Hunter

Release 1.1.0

January 21, 2016

Contents

1 Overview **1**
- 1.1 Installation . . . 1
- 1.2 Documentation . . . 1
- 1.3 Overview . . . 1
- 1.4 Development . . . 3
- 1.5 FAQ . . . 3

2 Installation **5**

3 Introduction **7**
- 3.1 Installation . . . 7
- 3.2 The `trace` function . . . 7
- 3.3 The `Q` function . . . 7
- 3.4 Composing . . . 8
- 3.5 Operators . . . 8
- 3.6 Activation . . . 8

4 Cookbook **11**
- 4.1 Packaging . . . 11
- 4.2 Typical . . . 11
- 4.3 Needle in the haystack . . . 11

5 Reference **13**
- 5.1 Functions . . . 13
- 5.2 Predicates . . . 13
- 5.3 Objects . . . 15
- 5.4 Other . . . 16

6 Contributing **19**
- 6.1 Bug reports . . . 19
- 6.2 Documentation improvements . . . 19
- 6.3 Feature requests and feedback . . . 19
- 6.4 Development . . . 19

7 Authors **21**

8 Changelog **23**
- 8.1 1.1.0 (2016-01-21) . . . 23
- 8.2 1.0.2 (2016-01-05) . . . 23

8.3 1.0.1 (2015-12-24) . . . 23
8.4 1.0.0 (2015-12-24) . . . 23
8.5 0.6.0 (2015-10-10) . . . 24
8.6 0.5.1 (2015-04-15) . . . 24
8.7 0.5.0 (2015-04-06) . . . 24
8.8 0.4.0 (2015-03-29) . . . 25
8.9 0.3.1 (2015-03-29) . . . 25
8.10 0.3.0 (2015-03-29) . . . 25
8.11 0.2.1 (2015-03-28) . . . 25
8.12 0.2.0 (2015-03-27) . . . 25
8.13 0.1.0 (2015-03-22) . . . 26

9 Indices and tables **27**

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 and a convenient terminal API (see *Environment variable activation*).

- Free software: BSD license

1.1 Installation

```
pip install hunter
```

1.2 Documentation

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

1.3 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:

```
python2.7/posixpath.py:60    call      def join(a, *p):
python2.7/posixpath.py:64    line          path = a
python2.7/posixpath.py:65    line          for b in p:
```

```
python2.7/posixpath.py:66    line              if b.startswith('/'):
python2.7/posixpath.py:68    line              elif path == '' or path.endswith('/'):
python2.7/posixpath.py:71    line                  path += '/' + b
python2.7/posixpath.py:65    line          for b in p:
python2.7/posixpath.py:72    line          return path
python2.7/posixpath.py:72    return        return path
                             ...       return value: 'a/b'
```

- or in a terminal:

You can have custom actions, like a variable printer - example:

```
import hunter
hunter.trace(hunter.Q(module='posixpath', action=hunter.VarsPrinter('path')))

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

Would result in:

```
python2.7/posixpath.py:60    call      def join(a, *p):
python2.7/posixpath.py:64    line          path = a
                             vars      path => 'a'
python2.7/posixpath.py:65    line          for b in p:
                             vars      path => 'a'
python2.7/posixpath.py:66    line              if b.startswith('/'):
                             vars      path => 'a'
python2.7/posixpath.py:68    line              elif path == '' or path.endswith('/'):
                             vars      path => 'a'
python2.7/posixpath.py:71    line                  path += '/' + b
                             vars      path => 'a/b'
python2.7/posixpath.py:65    line          for b in p:
                             vars      path => 'a/b'
python2.7/posixpath.py:72    line          return path
                             vars      path => 'a/b'
python2.7/posixpath.py:72    return        return path
                             ...       return value: 'a/b'
```

- or in a terminal:

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):

TODO: More examples.

1.3.1 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=""`.

1.3.2 Filtering DSL

Hunter supports a flexible query DSL, see the introduction.

1.4 Development

To run the all tests run:

```
tox
```

1.5 FAQ

1.5.1 Why not Smiley?

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

- Complexity. Smiley is simply over-engineered:
  - It's 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 might way 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.

1.5.2 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

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`. 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.

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.And*.

3.4 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'))`

3.5 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)`.

3.6 Activation

You can activate Hunter in two ways.

3.6.1 *via* code

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

3.6.2 *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

Cookbook

When in doubt, use Hunter.

4.1 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, VarsPrint
```

4.2 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
```

4.3 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:

```
PYTHONHUNTER='~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:

```
PYTHONHUNTER='source_contains=".build_dir"'
```

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

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

CHAPTER 5

Reference

- *Functions*
- *Predicates*
- *Objects*
- *Other*

5.1 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 any of the given predicates match.

Keyword Arguments

- **clear_env_var** – Disables tracing in subprocess. 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.

5.2 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__

Parameters **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

__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`.

5.3 Objects

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

__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*[, ...]]], *stream=sys.stderr*, *filename_alignment=40*, *globals=False*)

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`.
- **globals** (*bool*) – Allow access to globals. Default: `False` (only looks at locals).

__call__(*event*)

Handle event and print the specified variables.

5.4 Other

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.

CHAPTER 6

Contributing

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

6.1 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.

6.2 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.

6.3 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 :)

6.4 Development

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

1. Fork python-hunter on GitHub.
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.

6.4.1 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`.

6.4.2 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 7

Authors

- Ionel Cristian Mărie - http://blog.ionelmc.ro

CHAPTER 8

Changelog

8.1 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.

8.2 1.0.2 (2016-01-05)

- Fixed missing import in `setup.py`.

8.3 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`).

8.4 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.

8.5 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`).

8.6 0.5.1 (2015-04-15)

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

8.7 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`.

8.8 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.

8.9 0.3.1 (2015-03-29)

- Forgot to merge some commits ...

8.10 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.

8.11 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.

8.12 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").

8.13 0.1.0 (2015-03-22)

- First release on PyPI.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Symbols

__and__ (hunter.Query attribute), 13
__and__ (hunter.When attribute), 14
__call__ (hunter.Query attribute), 13
__call__ (hunter.When attribute), 14
__call__() (hunter.CodePrinter method), 15
__call__() (hunter.Debugger method), 15
__call__() (hunter.VarsPrinter method), 16
__eq__ (hunter.Query attribute), 14
__eq__ (hunter.When attribute), 14
__ge__ (hunter.Query attribute), 14
__ge__ (hunter.When attribute), 14
__gt__ (hunter.Query attribute), 14
__gt__ (hunter.When attribute), 14
__hash__ (hunter.Query attribute), 14
__hash__ (hunter.When attribute), 15
__init__ (hunter.Query attribute), 14
__init__ (hunter.When attribute), 15
__invert__ (hunter.Query attribute), 14
__le__ (hunter.Query attribute), 14
__le__ (hunter.When attribute), 15
__lt__ (hunter.Query attribute), 14
__lt__ (hunter.When attribute), 15
__ne__ (hunter.Query attribute), 14
__ne__ (hunter.When attribute), 15
__new__() (hunter.Query method), 14
__new__() (hunter.When method), 15
__or__ (hunter.Query attribute), 14
__or__ (hunter.When attribute), 15
__rand__ (hunter.Query attribute), 14
__rand__ (hunter.When attribute), 15
__repr__ (hunter.Query attribute), 14
__repr__ (hunter.When attribute), 15
__ror__ (hunter.Query attribute), 14
__ror__ (hunter.When attribute), 15
__str__ (hunter.Query attribute), 14
__str__ (hunter.When attribute), 15

A

And() (in module hunter), 15

C

code (hunter.event.Event attribute), 16
CodePrinter (class in hunter), 15

D

Debugger (class in hunter), 15

E

Event (class in hunter.event), 16

F

filename (hunter.event.Event attribute), 16
fullsource (hunter.event.Event attribute), 16
function (hunter.event.Event attribute), 16

G

globals (hunter.event.Event attribute), 16

L

lineno (hunter.event.Event attribute), 16
locals (hunter.event.Event attribute), 16

M

module (hunter.event.Event attribute), 16

O

Or() (in module hunter), 15

Q

Q() (in module hunter), 13
Query (class in hunter), 13

S

source (hunter.event.Event attribute), 16
stdlib (hunter.event.Event attribute), 16
stop() (in module hunter), 13

T

trace() (in module hunter), 13

V

VarsPrinter (class in hunter), 15

W

When (class in hunter), 14