line               return '/'
        /usr/lib/python3.6/posixpath.py:45    line         [path => 'a']
        /usr/lib/python3.6/posixpath.py:45    return          <= _get_sep: '/'
        /usr/lib/python3.6/posixpath.py:45    return       [path => 'a']
        /usr/lib/python3.6/posixpath.py:82    line            path = a
        /usr/lib/python3.6/posixpath.py:83    line            try:
        /usr/lib/python3.6/posixpath.py:83    line         [path => 'a']
        /usr/lib/python3.6/posixpath.py:84    line               if not p:
        /usr/lib/python3.6/posixpath.py:84    line         [path => 'a']
        /usr/lib/python3.6/posixpath.py:86    line               for b in map(os.fspath, p):
        /usr/lib/python3.6/posixpath.py:86    line         [path => 'a']
        /usr/lib/python3.6/posixpath.py:87    line               if b.startswith(sep):
        /usr/lib/python3.6/posixpath.py:87    line         [path => 'a']
        /usr/lib/python3.6/posixpath.py:89    line               elif not path or path.endswith(sep):
        /usr/lib/python3.6/posixpath.py:89    line         [path => 'a']
        /usr/lib/python3.6/posixpath.py:92    line               path += sep + b
        /usr/lib/python3.6/posixpath.py:92    line         [path => 'a']
        /usr/lib/python3.6/posixpath.py:86    line               for b in map(os.fspath, p):
        /usr/lib/python3.6/posixpath.py:86    line         [path => 'a/b']
        /usr/lib/python3.6/posixpath.py:96    line               return path
        /usr/lib/python3.6/posixpath.py:96    line         [path => 'a/b']
        /usr/lib/python3.6/posixpath.py:96    return          <= join: 'a/b'
        /usr/lib/python3.6/posixpath.py:96    return       [path => 'a/b']
'a/b'
```

You can give it a tree-like configuration where you can optionally configure specific actions for parts of the tree (like dumping variables or a pdb set_trace):

```
from hunter import trace, Q, Debugger
from pdb import Pdb

trace(
    # drop into a Pdb session if ``foo.bar()`` is called
    Q(module="foo", function="bar", kind="call", action=Debugger(klass=Pdb))
    |  # or
    Q(
        # show code that contains "mumbo.jumbo" on the current line
        lambda event: event.locals.get("mumbo") == "jumbo",
        # and it's not in Python's stdlib
        stdlib=False,
        # and it contains "mumbo" on the current line
        source__contains="mumbo"
    )
)

import foo
foo.func()
```

With a `foo.py` like this:

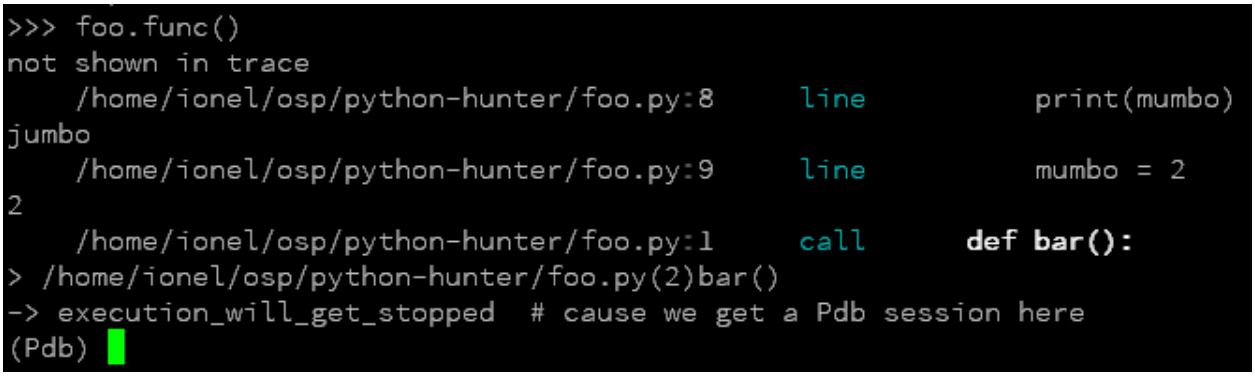
```
def bar():
    execution_will_get_stopped  # cause we get a Pdb session here

def func():
    mumbo = 1
    mumbo = "jumbo"
    print("not shown in trace")
    print(mumbo)
    mumbo = 2
    print(mumbo) # not shown in trace
    bar()
```

We get:

```
>>> foo.func()
not shown in trace
    /home/ionel/osp/python-hunter/foo.py:8     line          print(mumbo)
jumbo
    /home/ionel/osp/python-hunter/foo.py:9     line          mumbo = 2
2
    /home/ionel/osp/python-hunter/foo.py:1     call      def bar():
> /home/ionel/osp/python-hunter/foo.py(2)bar()
-> execution_will_get_stopped  # cause we get a Pdb session here
(Pdb)
```

In a terminal it would look like:



1.3.2 Tracing processes

In similar fashion to `strace` Hunter can trace other processes, eg:

```
hunter-trace --gdb -p 123
```

If you wanna play it safe (no messy GDB) then add this in your code:

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

Then you can do:

```
hunter-trace -p 123
```

See docs on the remote feature.

Note: Windows ain't supported.

1.3.3 Environment variable activation

For your convenience environment variable activation is available. Just run your app like this:

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

On Windows you'd do something like:

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

The activation works with a clever `.pth` file that checks for that env var presence and before your app runs does something like this:

```
from hunter import *
trace(<whatever-you-had-in-the-PYTHONHUNTER-env-var>)
```

Note that Hunter is activated even if the env var is empty, eg: `PYTHONHUNTER=""`.

Environment variable configuration

Sometimes you always use the same options (like `stdlib=False` or `force_colors=True`). To save typing you can set something like this in your environment:

```
PYTHONHUNTERCONFIG="stdlib=False,force_colors=True"
```

This is the same as `PYTHONHUNTER="stdlib=False,action=CallPrinter(force_colors=True)"`.

Notes:

- Setting `PYTHONHUNTERCONFIG` alone doesn't activate hunter.
- All the options for the builtin actions are supported.
- Although using predicates is supported it can be problematic. Example of setup that won't trace anything:

  ```
  PYTHONHUNTERCONFIG="Q(module_startswith='django')"
  PYTHONHUNTER="Q(module_startswith='celery')"
  ```

  which is the equivalent of:

  ```
  PYTHONHUNTER="Q(module_startswith='django'),Q(module_startswith='celery')"
  ```

  which is the equivalent of:

  ```
  PYTHONHUNTER="Q(module_startswith='django')&Q(module_startswith='celery')"
  ```

1.3.4 Filtering DSL

Hunter supports a flexible query DSL, see the introduction.

1.4 Development

To run the all tests run:

```
tox
```

1.5 Design notes

Hunter doesn't do everything. As a design goal of this library some things are made intentionally austere and verbose (to avoid complexity, confusion and inconsistency). This has few consequences:

- There are Operators but there's no negation operator. Instead you're expected to negate a Query object, eg: `~Q(module='re')`.
- There are no specialized operators or filters - all filters behave exactly the same. For example:
  - No filter for packages. You're expected to filter by module with an operator.
  - No filter for arguments, return values or variables. You're expected to write your own filter function and deal with the problems of poking into objects.
- Layering is minimal. There's are some helpers that do some argument processing and conversions to save you some typing but that's about it.
- The library doesn't try to hide the mechanics of tracing in Python - it's 1:1 regarding what Python sends to a trace function if you'd be using sys.settrace.
- Doesn't have any storage. You are expected to redirect output to a file.

You should look at it like it's a tool to help you understand and debug big applications, or a framework ridding you of the boring parts of settrace, not something that helps you learn Python.

1.6 FAQ

1.6.1 Why not Smiley?

There's some obvious overlap with smiley but there are few fundamental differences:

- Complexity. Smiley is simply over-engineered:
  - It uses IPC and a SQL database.
  - It has a webserver. Lots of dependencies.
  - It uses threads. Side-effects and subtle bugs are introduced in your code.
  - It records everything. Tries to dump any variable. Often fails and stops working.

  Why do you need all that just to debug some stuff in a terminal? Simply put, it's a nice idea but the design choices work against you when you're already neck-deep into debugging your own code. In my experience Smiley has been very buggy and unreliable. Your mileage may vary of course.
- Tracing long running code. This will make Smiley record lots of data, making it unusable.

  Now because Smiley records everything, you'd think it's better suited for short programs. But alas, if your program runs quickly then it's pointless to record the execution. You can just run it again.

It seems there's only one situation where it's reasonable to use Smiley: tracing io-bound apps remotely. Those apps don't execute lots of code, they just wait on network so Smiley's storage won't blow out of proportion and tracing overhead might be acceptable.

- Use-cases. It seems to me Smiley's purpose is not really debugging code, but more of a "non interactive monitoring" tool.

In contrast, Hunter is very simple:

- Few dependencies.
- Low overhead (tracing/filtering code has an optional Cython extension).
- No storage. This simplifies lots of things.

  The only cost is that you might need to run the code multiple times to get the filtering/actions right. This means Hunter is not really suited for "post-mortem" debugging. If you can't reproduce the problem anymore then Hunter won't be of much help.

1.6.2 Why not pytrace?

Pytrace is another tracer tool. It seems quite similar to Smiley - it uses a sqlite database for the events, threads and IPC, thus it's reasonable to expect the same kind of problems.

1.6.3 Why not PySnooper or snoop?

snoop is a refined version of PySnooper. Both are more suited to tracing small programs or functions as the output is more verbose and less suited to the needs of tracing a big application where Hunter provides more flexible setup, filtering capabilities, speed and brevity.

1.6.4 Why not coverage?

For purposes of debugging coverage is a great tool but only as far as "debugging by looking at what code is (not) run". Checking branch coverage is good but it will only get you as far.

From the other perspective, you'd be wondering if you could use Hunter to measure coverage-like things. You could do it but for that purpose Hunter is very "rough": it has no builtin storage. You'd have to implement your own storage. You can do it but it wouldn't give you any advantage over making your own tracer if you don't need to "pre-filter" whatever you're recording.

In other words, filtering events is the main selling point of Hunter - it's fast (cython implementation) and the query API is flexible enough.

1.7 Projects using Hunter

Noteworthy usages or Hunter (submit a PR with your project if you built a tool that relies on hunter):

- Crunch-io/diagnose - a runtime instrumentation library.
- talwrii/huntrace - an alternative cli (similar to ltrace).
- anki-code/xunter - a profiling tool made specifically for the xonsh shell.

More projects using it at https://github.com/ionelmc/python-hunter/network/dependents

CHAPTER 2

Installation

At the command line:

```
pip install hunter
```

CHAPTER 3

Introduction

3.1 Installation

To install hunter run:

```
pip install hunter
```

3.2 The `trace` function

The *hunter.trace* function can take 2 types of arguments:

- Keyword arguments like `module`, `function` or `action` (see `hunter.Event` for all the possible filters).
- Callbacks that take an `event` argument:
  - Builtin predicates like: *hunter.predicates.Query*, `hunter.When`, *hunter.And* or *hunter.Or*.
  - Actions like: *hunter.actions.CodePrinter*, *hunter.actions.Debugger* or *hunter.actions.VarsPrinter*
  - Any function. Or a disgusting lambda.

Note that *hunter.trace* will use *hunter.Q* when you pass multiple positional arguments or keyword arguments.

3.3 The `Q` function

The *hunter.Q()* function provides a convenience API for you:

- `Q(module='foobar')` is converted to `Query(module='foobar')`.
- `Q(module='foobar', action=Debugger)` is converted to `When(Query(module='foobar'), Debugger)`.

- `Q(module='foobar', actions=[CodePrinter, VarsPrinter('name')])` is converted to `When(Query(module='foobar'), CodePrinter, VarsPrinter('name'))`.
- `Q(Q(module='foo'), Q(module='bar'))` is converted to `And(Q(module='foo'), Q(module='bar'))`.
- `Q(your_own_callback, module='foo')` is converted to `And(your_own_callback, Q(module='foo'))`.

Note that the default junction *hunter.Q()* uses is *hunter.predicates.And*.

3.4 Composing

All the builtin predicates (*hunter.predicates.Query*, *hunter.predicates.When*, *hunter.predicates.And*, *hunter.predicates.Not* and *hunter.predicates.Or*) support the `|`, `&` and `~` operators:

- `Query(module='foo') | Query(module='bar')` is converted to `Or(Query(module='foo'), Query(module='bar'))`
- `Query(module='foo') & Query(module='bar')` is converted to `And(Query(module='foo'), Query(module='bar'))`
- `~Query(module='foo')` is converted to `Not(Query(module='foo'))`

3.5 Operators

New in version 1.0.0: You can add `startswith`, `endswith`, `in`, `contains`, `regex`, `lt`, `lte`, `gt`, `gte` to your keyword arguments, just like in Django. Double underscores are not necessary, but in case you got twitchy fingers it'll just work - `filename__startswith` is the same as `filename_startswith`.

New in version 2.0.0: You can also use these convenience aliases: `sw` (`startswith`), `ew` (`endswith`), `rx` (`regex`) and `has` (`contains`).

Examples:

- `Query(module_in=['re', 'sre', 'sre_parse'])` will match events from any of those modules.
- `~Query(module_in=['re', 'sre', 'sre_parse'])` will match events from any modules except those.
- `Query(module_startswith=['re', 'sre', 'sre_parse'])` will match any events from modules that starts with either of those. That means `repr` will match!
- `Query(module_regex='(re|sre.*)$')` will match any events from `re` or anything that starts with `sre`.

Note: If you want to filter out stdlib stuff you're better off with using `Query(stdlib=False)`.

3.6 Activation

You can activate Hunter in three ways.

3.6.1 from code

```
import hunter
hunter.trace(
    ...
)
```

3.6.2 with an environment variable

Set the `PYTHONHUNTER` environment variable. Eg:

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

On Windows you'd do something like:

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

The activation works with a clever `.pth` file that checks for that env var presence and before your app runs does something like this:

```
from hunter import *
trace(
    <whatever-you-had-in-the-PYTHONHUNTER-env-var>
)
```

That also means that it will do activation even if the env var is empty, eg: `PYTHONHUNTER=""`.

3.6.3 with a CLI tool

If you got an already running process you can attach to it with `hunter-trace`. See *Remote tracing* for details.

CHAPTER 4

Remote tracing

Hunter supports tracing local processes, with two backends: manhole and GDB. For now Windows isn't supported.

Using GDB is risky (if anything goes wrong your process will probably be hosed up badly) so the Manhole backend is recommended. To use it:

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

You should put this somewhere where it's run early in your project (settings or package's `__init__.py` file).

The `remote.install()` takes same arguments as `manhole.install()`. You'll probably only want to use `verbose=False`...

4.1 The CLI

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

positional arguments: OPTIONS

optional arguments:

-h, --help show this help message and exit

-p PID, --pid PID A numerical process id.

-t TIMEOUT, --timeout TIMEOUT Timeout to use. Default: 1 seconds.

--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.

CHAPTER 5

Configuration

Default predicates and action kwargs defaults can be configured via a `PYTHONHUNTERCONFIG` environment variable.

All the action kwargs:

- `klass`
- `stream`
- `force_colors`
- `force_pid`
- `filename_alignment`
- `thread_alignment`
- `pid_alignment`
- `repr_limit`
- `repr_func`

Example:

```
PYTHONHUNTERCONFIG="stdlib=False,force_colors=True"
```

This is the same as `PYTHONHUNTER="stdlib=False,action=CallPrinter(force_colors=True)"`.

Notes:

- Setting `PYTHONHUNTERCONFIG` alone doesn't activate hunter.
- All the options for the builtin actions are supported.
- Although using predicates is supported it can be problematic. Example of setup that won't trace anything:

  ```
  PYTHONHUNTERCONFIG="Q(module_startswith='django')"
  PYTHONHUNTER="Q(module_startswith='celery')"
  ```

  which is the equivalent of:

```
PYTHONHUNTER="Q(module_startswith='django'),Q(module_startswith='celery')"
```

which is the equivalent of:

```
PYTHONHUNTER="Q(module_startswith='django')&Q(module_startswith='celery')"
```

CHAPTER 6

Filtering

A list of all the keyword filters that *hunter.trace* or *hunter.Q* accept:

- `arg` - you probably don't care about this - it may have a value for return/exception events
- `builtin` (*bool*) - `True` if function is a builtin function
- `calls` (*int*) - a call counter, you can use it to limit output by using a `lt` operator
- `depth` (*int*) - call depth, starts from 0, increases for call events and decreases for returns
- `filename` (*str*)
- `fullsource` (*str*) - sourcecode for the executed lines (may be multiple lines in some situations)
- `function` (*str*) - function name
- `globals` (*dict*) - global variables
- `instruction` (*int* or *str*, depending on Python version) - current executed bytecode, see *Silenced exception runtime analysis* for example usage
- `kind` (*str*) - one of 'call', 'exception', 'line' or 'return'
- `lineno` (*int*)
- `locals` (*dict*) - local variables
- `module` (*str*) - dotted module
- `source` (*str*) - sourcecode for the executed line
- `stdlib` (*bool*) - `True` if module is from stdlib
- `threadid` (*int*)
- `threadname` (*str*) - whatever threading.Thread.name returns

You can append operators to the above filters. Note that some of of the filters won't work well with the *bool* or *int* types.

- `contains` - works best with *str*, for example `module_contains='foobar'` translates to `'foobar' in event.module`

- `has` - alias for `contains`
- `endswith` - works best with *str*, for example `module_endswith='foobar'` translates to `event.module.endswith('foobar')`. You can also pass in a iterable, example `module_endswith=('foo', 'bar')` is acceptable
- `ew` - alias for `endswith`
- `gt` - works best with *int*, for example `lineno_gt=100` translates to `event.lineno > 100`
- `gte` - works best with *int*, for example `lineno_gte=100` translates to `event.lineno >= 100`
- `in` - a membership test, for example `module_in=('foo', 'bar')` translates to `event.module in ('foo', 'bar')`. You can use any iterable, for example `module_in='foo bar'` translates to `event.module in 'foo bar'`, and that would probably have the same result as the first example
- `lt` - works best with *int*, for example `calls_lt=100` translates to `event.calls < 100`
- `lte` - works best with *int*, for example `depth_lte=100` translates to `event.depth <= 100`
- `regex` - works best with *str*, for example `module_regex=r'(test|test.*)\b'` translates to `re.match(r'(test|test.*)\b', event.module)`
- `rx` - alias for `regex`
- `startswith` - works best with *str*, for example `module_startswith='foobar'` translates to `event.module.startswith('foobar')`. You can also pass in a iterable, example `module_startswith=('foo', 'bar')` is acceptable
- `sw` - alias for `startswith`

Notes:

- you can also use double underscore (if you're too used to Django query lookups), eg: `module__has='foobar'` is acceptable
- there's nothing smart going on for the dots in module names so sometimes you might need to account for said dots:
  - `module_sw='foo'` will match `"foo.bar"` and `"foobar"` - if you want to avoid matchin the later you could do either of:
    * `Q(module='foo')|Q(module_sw='foo.')`
    * `Q(module_rx=r'foo($|\.)')` - but this might cost you in speed
    * `Q(filename_sw='/path/to/foo/')` - probably the fastest
    * `Q(filename_has='/foo/')` - avoids putting in the full path but might match unwanted paths

CHAPTER 7

Cookbook

When in doubt, use Hunter.

7.1 Walkthrough

Sometimes you just want to get an overview of an unfamiliar application code, eg: only see calls/returns/exceptions.

In this situation, you could use something like `~Q(kind="line"),~Q(module_in=["six", "pkg_resources"]),~Q(filename=""),stdlib=False`. Lets break that down:

- `~Q(kind="line")` means skip line events (~ is a negation of the filter).
- `stdlib=False` means we don't want to see anything from stdlib.
- `~Q(module_in=["six","pkg_resources")]` means we're tired of seeing stuff from those modules in site-packages.
- `~Q(filename="")` is necessary for filtering out events that come from code without a source (like the interpreter bootstrap stuff).

You would run the application (in Bash) like:

```
PYTHONHUNTER='~Q(kind="line"),~Q(module_in=["six","pkg_resources"]),~Q(filename=""),
↪stdlib=False' myapp (or python myapp.py)
```

Additionally you can also add a depth filter (eg: `depth_lt=10`) to avoid too deep output.

7.2 Packaging

I frequently use Hunter to figure out how distutils/setuptools work. It's very hard to figure out what's going on by just looking at the code - lots of stuff happens at runtime. If you ever tried to write a custom command you know what I mean.

To show everything that is being run:

```
PYTHONHUNTER='module_startswith=["setuptools", "distutils", "wheel"]' python setup.py␣
↪bdist_wheel
```

If you want too see some interesting variables:

```
PYTHONHUNTER='module_startswith=["setuptools", "distutils", "wheel"],␣
↪actions=[CodePrinter, VarsPrinter("self.bdist_dir")]' python setup.py bdist_wheel
```

7.3 Typical

Normally you'd only want to look at your code. For that purpose, there's the `stdlib` option. Set it to `False`.

Building a bit on the previous example, if I have a `build` Distutils command and I only want to see my code then I'd run this:

```
PYTHONHUNTER='stdlib=False' python setup.py build
```

But this also means I'd be seeing anything from `site-packages`. I could filter on only the events from the current directory (assuming the filename is going to be a relative path):

```
PYTHONHUNTER='~Q(filename_startswith="/")' python setup.py build
```

7.4 Debugging a test

In tests it is convenient to ignore everything that is in `stdlib` and `site-packages` and start hunter right before the tested function.

```
from hunter import trace, Q
trace(Q(stdlib=False), ~Q(filename_contains='site-packages'))
```

It also helps to save output into a file to compare different runs. An example below uses `pytest` with `-k` option to select and tun a test or tests with string `some` in name. The output is then piped to `testout1` file.

```
pytest test/test_simple.py -k some &> testout1
```

7.5 Needle in the haystack

If the needle might be though the stdlib then you got not choice. But some of the *hay* is very verbose and useless, like stuff from the `re` module.

Note that there are few "hidden" modules like `sre`, `sre_parse`, `sre_compile` etc. You can filter that out with:

```
~Q(module_regex="(re|sre.*)$")
```

Although filtering out that regex stuff can cut down lots of useless output you usually still get lots of output.

Another way, if you got at least some vague idea of what might be going on is to "grep" for sourcecode. Example, to show all the code that does something with a `build_dir` property:

```
source_contains=".build_dir"
```

You could even extend that a bit to dump some variables:

```
source_contains=".build_dir", actions=[CodePrinter, VarsPrinter("self.build_dir")]
```

7.6 Stop after N calls

Say you want to stop tracing after 1000 events, you'd do this:

```
~Q(calls_gt=1000, action=Stop)
```

Explanation:

`Q(calls_gt=1000, action=Stop)` will translate to `When(Query(calls_gt=1000), Stop)`

`Q(calls_gt=1000)` will return `True` when 1000 call count is hit.

`When(something, Stop)` will call `Stop` when `something` returns `True`. However it will also return the result of `something` - the net effect being nothing being shown up to 1000 calls. Clearly not what we want ...

So then we invert the result, `~When(...)` is the same as `Not(When)`.

This may not seem intuitive but for now it makes internals simpler. If `When` would always return `True` then `Or(When, When)` would never run the second `When` and we'd need to have all sorts of checks for this. This may change in the future however.

7.7 "Probe" - lightweight tracing

Based on Robert Brewer's FunctionProbe example.

The use-case is that you'd like to trace a huge application and running a tracer (even a cython one) would have a too great impact. To solve this you'd start the tracer only in placer where it's actually needed.

To make this work you'd monkeypatch the function that needs the tracing. This example uses aspectlib:

```
def probe(qualname, *actions, **filters):
    def tracing_decorator(func):
        @functools.wraps(func)
        def tracing_wrapper(*args, **kwargs):
            # create the Tracer manually to avoid spending time in likely useless␣
↪things like:
            # - loading PYTHONHUNTERCONFIG
            # - setting up the clear_env_var or thread_support options
            # - atexit cleanup registration
            with hunter.Tracer().trace(hunter.When(hunter.Query(**filters),␣
↪*actions)):
                return func(*args, **kwargs)

        return tracing_wrapper

    aspectlib.weave(qualname, tracing_decorator)  # this does the monkeypatch
```

Suggested use:

- to get the regular tracing for that function:

```
probe('module.func', hunter.VarsPrinter('var1', 'var2'))
```

- to log some variables at the end of the target function, and nothing deeper:

```
probe('module.func', hunter.VarsPrinter('var1', 'var2'), kind="return", depth=0)
```

Another interesting thing is that you may note that you can reduce the implementation of the `probe` function down to just:

```
def probe(qualname, *actions, **kwargs):
    aspectlib.weave(qualname, functools.partial(hunter.wrap, actions=actions,␣
↪**kwargs))
```

It will work the same, *`hunter.wrap`* being a decorator. However, while *`hunter.wrap`* offers the convenience of tracing just inside the target function (eg: `probe('module.func', local=True)`) it will also add a lot of extra filtering to trim irrelevant events from around the function (like return from tracer setup, and the internals of the decorator), in addition to what *`hunter.trace()`* does. Not exactly lightweight...

7.8 Silenced exception runtime analysis

Finding code that discards exceptions is sometimes really hard.

Note: This was made available in *`hunter.actions.ErrorSnooper`* for convenience. This cookbook entry will remain for educational purposes.

While this is easy to find with a `grep "except:" -R .`:

```
def silenced_easy():
    try:
        error()
    except:
        pass
```

Variants of this ain't easy to grep:

```
def silenced_easy():
    try:
        error()
    except Exception:
        pass
```

If you can't simply review all the sourcecode then runtime analysis is one way to tackle this:

```
class DumpExceptions(hunter.CodePrinter):
    events = ()
    depth = 0
    count = 0
    exc = None

    def __init__(self, max_count=10, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        self.max_count = max_count
        self.backlog = collections.deque(maxlen=5)
        super(DumpExceptions, self).__init__(**kwargs)

    def __call__(self, event):
        self.count += 1
        if event.kind == 'exception':  # something interesting happened ;)
            self.events = list(self.backlog)
            self.events.append(event.detach(self.try_repr))
            self.exc = self.try_repr(event.arg[1])
            self.depth = event.depth
            self.count = 0
        elif self.events:
            if event.depth > self.depth:  # too many details
                return
            elif event.depth < self.depth and event.kind == 'return':  # stop if␣
↪function returned
                op = event.instruction
                op = op if isinstance(op, int) else ord(op)
                if op == RETURN_VALUE:
                    self.output("{BRIGHT}{fore(BLUE)}{} tracing {} on {}{RESET}\n",
                                ">" * 46, event.function, self.exc)
                    for event in self.events:
                        super(DumpExceptions, self).__call__(event)
                    if self.count > 10:
                        self.output("{BRIGHT}{fore(BLACK)}{} too many lines{RESET}\n",
                                    "-" * 46)
                    else:
                        self.output("{BRIGHT}{fore(BLACK)}{} function exit{RESET}\n",
                                    "-" * 46)
                self.events = []
                self.exc = None
            elif self.count < self.max_count:
                self.events.append(event.detach(self.try_repr))
        else:
            self.backlog.append(event.detach(self.try_repr))
```

Take note about the use of *detach()* and *output()*.

7.9 Profiling

Hunter can be used to implement profiling (measure function timings).

The most basic implementation that only measures timings looks like this:

```
from hunter.actions import Action
from hunter.actions import RETURN_VALUE

class ProfileAction(Action):
    def __init__(self):
        self.timings = {}

    def __call__(self, event):
        if event.kind == 'call':
            self.timings[id(event.frame)] = time()
```

(continues on next page)

(continued from previous page)

```
        elif event.kind == 'return':
            start_time = self.timings.pop(id(event.frame), None)
            if start_time is None:
                return
            delta = time() - start_time
            print(f'{event.function} returned: {event.arg}. Duration: {delta:.4f}s\n')
```

If you don't care about exceptions at all this will be fine, but then you might just as well use a real profiler.

When exceptions occur Python send this to the tracer:

- `event.kind="exception", event.arg=(exc_value, exc_type, tb)`
- `event.kind="return", event.arg=None`

Unfortunately Python emits the return event even if the exception wasn't discarded so we need to do some extra checks around the last bytecode instruction that run at the return event.

This means that we have to store the exception for a little while, and do the check at return:

```
from hunter.actions import Action
from hunter.actions import RETURN_VALUE

class ProfileAction(Action):
    def __init__(self):
        self.timings = {}

    def __call__(self, event):
        current_time = time()
        frame_id = id(event.frame)

        if event.kind == 'call':
            self.timings[frame_id] = current_time, None
        elif frame_id in self.timings:
            start_time, depth, exception = self.timings.pop(frame_id)

            if event.kind == 'exception':
                # store the exception
                # (there will be a followup 'return' event in which we deal with it)
                self.timings[frame_id] = start_time, event.arg
            elif event.kind == 'return':
                delta = current_time - start_time
                if event.instruction == RETURN_VALUE:
                    # exception was discarded
                    print(f'{event.function} returned: {event.arg}. Duration: {delta:.
↪4f}s\n')
                else:
                    print(f'{event.function} raised exception: {exception}. Duration:
↪{delta:.4f}s\n')
```

If you try that example you may notice that it's not completely equivalent to any of the profilers available out there: data for builtin functions is missing.

Python does in fact have a profiling mode (eg: `hunter.trace(profile=True)` and that will make hunter use `sys.setprofile` instead of `sys.setrace`. However there are some downsides with that API:

- exception data will be missing (most likely because profiling is designed for speed and tracebacks are costly to build)

- trace events for builtin functions do not have their own frame objects (so we need to cater for that)

Behold, a *ProfileAction* that works in any mode:

```
from hunter.actions import ColorStreamAction
from hunter.actions import RETURN_VALUE

class ProfileAction(ColorStreamAction):
    # using ColorStreamAction brings this more in line with the other actions
    # (stream option, coloring and such, see the other examples for colors)
    def __init__(self, **kwargs):
        self.timings = {}
        super(ProfileAction, self).__init__(**kwargs)

    def __call__(self, event):
        current_time = time()
        # include event.builtin in the id so we don't have problems
        # with Python reusing frame objects from the previous call for builtin calls
        frame_id = id(event.frame), str(event.builtin)

        if event.kind == 'call':
            self.timings[frame_id] = current_time, None
        elif frame_id in self.timings:
            start_time, exception = self.timings.pop(frame_id)

            # try to find a complete function name for display
            function_object = event.function_object
            if event.builtin:
                function = '<builtin>.{}'.format(event.arg.__name__)
            elif function_object:
                if hasattr(function_object, '__qualname__'):
                    function = '{}.{}'.format(
                        function_object.__module__, function_object.__qualname__
                    )
                else:
                    function = '{}.{}'.format(
                        function_object.__module__,
                        function_object.__name__
                    )
            else:
                function = event.function

            if event.kind == 'exception':
                # store the exception
                # (there will be a followup 'return' event in which we deal with it)
                self.timings[frame_id] = start_time, event.arg
            elif event.kind == 'return':
                delta = current_time - start_time
                if event.instruction == RETURN_VALUE:
                    # exception was discarded
                    self.output(
                        '{fore(BLUE)}{} returned: {}. Duration: {:.4f}s{RESET}\n',
                        function, event.arg, delta
                    )
                else:
                    self.output(
                        '{fore(RED)}{} raised exception: {}. Duration: {:.4f}s{RESET}
↪\n',
```

(continues on next page)

(continued from previous page)

```
                function, exception, delta
            )
```

CHAPTER 8

Reference

Helpers

hunter.trace(*predicates, **options)	Starts tracing.
hunter.stop()	Stop tracing.
hunter.wrap([function_to_trace])	Functions decorated with this will be traced.
hunter.And(*predicates, **kwargs)	Helper that flattens out `predicates` in a single *hunter.predicates.And* object if possible.
hunter.Backlog(*conditions, **kwargs)	Helper that merges kwargs and conditions prior to creating the *Backlog*.
hunter.From([condition, predicate, watermark])	Helper that converts keyword arguments to `From(condition=Q(**normal_kwargs), predicate=Q(**rel_kwargs)` where `rel_kwargs` are all the kwargs that start with "depth" or "calls".
hunter.Not(*predicates, **kwargs)	Helper that flattens out `predicates` in a single *hunter.predicates.And* object if possible.
hunter.Or(*predicates, **kwargs)	Helper that flattens out `predicates` in a single *hunter.predicates.Or* object if possible.
hunter.Q(*predicates, **query)	Helper that handles situations where *hunter.predicates.Query* objects (or other callables) are passed in as positional arguments - it conveniently converts those to a *hunter.predicates.And* predicate.

Actions

hunter.actions.CallPrinter(*args, **kwargs)	An action that just prints the code being executed, but unlike `hunter.CodePrinter` it indents based on callstack depth and it also shows `repr()` of function arguments.
hunter.actions.CodePrinter([stream, ...])	An action that just prints the code being executed.

Continued on next page

Table 2 – continued from previous page

hunter.actions.ColorStreamAction([stream, ...])	Baseclass for your custom action.
hunter.actions.Debugger([klass])	An action that starts pdb.
hunter.actions.ErrorSnooper(*args, **kwargs)	An action that prints events around silenced exceptions.
hunter.actions.Manhole(**options)	
hunter.actions.StackPrinter([depth, limit])	An action that prints a one-line stacktrace.
hunter.actions.VarsPrinter(*names, **options)	An action that prints local variables and optionally global variables visible from the current executing frame.
hunter.actions.VarsSnooper(**options)	A PySnooper-inspired action, similar to *VarsPrinter*, but only show variable changes.

Warning: The following (Predicates and Internals) have Cython implementations in modules prefixed with "_". They should be imported from the hunter module, not hunter.something to be sure you get the best available implementation.

Predicates

hunter.predicates.And(*predicates)	Logical conjunction.
hunter.predicates.Backlog(condition[, size, ...])	Until-point buffering mechanism.
hunter.predicates.From(condition[, ...])	From-point filtering mechanism.
hunter.predicates.Not(predicate)	Logical complement (negation).
hunter.predicates.Or(*predicates)	Logical disjunction.
hunter.predicates.Query(**query)	Event-filtering predicate.
hunter.predicates.When(condition, *actions)	Conditional predicate.

Internals

hunter.event.Event(frame, kind, arg[, ...])	A wrapper object for Frame objects.
hunter.tracer.Tracer([threading_support, ...])	Tracer object.

8.1 Helpers

hunter.**trace**(**predicates*, *clear_env_var=False*, *action=CodePrinter*, *actions=[]*, ***kwargs*)

Starts tracing. Can be used as a context manager (with slightly incorrect semantics - it starts tracing before __enter__ is called).

Parameters ***predicates** (*callables*) – Runs actions if **all** of the given predicates match.

Keyword Arguments

- **clear_env_var** – Disables tracing in subprocess. Default: `False`.
- **threading_support** – Enable tracing *new* threads. Default: `None`.

  Modes:

  - `None` - automatic (enabled but actions only prefix with thread name if more than 1 thread)
  - `False` - completely disabled
  - `True` - enabled (actions always prefix with thread name)

  You can also use: `threads_support`, `thread_support`, `threadingsupport`, `threadssupport`, `threadsupport`, `threading`, `threads` or `thread`.
- **action** – Action to run if all the predicates return `True`. Default: `CodePrinter`.
- **actions** – Actions to run (in case you want more than 1).
- ****kwargs** – for convenience you can also pass anything that you'd pass to *hunter.Q*

See also:

hunter.tracer.Tracer or *hunter.event.Event*

hunter.**stop**()
Stop tracing. Restores previous tracer (if there was any).

hunter.**wrap**(*function_to_trace=None*, ***trace_options*)
Functions decorated with this will be traced.

Use `local=True` to only trace local code, eg:

```
@hunter.wrap(local=True)
def my_function():
    ...
```

Keyword arguments are allowed, eg:

```
@hunter.wrap(action=hunter.CallPrinter)
def my_function():
    ...
```

Or, filters:

```
@hunter.wrap(module='foobar')
def my_function():
    ...
```

hunter.**And**(**predicates*, ***kwargs*)
Helper that flattens out `predicates` in a single *hunter.predicates.And* object if possible. As a convenience it converts `kwargs` to a single *hunter.predicates.Query* instance.

Parameters

- ***predicates** (*callables*) – Callables that returns True/False or *hunter.predicates.Query* objects.
- ****kwargs** – Arguments that may be passed to *hunter.predicates.Query*.

Returns: A *hunter.predicates.And* instance.

See also:

hunter.predicates.And

hunter.**Backlog**(**conditions*, ***kwargs*)

Helper that merges kwargs and conditions prior to creating the *Backlog*.

Parameters

- ***conditions** (*callable*) – Optional *Query* object or a callable that returns True/False.
- **size** (*int*) – Number of events that the backlog stores. Effectively this is the `maxlen` for the internal deque.
- **stack** (*int*) – Stack size to fill. Setting this to `0` disables creating fake call events.
- **vars** (*bool*) – Makes global/local variables available in the stored events. This is an expensive option - it will use `action.try_repr` on all the variables.
- **strip** (*bool*) – If this option is set then the backlog will be cleared every time an event matching the `condition` is found. Disabling this may show more context every time an event matching the `condition` is found but said context may also be duplicated across multiple matches.
- **action** (*ColorStreamAction*) – A ColorStreamAction to display the stored events when an event matching the `condition` is found.
- **filter** (*callable*) – Optional *Query* object or a callable that returns True/False to filter the stored events with.
- ****kwargs** – Arguments that are passed to *hunter.Q()*. Any kwarg that starts with "depth" or "calls" will be included *predicate*.

See also:

hunter.predicates.Backlog

hunter.**From**(*condition=None*, *predicate=None*, *watermark=0*, ***kwargs*)

Helper that converts keyword arguments to `From(condition=Q(**normal_kwargs), predicate=Q(**rel_kwargs)` where `rel_kwargs` are all the kwargs that start with "depth" or "calls".

Parameters

- **condition** (*callable*) – A callable that returns True/False or a *hunter.predicates.Query* object.
- **predicate** (*callable*) – Optional callable that returns True/False or a *hunter.predicates.Query* object to run after `condition` first returns `True`.
- ****kwargs** – Arguments that are passed to *hunter.Q()*. Any kwarg that starts with "depth" or "calls" will be included *predicate*.

Examples

`From(function='foobar', depth_lt=5)` coverts to `From(Q(function='foobar'), Q(depth_lt=5))`. The depth filter is moved in the predicate because it would not have any effect as a condition - it stop being called after it returns True, thus it doesn't have the intended effect (a limit to how deep to trace from `foobar`).

See also:

hunter.predicates.From

hunter.**Not**(**predicates*, ***kwargs*)

Helper that flattens out `predicates` in a single *hunter.predicates.And* object if possible. As a convenience it converts `kwargs` to multiple *hunter.predicates.Query* instances.

Parameters

- ***predicates** (*callables*) – Callables that returns True/False or *hunter.predicates.Query* objects.
- ****kwargs** – Arguments that may be passed to *hunter.predicates.Query*.

Returns: A *hunter.predicates.Not* instance (possibly containing a *hunter.predicates.And* instance).

See also:

hunter.predicates.Not

hunter.**Or**(**predicates*, ***kwargs*)

Helper that flattens out `predicates` in a single *hunter.predicates.Or* object if possible. As a convenience it converts `kwargs` to multiple *hunter.predicates.Query* instances.

Parameters

- ***predicates** (*callables*) – Callables that returns True/False or *hunter.predicates.Query* objects.
- ****kwargs** – Arguments that may be passed to *hunter.predicates.Query*.

Returns: A *hunter.predicates.Or* instance.

See also:

hunter.predicates.Or

hunter.**Q**(**predicates*, ***query*)

Helper that handles situations where *hunter.predicates.Query* objects (or other callables) are passed in as positional arguments - it conveniently converts those to a *hunter.predicates.And* predicate.

See also:

hunter.predicates.Query

8.2 Actions

class hunter.actions.**CallPrinter**(*stream=sys.stderr*, *force_colors=False*, *force_pid=False*, *filename_alignment=40*, *thread_alignment=12*, *pid_alignment=9*, *repr_limit=1024*, *repr_func='safe_repr'*)

An action that just prints the code being executed, but unlike `hunter.CodePrinter` it indents based on callstack depth and it also shows `repr()` of function arguments.

Parameters

- **stream** (*file-like*) – Stream to write to. Default: `sys.stderr`.
- **filename_alignment** (*int*) – Default size for the filename column (files are right-aligned). Default: `40`.
- **force_colors** (*bool*) – Force coloring. Default: `False`.
- **repr_limit** (*bool*) – Limit length of `repr()` output. Default: `512`.

- **repr_func** (*string or callable*) – Function to use instead of `repr`. If string must be one of 'repr' or 'safe_repr'. Default: `'safe_repr'`.

New in version 1.2.0.

__call__(*event*)
Handle event and print filename, line number and source code. If event.kind is a *return* or *exception* also prints values.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class hunter.actions.**CodePrinter**(*stream=sys.stderr*, *force_colors=False*, *force_pid=False*, *filename_alignment=40*, *thread_alignment=12*, *pid_alignment=9*, *repr_limit=1024*, *repr_func='safe_repr'*)
An action that just prints the code being executed.

Parameters

- **stream** (*file-like*) – Stream to write to. Default: `sys.stderr`.
- **filename_alignment** (*int*) – Default size for the filename column (files are right-aligned). Default: `40`.
- **force_colors** (*bool*) – Force coloring. Default: `False`.
- **repr_limit** (*bool*) – Limit length of `repr()` output. Default: `512`.
- **repr_func** (*string or callable*) – Function to use instead of `repr`. If string must be one of 'repr' or 'safe_repr'. Default: `'safe_repr'`.

__call__(*event*)
Handle event and print filename, line number and source code. If event.kind is a *return* or *exception* also prints values.

class hunter.actions.**ColorStreamAction**(*stream=sys.stderr*, *force_colors=False*, *force_pid=False*, *filename_alignment=40*, *thread_alignment=12*, *pid_alignment=9*, *repr_limit=1024*, *repr_func='safe_repr'*)
Baseclass for your custom action. Just implement your own `__call__`.

__eq__(*other*)
Return self==value.

__init__(*stream=None*, *force_colors=False*, *force_pid=False*, *filename_alignment=40*, *thread_alignment=12*, *pid_alignment=9*, *repr_limit=1024*, *repr_func='safe_repr'*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

__str__()
Return str(self).

filename_prefix(*event=None*)
Get an aligned and trimmed filename prefix for the given `event`.

Returns: string

output(*format_str*, **args*, ***kwargs*)
Write `format_str.format(*args, **ANSI_COLORS, **kwargs)` to `self.stream`.

For ANSI coloring you can place these in the `format_str`:

- `{BRIGHT}`

- {DIM}
- {NORMAL}
- {RESET}
- {fore(BLACK)}
- {fore(RED)}
- {fore(GREEN)}
- {fore(YELLOW)}
- {fore(BLUE)}
- {fore(MAGENTA)}
- {fore(CYAN)}
- {fore(WHITE)}
- {fore(RESET)}
- {back(BLACK)}
- {back(RED)}
- {back(GREEN)}
- {back(YELLOW)}
- {back(BLUE)}
- {back(MAGENTA)}
- {back(CYAN)}
- {back(WHITE)}
- {back(RESET)}

Parameters

- **format_str** – a PEP-3101 format string
- ***args**
- ****kwargs**

Returns: string

pid_prefix()
Get an aligned and trimmed pid prefix.

thread_prefix(*event*)
Get an aligned and trimmed thread prefix for the given `event`.

try_repr(*obj*)
Safely call `self.repr_func(obj)`. Failures will have special colored output and output is trimmed according to `self.repr_limit`.

Returns: string

try_source(*event*, *full=False*)
Get a failure-colorized source for the given `event`.

Return: string

try_str(*obj*)
Safely call `str(obj)`. Failures will have special colored output and output is trimmed according to `self.repr_limit`.

Only used when dumping detached events.

Returns: string

class hunter.actions.**Debugger**(*klass=pdb.Pdb*, ***kwargs*)
An action that starts `pdb`.

__call__(*event*)
Runs a `pdb.set_trace` at the matching frame.

__eq__(*other*)
Return self==value.

__init__(*klass=<class 'pdb.Pdb'>*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

__str__()
Return str(self).

class hunter.actions.**ErrorSnooper**(*max_events=50*, *max_depth=1*, *stream=sys.stderr*, *force_colors=False*, *force_pid=False*, *filename_alignment=40*, *thread_alignment=12*, *pid_alignment=9*, *repr_limit=1024*, *repr_func='safe_repr'*)
An action that prints events around silenced exceptions. Note that it inherits the output of *CodePrinter* so no fancy call indentation.

Warning: Should be considered experimental. May show lots of false positives especially if you're tracing lots of clumsy code like:

```
try:
    stuff = something[key]
except KeyError:
    stuff = "default"
```

Parameters

- **max_backlog** (*int*) – Maximum number of events to record and display before the silenced exception is raised. Set to 0 to disable and get a speed boost. Default: `10`.
- **max_events** (*int*) – Maximum number of events to record and display for each detected silenced exception. Default: `50`.
- **max_depth** (*int*) – Increase if you want to drill into subsequent calls after an exception is raised. If you increase this you might want to also increase `max_events` since subsequent calls may have so many events you won't get to see the return event. Default: `0` (doesn't drill into any calls).
- **stream** (*file-like*) – Stream to write to. Default: `sys.stderr`.
- **filename_alignment** (*int*) – Default size for the filename column (files are right-aligned). Default: `40`.
- **force_colors** (*bool*) – Force coloring. Default: `False`.

- **repr_limit** (*bool*) – Limit length of `repr()` output. Default: `512`.
- **repr_func** (*string or callable*) – Function to use instead of `repr`. If string must be one of 'repr' or 'safe_repr'. Default: `'safe_repr'`.

New in version 3.1.0.

__call__(*event*)
Handle event and print filename, line number and source code. If event.kind is a *return* or *exception* also prints values.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class hunter.actions.**Manhole**(***options*)

__call__(*event*)
Call self as a function.

__eq__(*other*)
Return self==value.

__init__(***options*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

__str__()
Return str(self).

class hunter.actions.**StackPrinter**(*depth=15*, *limit=2*, *stream=sys.stderr*, *force_colors=False*, *force_pid=False*, *filename_alignment=40*, *thread_alignment=12*, *pid_alignment=9*, *repr_limit=1024*, *repr_func='safe_repr'*)
An action that prints a one-line stacktrace.

Parameters

- **depth** (*int*) – The maximum number of frames to show.
- **limit** (*int*) – The maximum number of components to show in path. Eg: `limit=2` means it will show 1 parent: `foo/bar.py`.
- **stream** (*file-like*) – Stream to write to. Default: `sys.stderr`.
- **filename_alignment** (*int*) – Default size for the filename column (files are right-aligned). Default: `40`.
- **force_colors** (*bool*) – Force coloring. Default: `False`.
- **repr_limit** (*bool*) – Limit length of `repr()` output. Default: `512`.
- **repr_func** (*string or callable*) – Function to use instead of `repr`. If string must be one of 'repr' or 'safe_repr'. Default: `'safe_repr'`.

__call__(*event*)
Handle event and print the stack.

__init__(*depth=15*, *limit=2*, ***options*)
Initialize self. See help(type(self)) for accurate signature.

class hunter.actions.**VarsPrinter**(*name*[, *name*[, *name*[, *...*]]], *stream=sys.stderr*, *force_colors=False*, *force_pid=False*, *filename_alignment=40*, *thread_alignment=12*, *pid_alignment=9*, *repr_limit=1024*, *repr_func='safe_repr'*)

An action that prints local variables and optionally global variables visible from the current executing frame.

Parameters

- ***names** (*strings*) – Names to evaluate. Expressions can be used (will only try to evaluate if all the variables are present on the frame.
- **stream** (*file-like*) – Stream to write to. Default: `sys.stderr`.
- **filename_alignment** (*int*) – Default size for the filename column (files are right-aligned). Default: `40`.
- **force_colors** (*bool*) – Force coloring. Default: `False`.
- **repr_limit** (*bool*) – Limit length of `repr()` output. Default: `512`.
- **repr_func** (*string or callable*) – Function to use instead of `repr`. If string must be one of 'repr' or 'safe_repr'. Default: `'safe_repr'`.

__call__(*event*)

Handle event and print the specified variables.

__init__(**names*, ***options*)

Initialize self. See help(type(self)) for accurate signature.

class hunter.actions.**VarsSnooper**(*stream=sys.stderr*, *force_colors=False*, *force_pid=False*, *filename_alignment=40*, *thread_alignment=12*, *pid_alignment=9*, *repr_limit=1024*, *repr_func='safe_repr'*)

A PySnooper-inspired action, similar to *VarsPrinter*, but only show variable changes.

> **Warning:** Should be considered experimental. Use judiciously.
>
> - It stores reprs for all seen variables, therefore it can use lots of memory.
> - Will leak memory if you filter the return events (eg: `~Q(kind="return")`).
> - Not thoroughly tested. May misbehave on code with closures/nonlocal variables.

Parameters

- **stream** (*file-like*) – Stream to write to. Default: `sys.stderr`.
- **filename_alignment** (*int*) – Default size for the filename column (files are right-aligned). Default: `40`.
- **force_colors** (*bool*) – Force coloring. Default: `False`.
- **repr_limit** (*bool*) – Limit length of `repr()` output. Default: `512`.
- **repr_func** (*string or callable*) – Function to use instead of `repr`. If string must be one of 'repr' or 'safe_repr'. Default: `'safe_repr'`.

__call__(*event*)

Handle event and print the specified variables.

__init__(***options*)

Initialize self. See help(type(self)) for accurate signature.

8.3 Predicates

Warning: These have Cython implementations in modules prefixed with "_".

Note that:

- Every predicate except *When* has a *helper* importable directly from the `hunter` package.
- Ideally you'd use the helpers instead of these to get the best available implementation, extra validation and better argument handling.

class hunter.predicates.**And**(**predicates*)

Logical conjunction. Returns `False` at the first sub-predicate that returns `False`, otherwise returns `True`.

__and__(*other*)

Convenience API so you can do `And(...) & other`. It converts that to `And(..., other)`.

__call__(*event*)

Handles the event.

__eq__(*other*)

Return self==value.

__init__(**predicates*)

Initialize self. See help(type(self)) for accurate signature.

__invert__()

Convenience API so you can do `~And(...)`. It converts that to `Not(And(...))`.

__or__(*other*)

Convenience API so you can do `And(...) | other`. It converts that to `Or(And(...), other)`.

__rand__(*other*)

Convenience API so you can do `other & And(...)`. It converts that to `And(other, And(...))`.

__repr__()

Return repr(self).

__ror__(*other*)

Convenience API so you can do `other | And(...)`. It converts that to `Or(other, And(...))`.

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

class hunter.predicates.**Backlog**(*condition*, *size=100*, *stack=10*, *vars=False*, *strip=True*, *action=None*, *filter=None*)

Until-point buffering mechanism. It will buffer detached events up to the given `size` and display them using the given `action` when `condition` returns True.

This is a complement to *From* - essentially working the other way. While *From* shows events after something interesting occurred the Backlog will show events prior to something interesting occurring.

If the depth delta from the first event in the backlog and the event that matched the condition is less than the given `stack` then it will create fake call events to be passed to the action before the events from the backlog are passed in.

Using a `filter` or pre-filtering is recommended to reduce storage work and improve tracing speed. Pre-filtering means that you use Backlog inside a `When` or *:class:'~hunter.And* - effectively reducing the number of Events that get to the Backlog.

Parameters

- **condition** (*callable*) – Optional *Query* object or a callable that returns True/False.
- **size** (*int*) – Number of events that the backlog stores. Effectively this is the `maxlen` for the internal deque.
- **stack** (*int*) – Stack size to fill. Setting this to `0` disables creating fake call events.
- **vars** (*bool*) – Makes global/local variables available in the stored events. This is an expensive option - it will use `action.try_repr` on all the variables.
- **strip** (*bool*) – If this option is set then the backlog will be cleared every time an event matching the `condition` is found. Disabling this may show more context every time an event matching the `condition` is found but said context may also be duplicated across multiple matches.
- **action** (*ColorStreamAction*) – A ColorStreamAction to display the stored events when an event matching the `condition` is found.
- **filter** (*callable*) – Optional *Query* object or a callable that returns True/False to filter the stored events with.

See also:

hunter.predicates.From

__and__(*other*)
Convenience API so you can do `Backlog(...) & other`. It converts that to `And(Backlog(...), other)`.

__call__(*event*)
Handles the event.

__eq__(*other*)
Return self==value.

__init__(*condition*, *size=100*, *stack=10*, *vars=False*, *strip=True*, *action=None*, *filter=None*)
Initialize self. See help(type(self)) for accurate signature.

__invert__()
Convenience API so you can do `~Backlog(...)`. It converts that to `Not(Backlog(...))`.

__or__(*other*)
Convenience API so you can do `Backlog(...) | other`. It converts that to `Or(Backlog(...), other)`.

__rand__(*other*)
Convenience API so you can do `other & Backlog(...)`. It converts that to `And(other, Backlog(...))`.

__repr__()
Return repr(self).

__ror__(*other*)
Convenience API so you can do `other | Backlog(...)`. It converts that to `Or(other, Backlog(...))`.

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

filter(**predicates*, ***kwargs*)
Returns another Backlog instance with extra output filtering. If the current instance already have filters they will be merged by using an *And* predicate.

Parameters

- ***predicates** (*callables*) – Callables that returns True/False or *Query* objects.
- ****kwargs** – Arguments that may be passed to *Query*.

Returns: A new *Backlog* instance.

class hunter.predicates.**From**(*condition*, *predicate=None*, *watermark=0*)
From-point filtering mechanism. Switches on to running the predicate after condition matches, and switches off when the depth goes lower than the initial level.

After `condition(event)` returns `True` the `event.depth` will be saved and calling this object with an `event` will return `predicate(event)` until `event.depth - watermark` is equal to the depth that was saved.

Parameters

- **condition** (*callable*) – Optional *Query* object or a callable that returns True/False.
- **predicate** (*callable*) – Optional *Query* object or a callable that returns True/False to run after `condition` first returns `True`. Note that this predicate will be called with a event-copy that has adjusted *depth* and *calls* to the initial point where the `condition` matched. In other words they will be relative.
- **watermark** (*int*) – Depth difference to switch off and wait again on `condition`.

See also:

hunter.predicates.Backlog

__and__(*other*)
Convenience API so you can do `From(...) & other`. It converts that to `And(From(...), other)`.

__call__(*event*)
Handles the event.

__eq__(*other*)
Return self==value.

__init__(*condition*, *predicate=None*, *watermark=0*)
Initialize self. See help(type(self)) for accurate signature.

__invert__()
Convenience API so you can do `~From(...)`. It converts that to `Not(From(...))`.

__or__(*other*)
Convenience API so you can do `From(...) | other`. It converts that to `Or(From(...), other)`.

__rand__(*other*)
Convenience API so you can do `other & From(...)`. It converts that to `And(other, From(...))`.

__repr__()
Return repr(self).

__ror__(*other*)
Convenience API so you can do `other | From(...)`. It converts that to `Or(other, From(...))`.

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

class hunter.predicates.**Not**(*predicate*)
Logical complement (negation). Simply returns `not predicate(event)`.

__and__(*other*)
Convenience API so you can do `Not(...) & other`. It converts that to `And(Not(...), other)`.

Note that `Not(...) & Not(...)` converts to `Not(Or(..., ...))`.

__call__(*event*)
Handles the event.

__eq__(*other*)
Return self==value.

__init__(*predicate*)
Initialize self. See help(type(self)) for accurate signature.

__invert__()
Convenience API so you can do `~Not(...)`. It converts that to `...`.

__or__(*other*)
Convenience API so you can do `Not(...) | other`. It converts that to `Or(Not(...), other)`.

Note that `Not(...) | Not(...)` converts to `Not(And(..., ...))`.

__rand__(*other*)
Convenience API so you can do `other & Not(...)`. It converts that to `And(other, Not(...))`.

__repr__()
Return repr(self).

__ror__(*other*)
Convenience API so you can do `other | Not(...)`. It converts that to `Or(other, Not(...))`.

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

class hunter.predicates.**Or**(**predicates*)
Logical disjunction. Returns `True` after the first sub-predicate that returns `True`.

__and__(*other*)
Convenience API so you can do `Or(...) & other`. It converts that to `And(Or(...), other)`.

__call__(*event*)
Handles the event.

__eq__(*other*)
Return self==value.

__init__(**predicates*)
Initialize self. See help(type(self)) for accurate signature.

__invert__()
Convenience API so you can do `~Or(...)`. It converts that to `Not(Or(...))`.

__or__(*other*)
Convenience API so you can do `Or(...) | other`. It converts that to `Or(..., other)`.

__rand__(*other*)
Convenience API so you can do `other & Or(...)`. It converts that to `And(other, Or(...))`.

__repr__()
Return repr(self).

__ror__(*other*)
Convenience API so you can do `other | Or(...)`. It converts that to `Or(other, Or(...))`.

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

class hunter.predicates.**Query**(***query*)
Event-filtering predicate.

See *hunter.event.Event* for details about the fields that can be filtered on.

Parameters **query** – criteria to match on.

Accepted arguments: `arg`, `builtin`, `calls`, `code`, `depth`, `filename`, `frame`, `fullsource`, `function`, `globals`, `kind`, `lineno`, `locals`, `module`, `source`, `stdlib`, `threadid`, `threadname`.

__and__(*other*)
Convenience API so you can do `Query(...) & Query(...)`. It converts that to `And(Query(...), Query(...))`.

__call__(*event*)
Handles event. Returns True if all criteria matched.

__eq__(*other*)
Return self==value.

__init__(***query*)
Initialize self. See help(type(self)) for accurate signature.

__invert__()
Convenience API so you can do `~Query(...)`. It converts that to `Not(Query(...))`.

__or__(*other*)
Convenience API so you can do `Query(...) | Query(...)`. It converts that to `Or(Query(...), Query(...))`.

__rand__(*other*)
Convenience API so you can do `other & Query(...)`. It converts that to `And(other, Query(...))`.

__repr__()
Return repr(self).

__ror__(*other*)
Convenience API so you can do `other | Query(...)`. It converts that to `Or(other, Query(...))`.

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

class hunter.predicates.**When**(*condition*, **actions*)
Conditional predicate. Runs `actions` when `condition(event)` is `True`.

Actions take a single `event` argument.

__and__(*other*)
Convenience API so you can do `When(...) & other`. It converts that to `And(When(...), other)`.

__call__(*event*)
Handles the event.

__eq__(*other*)
Return self==value.

__init__(*condition*, **actions*)
Initialize self. See help(type(self)) for accurate signature.

__invert__()
Convenience API so you can do `~When(...)`. It converts that to `Not(When(...))`.

__or__(*other*)
Convenience API so you can do `When(...) | other`. It converts that to `Or(When(...), other)`.

__rand__(*other*)
Convenience API so you can do `other & When(...)`. It converts that to `And(other, When(...))`.

__repr__()
Return repr(self).

__ror__(*other*)
Convenience API so you can do `other | When(...)`. It converts that to `Or(other, When(...))`.

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

8.4 Internals

Warning: These have Cython implementations in modules prefixed with "_". They should be imported from the `hunter` module, not `hunter.something` to be sure you get the best available implementation.

Normally these are not used directly. Perhaps just the *Tracer* may be used directly for performance reasons.

class hunter.event.**Event**(*frame*, *kind*, *arg*, *tracer=None*, *depth=None*, *calls=None*, *threading_support=?*)

A wrapper object for Frame objects. Instances of this are passed to your custom functions or predicates.

Provides few convenience properties.

Parameters

- **frame** (*Frame*) – A python Frame object.
- **kind** (*str*) – A string like `'call'`, `'line'`, `'return'` or `'exception'`.
- **arg** – A value that depends on `kind`. Usually is `None` but for `'return'` or `'exception'` other values may be expected.
- **tracer** (*hunter.tracer.Tracer*) – The *Tracer* instance that created the event. Needed for the `calls` and `depth` fields.

__eq__(*other*)
Return self==value.

__getitem__
Return getattr(self, name).

__init__(*frame*, *kind*, *arg*, *tracer=None*, *depth=None*, *calls=None*, *threading_support=?*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

__weakref__
list of weak references to the object (if defined)

arg = None
A value that depends on `kind`

builtin = None
If kind of the event is one of `'c_call'`, `'c_return'`, or `'c_exception'` then this will be True.

Type bool

calls = None
A counter for total number of calls up to this Event.

Type int

code
A code object (not a string).

depth = None
Tracing depth (increases on calls, decreases on returns).

Type int

detach(*value_filter=None*)
Return a copy of the event with references to live objects (like the frame) removed. You should use this if you want to store or use the event outside the handler.

You should use this if you want to avoid memory leaks or side-effects when storing the events.

Parameters **value_filter** – Optional callable that takes one argument: `value`.

If not specified then the `arg`, `globals` and `locals` fields will be `None`.

Example usage in a *ColorStreamAction* subclass:

```
def __call__(self, event):
    self.events = [event.detach(lambda field, value: self.try_repr(value))]
```

detached = None
Flag that is `True` if the event was created with *detach()*.

Type bool

filename
A string with the path to the module's file. May be empty if `__file__` attribute is missing. May be relative if running scripts.

Type str

frame = None
The original Frame object.

Note: Not allowed in the builtin predicates (it's the actual Frame object). You may access it from your custom predicate though.

---

fullsource
A string with the sourcecode for the current statement (from `linecache` - failures are ignored).

May include multiple lines if it's a class/function definition (will include decorators).

Type str

function
A string with function name.

Type str

function_object
The function instance.

Warning: Use with prudence.

- Will be `None` for decorated functions on Python 2 (methods may still work tho).
- May be `None` if tracing functions or classes not defined at module level.
- May be very slow if tracing modules with lots of variables.

Type function or None

globals
A dict with global variables.

Type dict

instruction
Last byte instruction. If no bytecode was used (Cython code) then it returns `None`. Depending on Python version it might be an int or a single char string.

Type int or single char string or None

kind = None
The kind of the event, could be one of `'call'`, `'line'`, `'return'`, `'exception'`.

Type str

lineno
An integer with line number in file.

Type int

locals
A dict with local variables.

Type dict

module
A string with module name (like `'foo.bar'`).

Type str

source
A string with the sourcecode for the current line (from `linecache` - failures are ignored).

Fast but sometimes incomplete.

Type str

stdlib
A boolean flag. `True` if frame is in stdlib.

Type bool

threadid
Current thread ident. If current thread is main thread then it returns `None`.

Type int or None

threading_support = None
A copy of the *hunter.tracer.Tracer.threading_support* flag.

Note: Not allowed in the builtin predicates. You may access it from your custom predicate though.

Type bool or None

threadname
Current thread name.

Type str

class hunter.tracer.**Tracer**(*threading_support=None*, *profiling_mode=False*)
Tracer object.

Parameters **threading_support** (*bool*) – Hooks the tracer into `threading.settrace` as well if True.

__call__(*frame*, *kind*, *arg*)
The settrace function.

Note: This always returns self (drills down) - as opposed to only drilling down when `predicate(event)` is True because it might match further inside.

__enter__()
Does nothing. Users are expected to call *trace()*.

Returns: self

__exit__(*exc_type*, *exc_val*, *exc_tb*)
Wrapper around *stop()*. Does nothing with the arguments.

__init__(*threading_support=None*, *profiling_mode=False*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

__weakref__
list of weak references to the object (if defined)

calls = None
A counter for total number of 'call' frames that this Tracer went through.

Type int

depth = None
Tracing depth (increases on calls, decreases on returns)

Type int

handler
The current predicate. Set via `hunter.Tracer.trace()`.

previous
The previous tracer, if any (whatever `sys.gettrace()` returned prior to `hunter.Tracer.trace()`).

profiling_mode = None
True if profiling mode was enabled. Should be considered read-only.

Type bool

stop()
Stop tracing. Reinstalls the *previous* tracer.

threading_support = None
True if threading support was enabled. Should be considered read-only.

Type bool

trace(*predicate*)
Starts tracing with the given callable.

Parameters **predicate** (callable that accepts a single *Event* argument)

Returns self

CHAPTER 9

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

9.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

9.2 Documentation improvements

Hunter could always use more documentation, whether as part of the official Hunter docs, in docstrings, or even on the web in blog posts, articles, and such.

9.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/ionelmc/python-hunter/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

9.4 Development

To set up *python-hunter* for local development:

1. Fork python-hunter (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:YOURGITHUBNAME/python-hunter.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes run all the checks and docs builder with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

9.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

9.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel*:

```
tox -p auto
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 10

Authors

- Ionel Cristian Mărieș - https://blog.ionelmc.ro
- Claudiu Popa - https://github.com/PCManticore
- Mikhail Borisov - https://github.com/borman
- Dan Ailenei - https://github.com/Dan-Ailenei
- Tom Schraitle - https://github.com/tomschr

CHAPTER 11

Changelog

11.1 3.3.2 (2021-03-25)

- Changed CI to build Python 3.9 wheels. Python 3.5 no longer tested and wheels no longer built to keep things simple.
- Documentation improvements.

11.2 3.3.1 (2020-10-24)

- Fixed CI/test issues that prevented all of 21 wheels being published.

11.3 3.3.0 (2020-10-23)

- Fixed handling so that *hunter.event.Event.module* is always the `"?"` string instead of *None*. Previously it was `None` when tracing particularly broken code and broke various predicates.
- Similarly *hunter.event.Event.filename* is now `"?"` if there's no filename available.
- Building on the previous changes the actions have simpler code for displaying missing module/filenames.
- Changed *hunter.actions.CallPrinter* so that trace events for builtin functions are displayed differently. These events appear when using profile mode (eg: `trace(profile=True)`).
- Fixed failure that could occur if *hunter.event.Event.module* is an unicode string. Now it's always a regular string. *Only applies to Python 2.*
- Fixed argument display when tracing functions with tuple arguments. Closes #88. *Only applies to Python 2.*
- Improved error reporting when internal failures occur. Now some details about the triggering event are logged.

11.4 3.2.2 (2020-09-04)

- Fixed oversight over what value is in *hunter.event.Event.builtin*. Now it's always a boolean, and can be used consistently in filters (eg: `builtin=True,function='getattr'`).

11.5 3.2.1 (2020-08-18)

- Added support for regex, date and datetime in `safe_repr`.
- Fixed call argument display when positional and keyword arguments are used in *hunter.actions.CallPrinter*.

11.6 3.2.0 (2020-08-16)

- Implemented the *StackPrinter* action.
- Implemented the *Backlog* predicate. Contributed by Dan Ailenei in #81.
- Improved contributing section in docs a bit. Contributed by Tom Schraitle in #85.
- Improved filtering performance by avoiding a lot of unnecessary `PyObject_GetAttr` calls in the Cython implementation of *Backlog*.
- Implemented the *ErrorSnooper* action.
- Added support for profiling mode (eg: `trace(profile=True)`). This mode will use `setprofile` instead of `settrace`.
- Added ARM64 wheels and CI.
- Added *hunter.event.Event.instruction* and *hunter.event.Event.builtin* (usable in profile mode).
- Added more cookbook entries.

11.7 3.1.3 (2020-02-02)

- Improved again the stdlib check to handle certain paths better.

11.8 3.1.2 (2019-01-19)

- Really fixed the `<frozen importlib.something` stdlib check.

11.9 3.1.1 (2019-01-19)

- Marked all the `<frozen importlib.something` files as part of stdlib.

11.10 3.1.0 (2019-01-19)

- Added *ErrorSnooper* - an action that detects silenced exceptions.
- Added `load_config()` and fixed issues with configuration being loaded too late from the `PYTHONHUNTERCONFIG` environment variable.
- Changed *From()* helper to automatically move `depth` and `calls` filters to the predicate (so they filter after *From* activates).
- Changed *From* to pass a copy of event to the predicate. The copy will have the `depth` and `calls` attributes adjusted to the point where *From* activated.
- Fixed a bunch of inconsistencies and bugs when using `&` and `|` operators with predicates.
- Fixed a bunch of broken fields on *detached events* (*function_object* and *arg*).
- Improved docstrings in various and added a configuration doc section.
- Improved testing (more coverage).

11.11 3.0.5 (2019-12-06)

- Really fixed `safe_repr` so it doesn't cause side-effects (now isinstance/issubclass are avoided - they can cause side-effects in code that abuses descriptors in special attributes/methods).

11.12 3.0.4 (2019-10-26)

- Really fixed `stream` setup in actions (using `force_colors` without any `stream` was broken). See: *ColorStreamAction*.
- Fixed `__repr__` for the *From* predicate to include `watermark`.
- Added binary wheels for Python 3.8.

11.13 3.0.3 (2019-10-13)

- Fixed `safe_repr` on pypy so it's safer on method objects. See: *ColorStreamAction*.

11.14 3.0.2 (2019-10-10)

- Fixed setting `stream` from `PYTHONHUNTERCONFIG` environment variable. See: *ColorStreamAction*.
- Fixed a couple minor documentation issues.

11.15 3.0.1 (2019-06-17)

- Fixed issue with coloring missing source message (coloring leaked into next line).

11.16 3.0.0 (2019-06-17)

- The package now uses setuptools-scm for development builds (available at https://test.pypi.org/project/hunter/). As a consequence installing the sdist will download setuptools-scm.
- Recompiled cython modules with latest Cython. Hunter can be installed without any Cython, as before.
- Refactored some of the cython modules to have more typing information and not use deprecated property syntax.
- Replaced `unsafe_repr` option with `repr_func`. Now you can use your custom repr function in the builtin actions. **BACKWARDS INCOMPATIBLE**
- Fixed buggy filename handling when using Hunter in ipython/jupyter. Source code should be properly displayed now.
- Removed `globals` option from `VarsPrinter` action. Globals are now always looked up. **BACKWARDS INCOMPATIBLE**
- Added support for locals in `VarsPrinter` action. Now you can do `VarsPrinter('len(foobar)')`.
- Always pass module_globals dict to linecache methods. Source code from PEP-302 loaders is now printed properly. Contributed by Mikhail Borisov in #65.
- Various code cleanup, style and docstring fixing.
- Added *`hunter.From()`* helper to allow passing in filters directly as keyword arguments.
- Added *`hunter.event.Event.detach()`* for storing events without leaks or side-effects (due to prolonged references to Frame objects, local or global variables).
- Refactored the internals of actions for easier subclassing.

  Added the *`filename_prefix()`*, *`output()`*, *`pid_prefix()`*, *`thread_prefix()`*, *`try_repr()`* and *`try_source()`* methods to the *`hunter.actions.ColorStreamAction`* baseclass.
- Added *`hunter.actions.VarsSnooper`* - a PySnooper-inspired variant of *`VarsPrinter`*. It will record and show variable changes, with the risk of leaking or using too much memory of course :)
- Fixed tracers to log error and automatically stop if there's an internal failure. Previously error may have been silently dropped in some situations.

11.17 2.2.1 (2019-01-19)

- Fixed a link in changelog.
- Fixed some issues in the Travis configuration.

11.18 2.2.0 (2019-01-19)

- Added *`hunter.predicates.From`* predicate for tracing from a specific point. It stop after returning back to the same call depth with a configurable offset.
- Fixed `PYTHONHUNTERCONFIG` not working in some situations (config values were resolved at the wrong time).
- Made tests in CI test the wheel that will eventually be published to PyPI (tox-wheel).
- Made `event.stdlib` more reliable: `pkg_resources` is considered part of stdlib and few more paths will be considered as stdlib.

- Dumbed down the `get_peercred` check that is done when attaching with `hunter-trace` CLI (via `hunter.remote.install()`). It will be slightly insecure but will work on OSX.
- Added OSX in the Travis test grid.

11.19 2.1.0 (2018-11-17)

- Made `threading_support` on by default but output automatic (also, now `1` or `0` allowed).
- Added `pid_alignment` and `force_pid` action options to show a pid prefix.
- Fixed some bugs around `__eq__` in various classes.
- Dropped Python 3.3 support.
- Dropped dependency on fields.
- Actions now repr using a simplified implementation that tries to avoid calling `__repr__` on user classes in order to avoid creating side-effects while tracing.
- Added support for the `PYTHONHUNTERCONFIG` environment variable (stores defaults and doesn't activate hunter).

11.20 2.0.2 (2017-11-24)

- Fixed indentation in *`hunter.actions.CallPrinter`* action (shouldn't deindent on exception).
- Fixed option filtering in Cython Query implementation (filtering on `tracer` was allowed by mistake).
- Various fixes to docstrings and docs.

11.21 2.0.1 (2017-09-09)

- Now `Py_AddPendingCall` is used instead of acquiring the GIL (when using GDB).

11.22 2.0.0 (2017-09-02)

- Added the `hunter.event.Event.count` and `hunter.event.Event.calls`` attributes.
- Added the `lt`/`lte`/`gt`/`gte` lookups.
- Added convenience aliases for `startswith` (`sw`), `endswith` (`ew`), `contains` (`has`) and `regex` (`rx`).
- Added a convenience *`hunter.wrap()`* decorator to start tracing around a function.
- Added support for remote tracing (with two backends: manhole and GDB) via the `hunter-trace` bin. Note: **Windows is NOT SUPPORTED**.
- Changed the default action to *`hunter.actions.CallPrinter`*. You'll need to use `action=CodePrinter` if you want the old output.

11.23 1.4.1 (2016-09-24)

- Fix support for getting sources for Cython module (it was broken on Windows and Python3.5+).

11.24 1.4.0 (2016-09-24)

- Added support for tracing Cython modules (#30). A *# cython: linetrace=True* stanza or equivalent is required in Cython modules for this to work.

11.25 1.3.0 (2016-04-14)

- Added `hunter.event.Event.thread`.
- Added *hunter.event.Event.threadid* and *hunter.event.Event.threadname* (available for filtering with *hunter.Q()*).
- Added *hunter.event.Event.threading_support* argument to *hunter.trace()*. It makes new threads be traced and changes action output to include thread name.
- Added support for using pdb++ in the *hunter.actions.Debugger* action.
- Added support for using manhole via a new *hunter.actions.Manhole* action.
- Made the `hunter.event.Event.handler` a public but readonly property.

11.26 1.2.2 (2016-01-28)

- Fix broken import. Require *fields>=4.0*.
- Simplify a string check in Cython code.

11.27 1.2.1 (2016-01-27)

- Fix "KeyError: 'normal'" bug in *hunter.actions.CallPrinter*. Create the NO_COLORS dict from the COLOR dicts. Some keys were missing.

11.28 1.2.0 (2016-01-24)

- Fixed printouts of objects that return very large string in `__repr__()`. Trimmed to 512. Configurable in actions with the `repr_limit` option.
- Improved validation of *hunter.actions.VarsPrinter*'s initializer.
- Added a *hunter.actions.CallPrinter* action.

11.29 1.1.0 (2016-01-21)

- Implemented a destructor (`__dealloc__`) for the Cython tracer.
- Improved the restoring of the previous tracer in the Cython tracer (use `PyEval_SetTrace`) directly.
- Removed `tracer` as an allowed filtering argument in `hunter.Query`.
- Add basic validation (must be callable) for positional arguments and actions passed into `hunter.Q`. Closes #23.
- Fixed `stdlib` checks (wasn't very reliable). Closes #24.

11.30 1.0.2 (2016-01-05)

- Fixed missing import in `setup.py`.

11.31 1.0.1 (2015-12-24)

- Fix a compile issue with the MSVC compiler (seems it don't like the inline option on the `fast_When_call`).

11.32 1.0.0 (2015-12-24)

- Implemented fast tracer and query objects in Cython. **MAY BE BACKWARDS INCOMPATIBLE**

  To force using the old pure-python implementation set the `PUREPYTHONHUNTER` environment variable to non-empty value.
- Added filtering operators: `contains`, `startswith`, `endswith` and `in`. Examples:
  - `Q(module_startswith='foo'` will match events from `foo`, `foo.bar` and `foobar`.
  - `Q(module_startswith=['foo', 'bar']` will match events from `foo`, `foo.bar`, `foobar`, `bar`, `bar.foo` and `baroo` .
  - `Q(module_endswith='bar'` will match events from `foo.bar` and `foobar`.
  - `Q(module_contains='ip'` will match events from `lipsum`.
  - `Q(module_in=['foo', 'bar']` will match events from `foo` and `bar`.
  - `Q(module_regex=r"(re|sre.*)\b")` will match events from ``re`, `re.foobar`, `srefoobar` but not from `repr`.
- Removed the `merge` option. Now when you call `hunter.trace(...)` multiple times only the last one is active. **BACKWARDS INCOMPATIBLE**
- Remove the *previous_tracer handling*. Now when you call `hunter.trace(...)` the previous tracer (whatever was in `sys.gettrace()`) is disabled and restored when `hunter.stop()` is called. **BACKWARDS INCOMPATIBLE**
- Fixed `CodePrinter` to show module name if it fails to get any sources.

11.33 0.6.0 (2015-10-10)

- Added a `clear_env_var` option on the tracer (disables tracing in subprocess).
- Added `force_colors` option on *hunter.actions.VarsPrinter* and *hunter.actions.CodePrinter*.
- Allowed setting the *stream* to a file name (option on *hunter.actions.VarsPrinter* and *hunter.actions.CodePrinter*).
- Bumped up the filename alignment to 40 cols.
- If not merging then *self* is not kept as a previous tracer anymore. Closes #16.
- Fixed handling in VarsPrinter: properly print eval errors and don't try to show anything if there's an AttributeError. Closes #18.
- Added a `stdlib` boolean flag (for filtering purposes). Closes #15.
- Fixed broken frames that have "None" for filename or module (so they can still be treated as strings).
- Corrected output files in the `install_lib` command so that pip can uninstall the pth file. This only works when it's installed with pip (sadly, `setup.py install/develop` and `pip install -e` will still leave pth garbage on `pip uninstall hunter`).

11.34 0.5.1 (2015-04-15)

- Fixed *hunter.event.Event.globals* to actually be the dict of global vars (it was just the locals).

11.35 0.5.0 (2015-04-06)

- Fixed *hunter.And()* and *hunter.Or()* "single argument unwrapping".
- Implemented predicate compression. Example: `Or(Or(a, b), c)` is converted to `Or(a, b, c)`.
- Renamed *hunter.event.Event.source* to *hunter.event.Event.fullsource*.
- Added *hunter.event.Event.source* that doesn't do any fancy sourcecode tokenization.
- Fixed *hunter.event.Event.fullsource* return value for situations where the tokenizer would fail.
- Made the print function available in the `PYTHONHUNTER` env var payload.
- Added a __repr__ for *hunter.event.Event*.

11.36 0.4.0 (2015-03-29)

- Disabled colors for Jython. Contributed by Claudiu Popa in #12.
- Test suite fixes for Windows. Contributed by Claudiu Popa in #11.
- Added an introduction section in the docs.
- Implemented a prettier fallback for when no sources are available for that frame.
- Implemented fixups in cases where you use action classes as a predicates.

11.37 0.3.1 (2015-03-29)

- Forgot to merge some commits . . .

11.38 0.3.0 (2015-03-29)

- Added handling for internal repr failures.
- Fixed issues with displaying code that has non-ascii characters.
- Implemented better display for `call` frames so that when a function has decorators the function definition is shown (instead of just the first decorator). See: #8.

11.39 0.2.1 (2015-03-28)

- Added missing color entry for exception events.
- Added `hunter.event.Event.line` property. It returns the source code for the line being run.

11.40 0.2.0 (2015-03-27)

- Added color support (and `colorama` as dependency).
- Added support for expressions in *hunter.actions.VarsPrinter*.
- Breaking changes:
  - Renamed `F` to *hunter.Q()*. And *hunter.Q()* is now just a convenience wrapper for *hunter.predicates.Query*.
  - Renamed the `PYTHON_HUNTER` env variable to `PYTHONHUNTER`.
  - Changed *hunter.predicates.When* to take positional arguments.
  - Changed output to show 2 path components (still not configurable).
  - Changed *hunter.actions.VarsPrinter* to take positional arguments for the names.
- Improved error reporting for env variable activation (`PYTHONHUNTER`).
- Fixed env var activator (the `.pth` file) installation with `setup.py install` (the "egg installs") and `setup.py develop`/`pip install -e` (the "egg links").

11.41 0.1.0 (2015-03-22)

- First release on PyPI.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

S

T

V

W