Hunter

Release 2.0.1

Sep 09, 2017

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 Python API, terminal activation (see *Environment variable activation*). and supports tracing other processes (see *Tracing processes*).

- Free software: BSD license

Installation

```
pip install hunter
```

Documentation

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

Overview

The default action is to just print the code being executed. Example:

```
import hunter
hunter.trace(module='posixpath')

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

Would result in:

```
>>> os.path.join('a', 'b')
         /usr/lib/python3.5/posixpath.py:71    call      def join(a, *p):
         /usr/lib/python3.5/posixpath.py:76    line          sep = _get_sep(a)
         /usr/lib/python3.5/posixpath.py:39    call      def _get_sep(path):
         /usr/lib/python3.5/posixpath.py:40    line          if isinstance(path,␣
↪bytes):
         /usr/lib/python3.5/posixpath.py:43    line              return '/'
         /usr/lib/python3.5/posixpath.py:43    return            return '/'
                                               ...       return value: '/'
         /usr/lib/python3.5/posixpath.py:77    line          path = a
         /usr/lib/python3.5/posixpath.py:78    line          try:
         /usr/lib/python3.5/posixpath.py:79    line              if not p:
         /usr/lib/python3.5/posixpath.py:81    line              for b in p:
         /usr/lib/python3.5/posixpath.py:82    line                  if b.
↪startswith(sep):
         /usr/lib/python3.5/posixpath.py:84    line                  elif not path or␣
↪path.endswith(sep):
         /usr/lib/python3.5/posixpath.py:87    line                      path += sep␣
↪+ b
         /usr/lib/python3.5/posixpath.py:81    line              for b in p:
         /usr/lib/python3.5/posixpath.py:91    line          return path
         /usr/lib/python3.5/posixpath.py:91    return        return path
                                               ...       return value: 'a/b'
'a/b'
```

- or in a terminal:

Custom actions

The tracer allow custom actions like `CallPrinter` or `VarsPrinter`.

With `CallPrinter` (added in *hunter 1.2.0*, will be the default action in *2.0.1*):

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

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

Would result in:

```
>>> os.path.join('a', 'b')
         /usr/lib/python3.5/posixpath.py:71    call      => join(a='a')
         /usr/lib/python3.5/posixpath.py:76    line         sep = _get_sep(a)
         /usr/lib/python3.5/posixpath.py:39    call         => _get_sep(path='a')
```

```
        /usr/lib/python3.5/posixpath.py:40    line             if isinstance(path,␣
↪bytes):
        /usr/lib/python3.5/posixpath.py:43    line             return '/'
        /usr/lib/python3.5/posixpath.py:43    return         <= _get_sep: '/'
        /usr/lib/python3.5/posixpath.py:77    line           path = a
        /usr/lib/python3.5/posixpath.py:78    line           try:
        /usr/lib/python3.5/posixpath.py:79    line           if not p:
        /usr/lib/python3.5/posixpath.py:81    line           for b in p:
        /usr/lib/python3.5/posixpath.py:82    line           if b.startswith(sep):
        /usr/lib/python3.5/posixpath.py:84    line           elif not path or path.
↪endswith(sep):
        /usr/lib/python3.5/posixpath.py:87    line           path += sep + b
        /usr/lib/python3.5/posixpath.py:81    line           for b in p:
        /usr/lib/python3.5/posixpath.py:91    line           return path
        /usr/lib/python3.5/posixpath.py:91    return     <= join: 'a/b'
'a/b'
```

In a terminal it would look like:

With `VarsPrinter`:

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

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

Would result in:

```
>>> os.path.join('a', 'b')
        /usr/lib/python3.5/posixpath.py:71    call       def join(a, *p):
        /usr/lib/python3.5/posixpath.py:76    line           sep = _get_sep(a)
                                              vars       path => 'a'
        /usr/lib/python3.5/posixpath.py:39    call       def _get_sep(path):
                                              vars       path => 'a'
        /usr/lib/python3.5/posixpath.py:40    line           if isinstance(path,␣
↪bytes):
                                              vars       path => 'a'
        /usr/lib/python3.5/posixpath.py:43    line               return '/'
                                              vars       path => 'a'
        /usr/lib/python3.5/posixpath.py:43    return             return '/'
                                              ...        return value: '/'
        /usr/lib/python3.5/posixpath.py:77    line           path = a
                                              vars       path => 'a'
        /usr/lib/python3.5/posixpath.py:78    line           try:
                                              vars       path => 'a'
        /usr/lib/python3.5/posixpath.py:79    line               if not p:
                                              vars       path => 'a'
        /usr/lib/python3.5/posixpath.py:81    line               for b in p:
                                              vars       path => 'a'
        /usr/lib/python3.5/posixpath.py:82    line                   if b.
↪startswith(sep):
                                              vars       path => 'a'
        /usr/lib/python3.5/posixpath.py:84    line                   elif not path or␣
↪path.endswith(sep):
                                              vars       path => 'a'
```

```
         /usr/lib/python3.5/posixpath.py:87    line                      path += sep␣
↪+ b
                                               vars      path => 'a/b'
         /usr/lib/python3.5/posixpath.py:81    line              for b in p:
                                               vars      path => 'a/b'
         /usr/lib/python3.5/posixpath.py:91    line          return path
                                               vars      path => 'a/b'
         /usr/lib/python3.5/posixpath.py:91    return        return path
                                               ...       return value: 'a/b'
'a/b'
```

In a terminal it would look like:

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:

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 `pip install 'hunter[remote]'` and 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.

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=""`.

Filtering DSL

Hunter supports a flexible query DSL, see the introduction.

Development

To run the all tests run:

```
tox
```

FAQ

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.

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.

TODO: Expand this.

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.

CHAPTER 2

Installation

At the command line:

```
pip install hunter
```

CHAPTER 3

Introduction

Installation

To install hunter run:

```
pip install hunter
```

The `trace` function

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

- Keyword arguments like `module`, `function` or `action`. This is for convenience.
- Callbacks that take an `event` argument:
  - Builtin predicates like: *hunter.Query*, *hunter.When*, *hunter.And* or *hunter.Or*.
  - Actions like: *hunter.CodePrinter*, *hunter.Debugger* or *hunter.VarsPrinter*
  - Any function. Or a disgusting lambda.

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

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.And*.

Composing

All the builtin predicates (*hunter.Query*, *hunter.When*, *hunter.And* and *hunter.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'))`

Operators

New in version 1.0.0.

You can add `startswith`, `endswith`, `in`, `contains`, `regex` 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`.

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)`.

Activation

You can activate Hunter in two ways.

via code

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

via 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="".

CHAPTER 4

Remote tracing

Hunter supports tracing local processes, with two backends: manhole (`pip install 'hunter[remote]'`) and GDB.

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` ...

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

Cookbook

When in doubt, use Hunter.

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
```

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
```

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:

```
.. sourcecode:: python
```

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

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.

CHAPTER 6

Reference

Functions

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

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: `False`.
- **`action`** – Action to run if all the predicates return `True`. Default: `CodePrinter`.
- **`actions`** – Actions to run (in case you want more than 1).

`hunter.`**`stop`**`()`

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

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

Handles situations where *`hunter.Query`* objects (or other callables) are passed in as positional arguments. Conveniently converts that to an *`hunter.And`* predicate.

Predicates

class hunter.**Query**

A query class.

See `hunter.Event` for fields that can be filtered on.

__and__

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

__call__

Handles event. Returns True if all criteria matched.

__eq__

x.__eq__(y) <==> x==y

__ge__

x.__ge__(y) <==> x>=y

__gt__

x.__gt__(y) <==> x>y

__hash__

__init__

Args –

query: criteria to match on.

Accepted arguments: `arg`, `code`, `filename`, `frame`, `fullsource`, `function`, `globals`, `kind`, `lineno`, `locals`, `module`, `source`, `stdlib`, `tracer`.

__invert__

x.__invert__() <==> ~x

__le__

x.__le__(y) <==> x<=y

__lt__

x.__lt__(y) <==> x<y

__ne__

x.__ne__(y) <==> x!=y

__new__(*S*, ...) → a new object with type S, a subtype of T

__or__

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

__rand__

x.__rand__(y) <==> y&x

__repr__

__ror__

x.__ror__(y) <==> y|x

__str__

class hunter.**When**

Runs `actions` when `condition(event)` is `True`.

Actions take a single `event` argument.

__and__
x.__and__(y) <==> x&y

__call__
Handles the event.

__eq__
x.__eq__(y) <==> x==y

__ge__
x.__ge__(y) <==> x>=y

__gt__
x.__gt__(y) <==> x>y

__hash__

__init__
x.__init__(...) initializes x; see help(type(x)) for signature

__invert__
x.__invert__() <==> ~x

__le__
x.__le__(y) <==> x<=y

__lt__
x.__lt__(y) <==> x<y

__ne__
x.__ne__(y) <==> x!=y

__new__ (*S*, ...) → a new object with type S, a subtype of T

__or__
x.__or__(y) <==> x|y

__rand__
x.__rand__(y) <==> y&x

__repr__

__ror__
x.__ror__(y) <==> y|x

__str__

hunter.**And**(**predicates*, ***kwargs*)
And predicate. Returns `False` at the first sub-predicate that returns `False`.

hunter.**Or**(**predicates*, ***kwargs*)
Or predicate. Returns `True` at the first sub-predicate that returns `True`.

Actions

class hunter.**CallPrinter**(*stream=sys.stderr*, *filename_alignment=40*, *force_colors=False*, *repr_limit=512*)
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`.

New in version 1.2.0.

Note: This will be the default action in *hunter 2.0*.

__call__(*event*, *sep='/'*, *join=<function join>*)
Handle event and print filename, line number and source code. If event.kind is a *return* or *exception* also prints values.

class hunter.**CodePrinter**(*stream=sys.stderr*, *filename_alignment=40*, *force_colors=False*, *repr_limit=512*)
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`.

__call__(*event*, *sep='/'*, *join=<function join>*)
Handle event and print filename, line number and source code. If event.kind is a *return* or *exception* also prints values.

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

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

class hunter.**VarsPrinter**(*name*[, *name*[, *name*[, *...*]]], *globals=False*, *stream=sys.stderr*, *filename_alignment=40*, *force_colors=False*, *repr_limit=512*)
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.
- **globals** (*bool*) – Allow access to globals. Default: `False` (only looks at locals).
- **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`.

__call__(*event*)
Handle event and print the specified variables.

Objects

class `hunter.event.`**`Event`**(*frame*, *kind*, *arg*, *tracer*)

Event wrapper for `frame, kind, arg` (the arguments the settrace function gets). This objects is passed to your custom functions or predicates.

Provides few convenience properties.

> **Warning:** Users do not instantiate this directly.

`code`
A code object (not a string).

`filename`
A string with absolute path to file.

`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).

`function`
A string with function name.

`globals`
A dict with global variables.

`lineno`
An integer with line number in file.

`locals`
A dict with local variables.

`module`
A string with module name (eg – `"foo.bar"`).

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

Fast but sometimes incomplete.

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

`thread`
Current thread object.

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

`threadname`
Current thread name.

CHAPTER 7

Contributing

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

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.

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.

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 :)

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:your_name_here/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, doc builder and spell checker 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.

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`.

Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[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 8

Authors

- Ionel Cristian Mărieș - https://blog.ionelmc.ro

CHAPTER 9

Changelog

2.0.1 (2017-09-09)

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

2.0.0 (2017-09-02)

- Added the `Event.count` and `Event.calls` attributes.
- Added the `lt`/`lte`/`gt`/`gte` lookups.
- Added convenience aliases for `startswith` (`sw`), `endswith` (`ew`) 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 `CallPrinter`. You'll need to use `action=CodePrinter` if you want the old output.

1.4.1 (2016-09-24)

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

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.

1.3.0 (2016-04-14)

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

1.2.2 (2016-01-28)

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

1.2.1 (2016-01-27)

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

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 `VarsPrinter`'s initializer.
- Added a `CallPrinter` action.

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.

1.0.2 (2016-01-05)

- Fixed missing import in `setup.py`.

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`).

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.

0.6.0 (2015-10-10)

- Added a `clear_env_var` option on the tracer (disables tracing in subprocess).
- Added `force_colors` option on `VarsPrinter` and `CodePrinter`.
- Allowed setting the *stream* to a file name (option on `VarsPrinter` and `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`).

0.5.1 (2015-04-15)

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

0.5.0 (2015-04-06)

- Fixed `And` and `Or` "single argument unwrapping".
- Implemented predicate compression. Example: `Or(Or(a, b), c)` is converted to `Or(a, b, c)`.
- Renamed the `Event.source` to `Event.fullsource`.
- Added `Event.source` that doesn't do any fancy sourcecode tokenization.
- Fixed `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 `Event`.

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.

0.3.1 (2015-03-29)

- Forgot to merge some commits ...

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.

0.2.1 (2015-03-28)

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

0.2.0 (2015-03-27)

- Added color support (and `colorama` as dependency).
- Added support for expressions in `VarsPrinter`.
- Breaking changes:
  - Renamed `F` to `Q`. And `Q` is now just a convenience wrapper for `Query`.
  - Renamed the `PYTHON_HUNTER` env variable to `PYTHONHUNTER`.
  - Changed `When` to take positional arguments.
  - Changed output to show 2 path components (still not configurable).
  - Changed `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").

0.1.0 (2015-03-22)

- First release on PyPI.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

C

D

E

F

G

L

M

O

Q

S

T

V

W