pytracemalloc Documentation

Release latest

Victor Stinner

February 20, 2017

Contents

The tracemalloc module is a debug tool to trace memory blocks allocated by Python. It provides the following information:

- Traceback where an object was allocated
- Statistics on allocated memory blocks per filename and per line number: total size, number and average size of allocated memory blocks
- Compute the differences between two snapshots to detect memory leaks

To trace most memory blocks allocated by Python, the module should be started as early as possible by setting the `PYTHONTRACEMALLOC` environment variable to `1`. The `tracemalloc.start()` function can be called at runtime to start tracing Python memory allocations.

By default, a trace of an allocated memory block only stores the most recent frame (1 frame). To store 25 frames at startup: set the `PYTHONTRACEMALLOC` environment variable to `25`.

Websites:

- Project homepage (this documentation)
- Entry in the Python Cheeseshop (PyPI)
- Source code at Github
- Statistics on the project at Ohloh
- Qt graphical interface: tracemallocqt

The tracemalloc module has been integrated in Python 3.4: read tracemalloc module documentation.

CHAPTER 1

Table of Contents

Installation

Use Python 3.4 or newer

tracemalloc is now part of Python 3.4 standard library! Nothing to do, enjoy!

Note: Installing pytracemalloc on Python older than 3.4 is much more complex, it requires to recompile a patched version of Python. It is worth to try to run your application on Python 3.4 rather than trying to compile and install manually pytracemalloc on older versions of Python.

Linux packages

Ubuntu packages for pytracemalloc 1.2: pytracemalloc 1.0 PPA by Ionel Cristian Maries.

Manual installation

First, create the directory `/opt/tracemalloc`. Example:

```
sudo mkdir /opt/tracemalloc
sudo chown $USER: /opt/tracemalloc
```

Go into the */opt/tracemalloc* directory. Then follow these commands to compile a patched Python and install pytracemalloc:

```
wget http://www.python.org/ftp/python/2.7.8/Python-2.7.8.tgz
wget https://pypi.python.org/packages/source/p/pytracemalloc/pytracemalloc-1.2.tar.gz
tar -xf Python-2.7.8.tgz
tar -xf pytracemalloc-1.2.tar.gz
cd Python-2.7.8
patch -p1 < ../pytracemalloc-1.2/patches/2.7/pep445.patch
./configure --enable-unicode=ucs4 --prefix=/opt/tracemalloc/py27
make install
cd ../pytracemalloc-1.2
/opt/tracemalloc/py27/bin/python2.7 setup.py install
```

You have now a patched Python 2.7 installed in `/opt/tracemalloc/py27/bin/python2.7` with the `tracemalloc` module installed, congrats!

To use modules installed for the system Python, directories of `sys.path` should be copied from the system Python to the patched Python. Example of command to generate an environment variable to use system modules:

```
python -c 'import sys; print("PYTHONPATH=%s" % ":".join(filter(bool, sys.path)))'
```

Patch Python

To install pytracemalloc, you need a modified Python runtime:

- Download Python source code (tarball)
- Uncompress the tarball and enter the newly created directory (ex: Python-2.7.8)
- Apply the patch of your Python version, example:

```
patch -p1 < ~/pytracemalloc-1.0/patches/2.7/pep445.patch
```

- Compile and install Python:

```
./configure --enable-unicode=ucs4 --prefix=/opt/python && make && sudo make install
```

Note: `--enable-unicode=ucs4` uses the wide mode: store Unicode code points in 32-bit (4 bytes per character). It is the mode used by all Linux distributions. Your modified Python will have the same ABI and so you should be able to use extension modules of the system.

`--enable-unicode=ucs4` is no more needed with Python 3.3 which always uses compact strings: see the PEP 393.

Note: Currently, only patches for Python 2.7 and 3.3 are provided. If you need patches for other Python versions, please ask. The code should work on Python 2.5-3.3.

Compile and install pytracemalloc

Dependencies:

- Python 2.5 - 3.3

Download pytracemalloc from the Python Cheeseshop (PyPI).

Install pytracemalloc:

```
/opt/python/bin/python setup.py install
```

Examples

Display the top 10

Display the 10 files allocating the most memory:

```
import tracemalloc

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output of the Python test suite:

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB, count=2315, average=108 B
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB, count=378, average=416 B
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347, average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

We can see that Python loaded `4.8 MiB` data (bytecode and constants) from modules and that the `collections` module allocated `244 KiB` to build `namedtuple` types.

See *`Snapshot.statistics()`* for more options.

Compute differences

Take two snapshots and display the differences:

```
import tracemalloc
tracemalloc.start()
# ... start your application ...

snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output before/after running some tests of the Python test suite:

```
[ Top 10 differences ]
<frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB), count=71332 (+39369), average=117 B
/usr/lib/python3.4/linecache.py:127: size=940 KiB (+940 KiB), count=8106 (+8106), average=119 B
/usr/lib/python3.4/unittest/case.py:571: size=298 KiB (+298 KiB), count=589 (+589), average=519 B
<frozen importlib._bootstrap>:284: size=1005 KiB (+166 KiB), count=7423 (+1526), average=139 B
/usr/lib/python3.4/mimetypes.py:217: size=112 KiB (+112 KiB), count=1334 (+1334), average=86 B
```

```
/usr/lib/python3.4/http/server.py:848: size=96.0 KiB (+96.0 KiB), count=1 (+1), average=96.0 KiB
/usr/lib/python3.4/inspect.py:1465: size=83.5 KiB (+83.5 KiB), count=109 (+109), average=784 B
/usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7 KiB), count=143 (+143), average=557 B
/usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8 KiB), count=969 (+969), average=76 B
/usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB), count=126 (+126), average=546 B
```

We can see that Python has loaded `8.2 MiB` of module data (bytecode and constants), and that this is `4.4 MiB` more than had been loaded before the tests, when the previous snapshot was taken. Similarly, the `linecache` module has cached `940 KiB` of Python source code to format tracebacks, all of it since the previous snapshot.

If the system has little free memory, snapshots can be written on disk using the *Snapshot.dump()* method to analyze the snapshot offline. Then use the *Snapshot.load()* method reload the snapshot.

Get the traceback of a memory block

Code to display the traceback of the biggest memory block:

```
import tracemalloc

# Store 25 frames
tracemalloc.start(25)

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# pick the biggest memory block
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
for line in stat.traceback.format():
    print(line)
```

Example of output of the Python test suite (traceback limited to 25 frames):

```
903 memory blocks: 870.1 KiB
  File "<frozen importlib._bootstrap>", line 716
  File "<frozen importlib._bootstrap>", line 1036
  File "<frozen importlib._bootstrap>", line 934
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/doctest.py", line 101
    import pdb
  File "<frozen importlib._bootstrap>", line 284
  File "<frozen importlib._bootstrap>", line 938
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/test/support/__init__.py", line 1728
    import doctest
  File "/usr/lib/python3.4/test/test_pickletools.py", line 21
    support.run_doctest(pickletools)
  File "/usr/lib/python3.4/test/regrtest.py", line 1276
    test_runner()
  File "/usr/lib/python3.4/test/regrtest.py", line 976
```

```
    display_failure=not verbose)
  File "/usr/lib/python3.4/test/regrtest.py", line 761
    match_tests=ns.match_tests)
  File "/usr/lib/python3.4/test/regrtest.py", line 1563
    main()
  File "/usr/lib/python3.4/test/__main__.py", line 3
    regrtest.main_in_temp_cwd()
  File "/usr/lib/python3.4/runpy.py", line 73
    exec(code, run_globals)
  File "/usr/lib/python3.4/runpy.py", line 160
    "__main__", fname, loader, pkg_name)
```

We can see that the most memory was allocated in the `importlib` module to load data (bytecode and constants) from modules: `870 KiB`. The traceback is where the `importlib` loaded data most recently: on the `import pdb` line of the `doctest` module. The traceback may change if a new module is loaded.

Pretty top

Code to display the 10 lines allocating the most memory with a pretty output, ignoring `<frozen importlib._bootstrap>` and `<unknown>` files:

```
import os
import tracemalloc

def display_top(snapshot, group_by='lineno', limit=10):
    snapshot = snapshot.filter_traces((
        tracemalloc.Filter(False, "<frozen importlib._bootstrap>"),
        tracemalloc.Filter(False, "<unknown>"),
    ))
    top_stats = snapshot.statistics(group_by)

    print("Top %s lines" % limit)
    for index, stat in enumerate(top_stats[:limit], 1):
        frame = stat.traceback[0]
        # replace "/path/to/module/file.py" with "module/file.py"
        filename = os.sep.join(frame.filename.split(os.sep)[-2:])
        print("#%s: %s:%s: %.1f KiB"
              % (index, filename, frame.lineno,
                 stat.size / 1024))

    other = top_stats[limit:]
    if other:
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size / 1024))
    total = sum(stat.size for stat in top_stats)
    print("Total allocated size: %.1f KiB" % (total / 1024))

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)
```

Example of output of the Python test suite:

```
2013-11-08 14:16:58.149320: Top 10 lines
#1: collections/__init__.py:368: 291.9 KiB
#2: Lib/doctest.py:1291: 200.2 KiB
#3: unittest/case.py:571: 160.3 KiB
#4: Lib/abc.py:133: 99.8 KiB
#5: urllib/parse.py:476: 71.8 KiB
#6: <string>:5: 62.7 KiB
#7: Lib/base64.py:140: 59.8 KiB
#8: Lib/_weakrefset.py:37: 51.8 KiB
#9: collections/__init__.py:362: 50.6 KiB
#10: test/test_site.py:56: 48.0 KiB
7496 other: 4161.9 KiB
Total allocated size: 5258.8 KiB
```

See *Snapshot.statistics()* for more options.

Thread to write snapshots into files every minutes

Create a daemon thread writing snapshots every minutes into `/tmp/tracemalloc-PPP-CCCC.pickle` where `PPP` is the identifier of the process and `CCCC` is a counter:

```
import pickle, gc, os, signal, threading, time, tracemalloc

class TakeSnapshot(threading.Thread):
    daemon = True

    def run(self):
        if hasattr(signal, 'pthread_sigmask'):
            # Available on UNIX with Python 3.3+
            signal.pthread_sigmask(signal.SIG_BLOCK, range(1, signal.NSIG))
        counter = 1
        while True:
            time.sleep(60)
            filename = ("/tmp/tracemalloc-%d-%04d.pickle"
                        % (os.getpid(), counter))
            print("Write snapshot into %s..." % filename)
            gc.collect()
            snapshot = tracemalloc.take_snapshot()
            with open(filename, "wb") as fp:
                # Pickle version 2 can be read by Python 2 and Python 3
                pickle.dump(snapshot, fp, 2)
            snapshot = None
            print("Snapshot written into %s" % filename)
            counter += 1

# save 25 frames
tracemalloc.start(25)
TakeSnapshot().start()
```

API

The version of the module is `tracemalloc.__version__` (string), example: `"1.2"`.

Functions

clear_traces()
Clear traces of memory blocks allocated by Python.

See also *stop()*.

get_object_traceback(*obj*)
Get the traceback where the Python object *obj* was allocated. Return a *Traceback* instance, or `None` if the *tracemalloc* module is not tracing memory allocations or did not trace the allocation of the object.

See also `gc.get_referrers()` and `sys.getsizeof()` functions.

get_traceback_limit()
Get the maximum number of frames stored in the traceback of a trace.

The *tracemalloc* module must be tracing memory allocations to get the limit, otherwise an exception is raised.

The limit is set by the *start()* function.

get_traced_memory()
Get the current size and peak size of memory blocks traced by the *tracemalloc* module as a tuple: `(current: int, peak: int)`.

get_tracemalloc_memory()
Get the memory usage in bytes of the *tracemalloc* module used to store traces of memory blocks. Return an `int`.

is_tracing()
`True` if the *tracemalloc* module is tracing Python memory allocations, `False` otherwise.

See also *start()* and *stop()* functions.

start(*nframe: int=1*)
Start tracing Python memory allocations: install hooks on Python memory allocators. Collected tracebacks of traces will be limited to *nframe* frames. By default, a trace of a memory block only stores the most recent frame: the limit is `1`. *nframe* must be greater or equal to `1`.

Storing more than `1` frame is only useful to compute statistics grouped by `'traceback'` or to compute cumulative statistics: see the *Snapshot.compare_to()* and *Snapshot.statistics()* methods.

Storing more frames increases the memory and CPU overhead of the *tracemalloc* module. Use the *get_tracemalloc_memory()* function to measure how much memory is used by the *tracemalloc* module.

See also *stop()*, *is_tracing()* and *get_traceback_limit()* functions.

stop()
Stop tracing Python memory allocations: uninstall hooks on Python memory allocators. Also clears all previously collected traces of memory blocks allocated by Python.

Call *take_snapshot()* function to take a snapshot of traces before clearing them.

See also *start()*, *is_tracing()* and *clear_traces()* functions.

take_snapshot()
Take a snapshot of traces of memory blocks allocated by Python. Return a new *Snapshot* instance.

The snapshot does not include memory blocks allocated before the *tracemalloc* module started to trace memory allocations.

Tracebacks of traces are limited to *get_traceback_limit()* frames. Use the *nframe* parameter of the *start()* function to store more frames.

The *tracemalloc* module must be tracing memory allocations to take a snapshot, see the the *start()* function.

See also the *get_object_traceback()* function.

Filter

class Filter(*inclusive: bool*, *filename_pattern: str*, *lineno: int=None*, *all_frames: bool=False*)
Filter on traces of memory blocks.

See the `fnmatch.fnmatch()` function for the syntax of *filename_pattern*. The `'.pyc'` and `'.pyo'` file extensions are replaced with `'.py'`.

Examples:

- `Filter(True, subprocess.__file__)` only includes traces of the `subprocess` module
- `Filter(False, tracemalloc.__file__)` excludes traces of the *tracemalloc* module
- `Filter(False, "<unknown>")` excludes empty tracebacks

inclusive
If *inclusive* is `True` (include), only trace memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

If *inclusive* is `False` (exclude), ignore memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

lineno
Line number (`int`) of the filter. If *lineno* is `None`, the filter matches any line number.

filename_pattern
Filename pattern of the filter (`str`).

all_frames
If *all_frames* is `True`, all frames of the traceback are checked. If *all_frames* is `False`, only the most recent frame is checked.

This attribute has no effect if the traceback limit is `1`. See the *get_traceback_limit()* function and *Snapshot.traceback_limit* attribute.

Frame

class Frame
Frame of a traceback.

The *Traceback* class is a sequence of *Frame* instances.

filename
Filename (`str`).

lineno
Line number (`int`).

Snapshot

class Snapshot
Snapshot of traces of memory blocks allocated by Python.

The *take_snapshot()* function creates a snapshot instance.

compare_to(*old_snapshot: Snapshot*, *group_by: str*, *cumulative: bool=False*)
Compute the differences with an old snapshot. Get statistics as a sorted list of *StatisticDiff* instances grouped by *group_by*.

See the *statistics()* method for *group_by* and *cumulative* parameters.

The result is sorted from the biggest to the smallest by: absolute value of *StatisticDiff.size_diff*, *StatisticDiff.size*, absolute value of *StatisticDiff.count_diff*, *Statistic.count* and then by *StatisticDiff.traceback*.

dump(*filename*)
Write the snapshot into a file.

Use *load()* to reload the snapshot.

filter_traces(*filters*)
Create a new *Snapshot* instance with a filtered *traces* sequence, *filters* is a list of *Filter* instances. If *filters* is an empty list, return a new *Snapshot* instance with a copy of the traces.

All inclusive filters are applied at once, a trace is ignored if no inclusive filters match it. A trace is ignored if at least one exclusive filter matchs it.

classmethod load(*filename*)
Load a snapshot from a file.

See also *dump()*.

statistics(*group_by: str*, *cumulative: bool=False*)
Get statistics as a sorted list of *Statistic* instances grouped by *group_by*:

group_by	description
`'filename'`	filename
`'lineno'`	filename and line number
`'traceback'`	traceback

If *cumulative* is `True`, cumulate size and count of memory blocks of all frames of the traceback of a trace, not only the most recent frame. The cumulative mode can only be used with *group_by* equals to `'filename'` and `'lineno'`.

The result is sorted from the biggest to the smallest by: *Statistic.size*, *Statistic.count* and then by *Statistic.traceback*.

traceback_limit
Maximum number of frames stored in the traceback of *traces*: result of the *get_traceback_limit()* when the snapshot was taken.

traces
Traces of all memory blocks allocated by Python: sequence of *Trace* instances.

The sequence has an undefined order. Use the *Snapshot.statistics()* method to get a sorted list of statistics.

Statistic

class Statistic
Statistic on memory allocations.

Snapshot.statistics() returns a list of *Statistic* instances.

See also the *StatisticDiff* class.

count
Number of memory blocks (int).

size
Total size of memory blocks in bytes (int).

traceback
Traceback where the memory block was allocated, *Traceback* instance.

StatisticDiff

class StatisticDiff
Statistic difference on memory allocations between an old and a new *Snapshot* instance.

Snapshot.compare_to() returns a list of *StatisticDiff* instances. See also the *Statistic* class.

count
Number of memory blocks in the new snapshot (int): 0 if the memory blocks have been released in the new snapshot.

count_diff
Difference of number of memory blocks between the old and the new snapshots (int): 0 if the memory blocks have been allocated in the new snapshot.

size
Total size of memory blocks in bytes in the new snapshot (int): 0 if the memory blocks have been released in the new snapshot.

size_diff
Difference of total size of memory blocks in bytes between the old and the new snapshots (int): 0 if the memory blocks have been allocated in the new snapshot.

traceback
Traceback where the memory blocks were allocated, *Traceback* instance.

Trace

class Trace
Trace of a memory block.

The *Snapshot.traces* attribute is a sequence of *Trace* instances.

size
Size of the memory block in bytes (int).

traceback
Traceback where the memory block was allocated, *Traceback* instance.

Traceback

class Traceback
Sequence of *Frame* instances sorted from the most recent frame to the oldest frame.

A traceback contains at least 1 frame. If the tracemalloc module failed to get a frame, the filename "<unknown>" at line number 0 is used.

When a snapshot is taken, tracebacks of traces are limited to *get_traceback_limit()* frames. See the *take_snapshot()* function.

The *Trace.traceback* attribute is an instance of *Traceback* instance.

Differences between pytracemalloc (PyPI) and tracemalloc (stdlib)

The tracemalloc module is part of the Python standard library since Python 3.4: read tracemalloc module documentation.

There are differences between the third party pytracemalloc module (downloaded from PyPI) and the tracemalloc which is part of the Python standard library:

- stdlib tracemalloc supports a `PYTHONTRACEMALLOC` environment variable to start tracing at Python startup.
- stdlib tracemalloc supports a `-X tracemalloc=NFRAMES` command line option to start tracing at Python startup.

tracemallocqt: GUI to analyze snapshots

tracemallocqt is graphical interface to analyze *tracemalloc* snapshots. It uses the Qt toolkit.

- tracemallocqt project at Bitbucket

Usage

Analyze a single snapshot:

```
./tracemallocqt.py snapshot.pickle
```

Compare two snapshots:

```
./tracemallocqt.py snapshot1.pickle snapshot2.pickle
```

You can pass more snapshots and then use the checkbox to select which snapshots are compared. The snpashots are sorted by the modification time of the files.

Installation

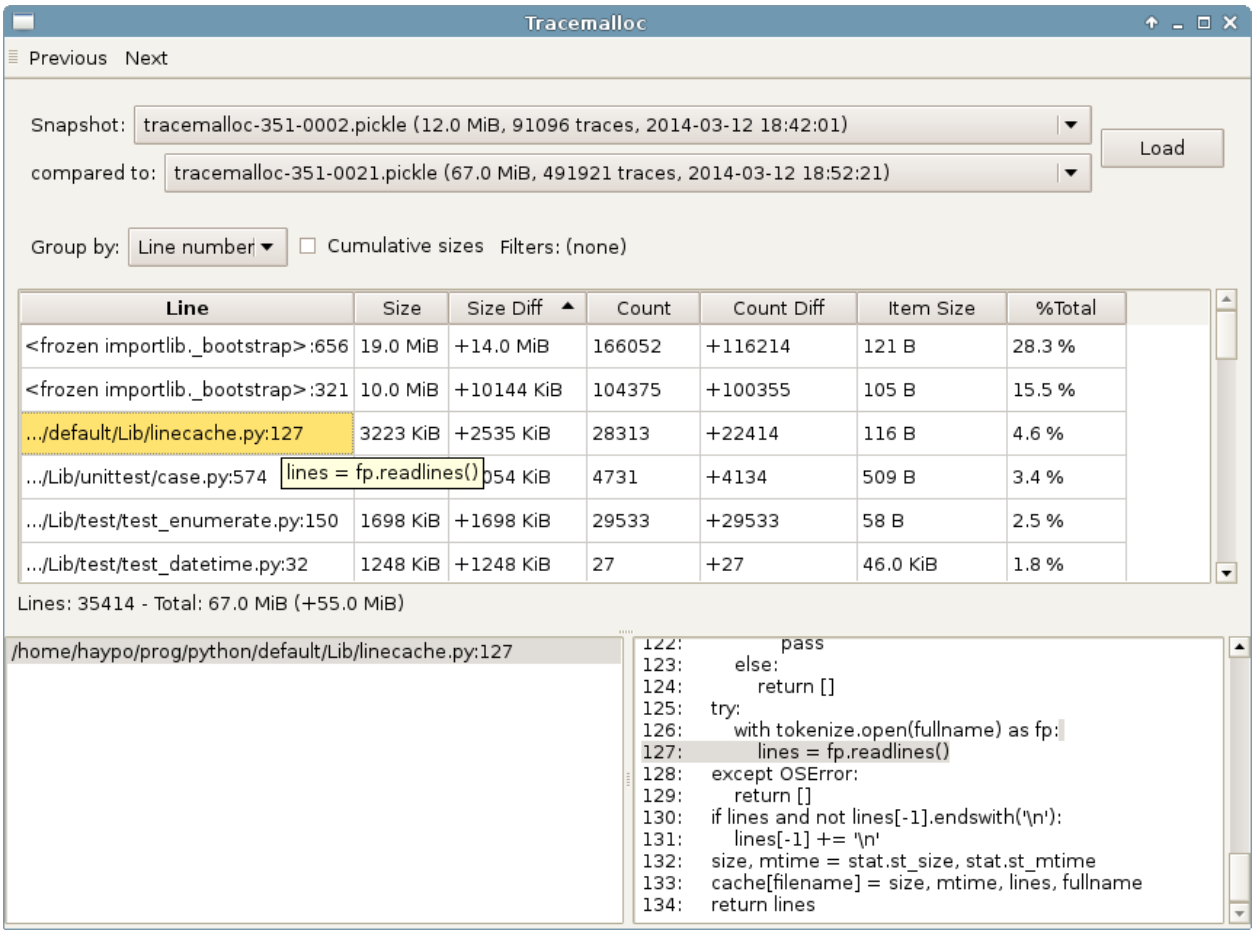
There is no release yet, you have to clone the Mercurial repository:

```
hg clone https://bitbucket.org/haypo/tracemallocqt
```

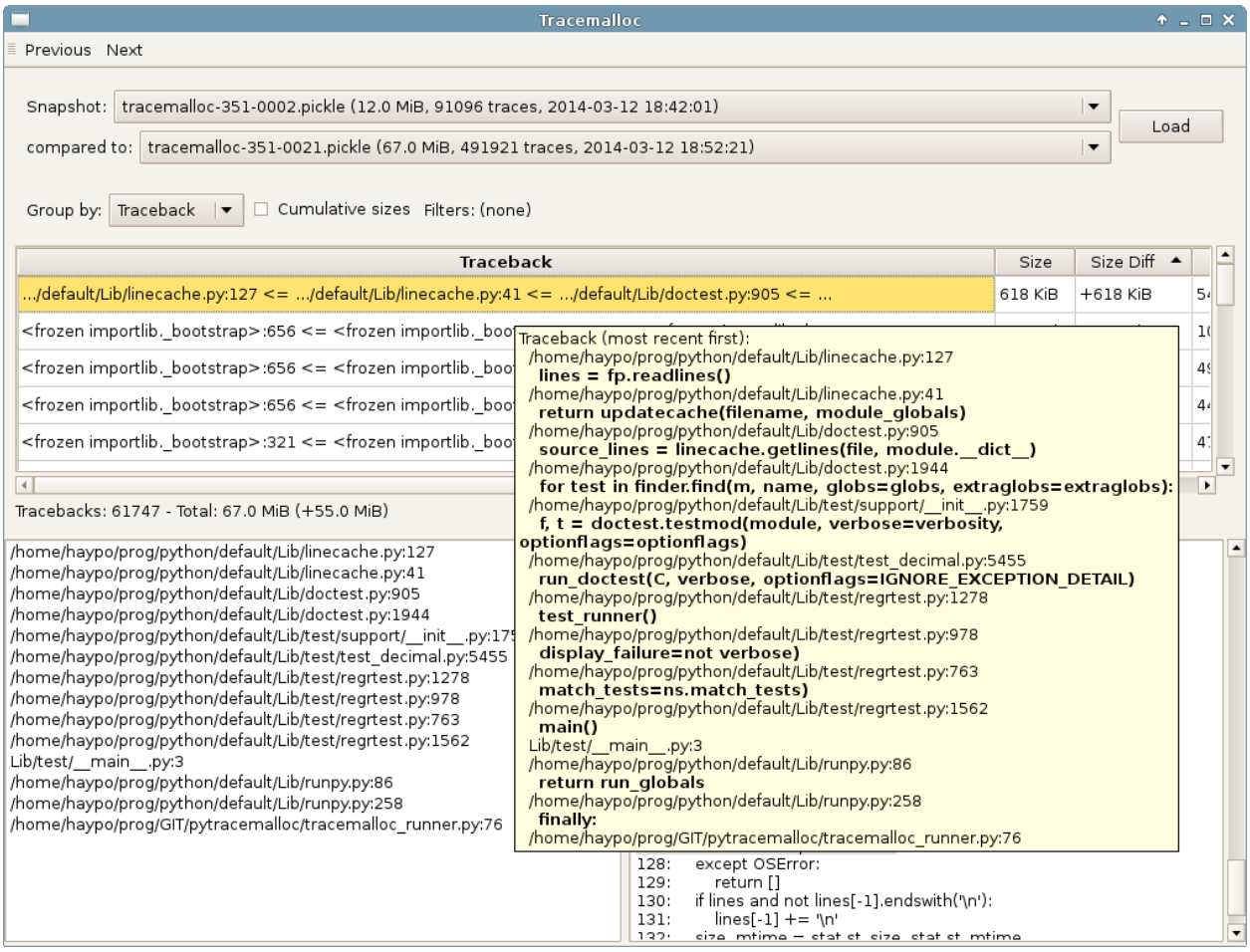
tracemallocqt works on Python 2 and 3 and requires PyQt4 or PySide.

Screenshots

Traces grouped by line number



Traces grouped by traceback



Changelog

Version 1.2 (2014-10-15)

- filter_traces() now raises a TypeError if filters is not an iterable
- Update Python 2.7 patch to try to keep the ABI unchanged, especially for Python compiled in debug mode (`./configure --with-pydebug`)
- Support Python 2.6
- Enhance the documentation (website)

Version 1.0 (2014-03-05)

- Python issue #20616: Add a format() method to tracemalloc.Traceback.
- Python issue #20354: Fix alignment issue in the tracemalloc module on 64-bit platforms. Bug seen on 64-bit Linux when using "make profile-opt".
- Fix slicing traces and fix slicing a traceback.

Version 1.0beta1 (2013-12-14)

- A trace of a memory block can now contain more than 1 frame, a whole traceback instead of just the most recent frame
- The malloc hook API has been proposed as the PEP 445. The PEP has been accepted and implemented in Python 3.4.
- The tracemalloc module has been proposed as the PEP 454. After many reviews, the PEP has been accepted and the code has been merged into Python 3.4.
- The code has been almost fully rewritten from scratch between the version 0.9.1 and 1.0. The tracemalloc has now a completly different API:
  - DisplayTop, TakeSnapshot and DisplayGarbage classes have been removed
  - Rename enable/disable to start/stop
  - start() now takes an optional nframe parameter which is the maximum number of frames stored in a trace of a memory block
  - Raw traces are accesible in Snapshot.traces
  - The get_process_memory() has been removed, but new functions are added like get_traced_memory()
- The glib hashtable has been replaced by a builtin hashtable based on the libcfu library. The glib dependency has been removed so it should be easier to install the module (ex: on Windows).

Version 0.9.1 (2013-06-01)

- Add `PYTRACEMALLOC` environment variable to trace memory allocation as early as possible at Python startup
- Disable the timer while calling its callback to not call the callback while it is running
- Fix pythonXXX_track_free_list.patch patches for zombie frames
- Use also MiB, GiB and TiB units to format a size, not only B and KiB

Version 0.9 (2013-05-31)

- Tracking free lists is now the recommended method to patch Python
- Fix code tracking Python free lists and python2.7_track_free_list.patch
- Add patches tracking free lists for Python 2.5.2 and 3.4.

Version 0.8.1 (2013-03-23)

- Fix python2.7.patch and python3.4.patch when Python is not compiled in debug mode (without –with-pydebug)
- Fix `DisplayTop`: display “0 B” instead of an empty string if the size is zero (ex: trace in user data)
- setup.py automatically detects which patch was applied on Python

Version 0.8 (2013-03-19)

- The top uses colors and displays also the memory usage of the process
- Add `DisplayGarbage` class
- Add `get_process_memory()` function
- Support collecting arbitrary user data using a callback: `Snapshot.create()`, `DisplayTop` and `TakeSnapshot` have has an optional user_data_callback parameter/attribute
- Display the name of the previous snapshot when comparing two snapshots
- Command line (`-m tracemalloc`):
  - Add `--color` and `--no-color` options
  - `--include` and `--exclude` command line options can now be specified multiple times
- Automatically disable tracemalloc at exit
- Remove `get_source()` and `get_stats()` functions: they are now private

Version 0.7 (2013-03-04)

- First public version

CHAPTER 2

Status of the module

pytracemalloc 1.0 contains patches for Python 2.7 and 3.3. The version 1.0 has been tested on Linux with Python 2.7 and 3.3: unit tests passed.

CHAPTER 3

Similar Projects

Python projects:

- Meliae: Python Memory Usage Analyzer
- Guppy-PE: umbrella package combining Heapy and GSL
- PySizer: developed for Python 2.4
- memory_profiler
- pympler
- memprof: based on sys.getsizeof() and sys.settrace()
- Dozer: WSGI Middleware version of the CherryPy memory leak debugger
- objgraph
- caulk

Perl projects:

- Devel::MAT by Paul Evans
- Devel::Size by Dan Sugalski
- Devel::SizeMe by Dan Sugalski

t