Pympler Documentation

Release 0.9

Jean Brouwers, Ludwig Haehne, Robert Schuppenies

October 14, 2020

Contents

Pympler is a development tool to measure, monitor and analyze the memory behavior of Python objects in a running Python application.

By pympling a Python application, detailed insight in the size and the lifetime of Python objects can be obtained. Undesirable or unexpected runtime behavior like memory bloat and other "pymples" can easily be identified.

Pympler integrates three previously separate modules into a single, comprehensive profiling tool. The *asizeof* module provides basic size information for one or several Python objects, module *muppy* is used for on-line monitoring of a Python application and module *Class Tracker* provides off-line analysis of the lifetime of selected Python objects.

A web profiling frontend exposes process statistics, garbage visualisation and class tracker statistics.

CHAPTER 1

Requirements

Pympler is written entirely in Python, with no dependencies to external libraries. It integrates Bottle and Flot. Pympler has been tested with Python 2.7, 3.5, 3.6, 3.7, 3.8 and 3.9.

Pympler is platform independent and has been tested on various Linux distributions (32bit and 64bit), Windows 7 and MacOS X.

CHAPTER 2

Download

If you have *pip* installed, the easiest way to get Pympler is:

```
pip install pympler
```

Alternately, download Pympler releases from the Python Package Index or check out the latest development revision with git. Please see the README file for installation instructions.

CHAPTER 3

Target Audience

Every Python developer interested in analyzing the memory consumption of their Python program should find a suitable, readily usable facility in Pympler.

CHAPTER 4

Usage Examples

`pympler.asizeof` can be used to investigate how much memory certain Python objects consume. In contrast to `sys.getsizeof`, `asizeof` sizes objects recursively. You can use one of the *asizeof* functions to get the size of these objects and all associated referents:

```
>>> from pympler import asizeof
>>> obj = [1, 2, (3, 4), 'text']
>>> asizeof.asizeof(obj)
176
>>> print(asizeof.asized(obj, detail=1).format())
[1, 2, (3, 4), 'text'] size=176 flat=48
    (3, 4) size=64 flat=32
    'text' size=32 flat=32
    1 size=16 flat=16
    2 size=16 flat=16
```

Memory leaks can be detected by using *muppy*. While the garbage collector debug output can report circular references this does not easily reveal where the leaks come from. Muppy can identify if objects are leaked out of a scope between two reference points:

```
>>> from pympler import tracker
>>> tr = tracker.SummaryTracker()
>>> function_without_side_effects()
>>> tr.print_diff()
  types |   # objects |   total size
======= | =========== | ============
   dict |           1 |    280     B
   list |           1 |    192     B
```

Tracking the lifetime of objects of certain classes can be achieved with the *Class Tracker*. This gives insight into instantiation patterns and helps to understand how specific objects contribute to the memory footprint over time:

```
>>> from pympler import classtracker
>>> tr = classtracker.ClassTracker()
>>> tr.track_class(Document)
```

(continues on next page)

(continued from previous page)

```
>>> tr.create_snapshot()
>>> create_documents()
>>> tr.create_snapshot()
>>> tr.stats.print_summary()
              active      1.42 MB      average   pct
   Document      1000    195.38 KB    200      B   13%
```

CHAPTER 5

History

Pympler was founded in August 2008 by Jean Brouwers, Ludwig Haehne, and Robert Schuppenies with the goal of providing a complete and stand-alone memory profiling solution for Python.

CHAPTER 6

Quick Links

Download pympler: https://pypi.org/projects/Pympler

File a bug report: https://github.com/pympler/pympler/issues

Check out repository: https://github.com/pympler/pympler

CHAPTER 7

Table of Content

Also available are

- genindex
- modindex
- search

7.1 Sitemap

Below you can find a complete overview of all pages of this documentation.

Pympler is a development tool to measure, monitor and analyze the memory behavior of Python objects in a running Python application.

By pympling a Python application, detailed insight in the size and the lifetime of Python objects can be obtained. Undesirable or unexpected runtime behavior like memory bloat and other "pymples" can easily be identified.

Pympler integrates three previously separate modules into a single, comprehensive profiling tool. The *asizeof* module provides basic size information for one or several Python objects, module *muppy* is used for on-line monitoring of a Python application and module *Class Tracker* provides off-line analysis of the lifetime of selected Python objects.

A web profiling frontend exposes process statistics, garbage visualisation and class tracker statistics.

7.1.1 Requirements

Pympler is written entirely in Python, with no dependencies to external libraries. It integrates Bottle and Flot. Pympler has been tested with Python 2.7, 3.5, 3.6, 3.7, 3.8 and 3.9.

Pympler is platform independent and has been tested on various Linux distributions (32bit and 64bit), Windows 7 and MacOS X.

7.1.2 Download

If you have *pip* installed, the easiest way to get Pympler is:

```
pip install pympler
```

Alternately, download Pympler releases from the Python Package Index or check out the latest development revision with git. Please see the README file for installation instructions.

7.1.3 Target Audience

Every Python developer interested in analyzing the memory consumption of their Python program should find a suitable, readily usable facility in Pympler.

7.1.4 Usage Examples

`pympler.asizeof` can be used to investigate how much memory certain Python objects consume. In contrast to `sys.getsizeof`, `asizeof` sizes objects recursively. You can use one of the *asizeof* functions to get the size of these objects and all associated referents:

```
>>> from pympler import asizeof
>>> obj = [1, 2, (3, 4), 'text']
>>> asizeof.asizeof(obj)
176
>>> print(asizeof.asized(obj, detail=1).format())
[1, 2, (3, 4), 'text'] size=176 flat=48
    (3, 4) size=64 flat=32
    'text' size=32 flat=32
    1 size=16 flat=16
    2 size=16 flat=16
```

Memory leaks can be detected by using *muppy*. While the garbage collector debug output can report circular references this does not easily reveal where the leaks come from. Muppy can identify if objects are leaked out of a scope between two reference points:

```
>>> from pympler import tracker
>>> tr = tracker.SummaryTracker()
>>> function_without_side_effects()
```

(continues on next page)

(continued from previous page)

```
>>> tr.print_diff()
  types |   # objects |   total size
======= | =========== | ============
   dict |           1 |    280     B
   list |           1 |    192     B
```

Tracking the lifetime of objects of certain classes can be achieved with the *Class Tracker*. This gives insight into instantiation patterns and helps to understand how specific objects contribute to the memory footprint over time:

```
>>> from pympler import classtracker
>>> tr = classtracker.ClassTracker()
>>> tr.track_class(Document)
>>> tr.create_snapshot()
>>> create_documents()
>>> tr.create_snapshot()
>>> tr.stats.print_summary()
              active      1.42 MB      average   pct
   Document     1000    195.38 KB    200      B   13%
```

7.1.5 History

Pympler was founded in August 2008 by Jean Brouwers, Ludwig Haehne, and Robert Schuppenies with the goal of providing a complete and stand-alone memory profiling solution for Python.

7.1.6 Sizing individual objects

Introduction

This module exposes 9 functions and 2 classes to obtain lengths and sizes of Python objects (for Python 2.6 or later).

Earlier versions of this module supported Python versions down to Python 2.2. If you are using Python 2.5 or older, please consider downgrading Pympler to version 0.3.x.

Public Functions[1]

Function **asizeof** calculates the combined (approximate) size in bytes of one or several Python objects.

Function **asizesof** returns a tuple containing the (approximate) size in bytes for each given Python object separately.

Function **asized** returns for each object an instance of class **Asized** containing all the size information of the object and a tuple with the referents[2].

Functions **basicsize** and **itemsize** return the *basic*- respectively *itemsize* of the given object, both in bytes. For objects as `array.array`, `numpy.array`, `numpy.matrix`, etc. where the item size varies depending on the instance-specific data type, function **itemsize** returns that item size.

Function **flatsize** returns the *flat size* of a Python object in bytes defined as the *basic size* plus the *item size* times the *length* of the given object.

[1] The functions and classes in this module are not thread-safe.

[2] The *referents* of an object are the objects referenced *by* that object. For example, the *referents* of a `list` are the objects held in the `list`, the *referents* of a `dict` are the key and value objects in the `dict`, etc.

Function **leng** returns the *length* of an object, like standard function `len` but extended for several types. E.g. the **leng** of a multi-precision int (or long) is the number of `digits`[4]. The length of most *mutable* sequence objects includes an estimate of the over-allocation and therefore, the **leng** value may differ from the standard `len` result. For objects like `array.array`, `numpy.array`, `numpy.matrix`, etc. function **leng** returns the proper number of items.

Function **refs** returns (a generator for) the referents[2] of the given object.

Certain classes are known to be sub-classes of or to behave as `dict` objects. Function **adict** can be used to register other class objects to be treated like `dict`.

Public Classes[1]

Class **Asizer** may be used to accumulate the results of several **asizeof** or **asizesof** calls. After creating an **Asizer** instance, use methods **asizeof** and **asizesof** as needed to size any number of additional objects.

Call methods **exclude_refs** and/or **exclude_types** to exclude references to respectively instances or types of certain objects.

Use one of the **print_...** methods to report the statistics.

An instance of class **Asized** is returned for each object sized by the **asized** function or method.

Duplicate Objects

Any duplicate, given objects are sized only once and the size is included in the accumulated total only once. But functions **asizesof** and **asized** will return a size value respectively an **Asized** instance for each given object, including duplicates.

Definitions[3]

The *length* of an objects like `dict`, `list`, `set`, `str`, `tuple`, etc. is defined as the number of items held in or allocated by the object. Held items are *references* to other objects, called the *referents*.

The *size* of an object is defined as the sum of the *flat size* of the object plus the sizes of any referents[2]. Referents are visited recursively up to the specified detail level. However, the size of objects referenced multiple times is included only once in the total *size*.

The *flat size* of an object is defined as the *basic size* of the object plus the *item size* times the number of allocated *items*, *references* to referents. The *flat size* does include the size for the *references* to the referents, but not the size of the referents themselves.

The *flat size* returned by function *flatsize* equals the result of function *asizeof* with options *code=True*, *ignored=False*, *limit=0* and option *align* set to the same value.

The accurate *flat size* for an object is obtained from function `sys.getsizeof()` where available. Otherwise, the *length* and *size* of sequence objects as `dicts`, `lists`, `sets`, etc. is based on an estimate for the number of allocated items. As a result, the reported *length* and *size* may differ substantially from the actual *length* and *size*.

The *basic* and *item size* are obtained from the `__basicsize__` respectively `__itemsize__` attributes of the (type of the) object. Where necessary (e.g. sequence objects), a zero `__itemsize__` is replaced by the size of a corresponding C type.

The overhead for Python's garbage collector (GC) is included in the *basic size* of (GC managed) objects as well as the space needed for `refcounts` (used only in certain Python builds).

Optionally, size values can be aligned to any power-of-2 multiple.

[4] See Python source file `.../Include/longinterp.h` for the C `typedef` of `digit` used in multi-precision int (or long) objects. The C `sizeof(digit)` in bytes can be obtained in Python from the int (or long) `__itemsize__` attribute. Function **leng** determines the number of `digits` of an int (or long) object.

[3] These definitions and other assumptions are rather arbitrary and may need corrections or adjustments.

Size of (byte)code

The *(byte)code size* of objects like classes, functions, methods, modules, etc. can be included by setting option *code=True*.

Iterators are handled like sequences: iterated object(s) are sized like *referents*[2], but only up to the specified level or recursion *limit* (and only if function `gc.get_referents()` returns the referent object of iterators).

Generators are sized as *(byte)code* only, but the generated objects are never sized.

Old- and New-style Classes

All old- and new-style `class`, instance and `type` objects are handled uniformly such that (a) instance objects are distinguished from class objects and (b) instances of different old-style classes can be dealt with separately.

Class and type objects are represented as `<class ....* def>` respectively `<type ... def>` where the `*` indicates an old-style class and the `... def` suffix marks the *definition object*. Instances of classes are shown as `<class module.name*>` without the `... def` suffix. The `*` after the name indicates an instance of an old-style class.

Ignored Objects

To avoid excessive sizes, several object types are ignored[3] by default, e.g. built-in functions, built-in types and classes[5], function globals and module referents. However, any instances thereof and module objects will be sized when passed as given objects. Ignored object types are included unless option *ignored* is set accordingly.

In addition, many `__...__` attributes of callable objects are ignored[3], except crucial ones, e.g. class attributes `__dict__`, `__doc__`, `__name__` and `__slots__`. For more details, see the type-specific `_..._refs()` and `_len_...()` functions below.

Asizer

class pympler.asizeof.**Asized**(*size*, *flat*, *refs=()*, *name=None*)

Stores the results of an **asized** object in the following 4 attributes:

size – total size of the object (including referents)

flat – flat size of the object (in bytes)

name – name or `repr` of the object

refs – tuple containing an **Asized** instance for each referent

class pympler.asizeof.**Asizer**(***opts*)

Sizer state and options to accumulate sizes.

asized(**objs*, ***opts*)

Size each object and return an **Asized** instance with size information and referents up to the given detail level (and with modified options, see method **set**).

If only one object is given, the return value is the **Asized** instance for that object. The **Asized** size of duplicate and ignored objects will be zero.

asizeof(**objs*, ***opts*)

Return the combined size of the given objects (with modified options, see method **set**).

[5] `Type``s and ``class``es are considered built-in if the ``__module__` of the type or class is listed in the private `_builtin_modules`.

asizesof(**objs*, ***opts*)
Return the individual sizes of the given objects (with modified options, see method **set**).

The size of duplicate and ignored objects will be zero.

exclude_refs(**objs*)
Exclude any references to the specified objects from sizing.

While any references to the given objects are excluded, the objects will be sized if specified as positional arguments in subsequent calls to methods **asizeof** and **asizesof**.

exclude_types(**objs*)
Exclude the specified object instances and types from sizing.

All instances and types of the given objects are excluded, even objects specified as positional arguments in subsequent calls to methods **asizeof** and **asizesof**.

print_profiles(*w=0*, *cutoff=0*, ***print3options*)
Print the profiles above *cutoff* percentage.

The available options and defaults are:

w=0 – indentation for each line

cutoff=0 – minimum percentage printed

print3options – some keyword arguments, like Python 3+ print

print_stats(*objs=()*, *opts={}*, *sized=()*, *sizes=()*, *stats=3*, ***print3options*)
Prints the statistics.

The available options and defaults are:

w=0 – indentation for each line

objs=() – optional, list of objects

opts={} – optional, dict of options used

sized=() – optional, tuple of **Asized** instances returned

sizes=() – optional, tuple of sizes returned

stats=3 – print stats, see function **asizeof**

print3options – some keyword arguments, like Python 3+ print

print_summary(*w=0*, *objs=()*, ***print3options*)
Print the summary statistics.

The available options and defaults are:

w=0 – indentation for each line

objs=() – optional, list of objects

print3options – some keyword arguments, like Python 3+ print

print_typedefs(*w=0*, ***print3options*)
Print the types and dict tables.

The available options and defaults are:

w=0 – indentation for each line

print3options – some keyword arguments, like Python 3+ print

reset(*above=1024*, *align=8*, *clip=80*, *code=False*, *cutoff=10*, *derive=False*, *detail=0*, *frames=False*, *ignored=True*, *infer=False*, *limit=100*, *stats=0*, *stream=None*, ***extra*)

Reset sizing options, state, etc. to defaults.

The available options and default values are:

above=0 – threshold for largest objects stats

align=8 – size alignment

code=False – incl. (byte)code size

cutoff=10 – limit large objects or profiles stats

derive=False – derive from super type

detail=0 – **Asized** refs level

frames=False – ignore frame objects

ignored=True – ignore certain types

infer=False – try to infer types

limit=100 – recursion limit

stats=0 – print statistics, see function **asizeof**

stream=None – output stream for printing

See function **asizeof** for a description of the options.

set(*above=None*, *align=None*, *code=None*, *cutoff=None*, *frames=None*, *detail=None*, *limit=None*, *stats=None*)

Set some sizing options. See also **reset**.

The available options are:

above – threshold for largest objects stats

align – size alignment

code – incl. (byte)code size

cutoff – limit large objects or profiles stats

detail – **Asized** refs level

frames – size or ignore frame objects

limit – recursion limit

stats – print statistics, see function **asizeof**

Any options not set remain unchanged from the previous setting.

Public Functions

`pympler.asizeof.`**adict**(**classes*)

Install one or more classes to be handled as dict.

`pympler.asizeof.`**asized**(**objs*, ***opts*)

Return a tuple containing an **Asized** instance for each object passed as positional argument.

The available options and defaults are:

above=0 – threshold for largest objects stats

align=8 – size alignment

code=False – incl. (byte)code size

cutoff=10 – limit large objects or profiles stats

derive=False – derive from super type

detail=0 – Asized refs level

frames=False – ignore stack frame objects

ignored=True – ignore certain types

infer=False – try to infer types

limit=100 – recursion limit

stats=0 – print statistics

If only one object is given, the return value is the **Asized** instance for that object. Otherwise, the length of the returned tuple matches the number of given objects.

The **Asized** size of duplicate and ignored objects will be zero.

Set *detail* to the desired referents level and *limit* to the maximum recursion depth.

See function **asizeof** for descriptions of the other options.

`pympler.asizeof.`**`asizeof`**`(`**objs*, ***opts*`)`

Return the combined size (in bytes) of all objects passed as positional arguments.

The available options and defaults are:

above=0 – threshold for largest objects stats

align=8 – size alignment

clip=80 – clip `repr()` strings

code=False – incl. (byte)code size

cutoff=10 – limit large objects or profiles stats

derive=False – derive from super type

frames=False – ignore stack frame objects

ignored=True – ignore certain types

infer=False – try to infer types

limit=100 – recursion limit

stats=0 – print statistics

Set *align* to a power of 2 to align sizes. Any value less than 2 avoids size alignment.

If *all* is True and if no positional arguments are supplied. size all current gc objects, including module, global and stack frame objects.

A positive *clip* value truncates all repr() strings to at most *clip* characters.

The (byte)code size of callable objects like functions, methods, classes, etc. is included only if *code* is True.

If *derive* is True, new types are handled like an existing (super) type provided there is one and only of those.

By default certain base types like object, super, etc. are ignored. Set *ignored* to False to include those.

If *infer* is True, new types are inferred from attributes (only implemented for dict types on callable attributes as get, has_key, items, keys and values).

Set *limit* to a positive value to accumulate the sizes of the referents of each object, recursively up to the limit. Using *limit=0* returns the sum of the flat sizes of the given objects. High *limit* values may cause runtime errors and miss objects for sizing.

A positive value for *stats* prints up to 9 statistics, (1) a summary of the number of objects sized and seen and a list of the largests objects with size over *above* bytes, (2) a simple profile of the sized objects by type and (3+) up to 6 tables showing the static, dynamic, derived, ignored, inferred and dict types used, found respectively installed. The fractional part of the *stats* value (x 100) is the number of largest objects shown for (*stats*1.+) or the cutoff percentage for simple profiles for (*stats*=2.+). For example, *stats=1.10* shows the summary and the 10 largest objects, also the default.

See this module documentation for the definition of flat size.

`pympler.asizeof.`**`asizesof`**`(`**objs*, ***opts*`)`

Return a tuple containing the size (in bytes) of all objects passed as positional arguments.

The available options and defaults are:

above=1024 – threshold for largest objects stats

align=8 – size alignment

clip=80 – clip `repr()` strings

code=False – incl. (byte)code size

cutoff=10 – limit large objects or profiles stats

derive=False – derive from super type

frames=False – ignore stack frame objects

ignored=True – ignore certain types

infer=False – try to infer types

limit=100 – recursion limit

stats=0 – print statistics

See function **asizeof** for a description of the options.

The length of the returned tuple equals the number of given objects.

The size of duplicate and ignored objects will be zero.

`pympler.asizeof.`**`basicsize`**`(`*obj*, ***opts*`)`

Return the basic size of an object (in bytes).

The available options and defaults are:

derive=False – derive type from super type

infer=False – try to infer types

save=False – save the object's type definition if new

See this module documentation for the definition of *basic size*.

`pympler.asizeof.`**`flatsize`**`(`*obj*, *align=0*, ***opts*`)`

Return the flat size of an object (in bytes), optionally aligned to the given power of 2.

See function **basicsize** for a description of other available options.

See this module documentation for the definition of *flat size*.

`pympler.asizeof.`**`itemsize`**`(`*obj*, ***opts*`)`

Return the item size of an object (in bytes).

See function **basicsize** for a description of the available options.

See this module documentation for the definition of *item size*.

`pympler.asizeof.`**`leng`**`(`*obj*, ***opts*`)`

Return the length of an object (in items).

See function **basicsize** for a description of the available options.

`pympler.asizeof.`**`refs`**`(`*obj*, ***opts*`)`

Return (a generator for) specific *referents* of an object.

See function **basicsize** for a description of the available options.

7.1.7 Tracking class instances

Introduction

The *ClassTracker* is a facility delivering insight into the memory distribution of a Python program. It can introspect memory consumption of certain classes and objects. Facilities are provided to track and size individual objects or all instances of certain classes. Tracked objects are sized recursively to provide an overview of memory distribution between the different tracked objects.

Usage

Let's start with a simple example. Suppose you have this module:

```
>>> class Employee:
...     pass
...
>>> class Factory:
...     pass
...
>>> def create_factory():
...     factory = Factory()
...     factory.name = "Assembly Line Unlimited"
...     factory.employees = []
...     return factory
...
>>> def populate_factory(factory):
...     for x in xrange(1000):
...         worker = Employee()
...         worker.assigned = factory.name
...         factory.employees.append(worker)
...
>>> factory = create_factory()
>>> populate_factory(factory)
```

The basic tools of the *ClassTracker* are tracking objects or classes, taking snapshots, and printing or dumping statistics. The first step is to decide what to track. Then spots of interest for snapshot creation have to be identified. Finally, the gathered data can be printed or saved:

```
>>> factory = create_factory()
>>> from pympler.classtracker import ClassTracker
>>> tracker = ClassTracker()
>>> tracker.track_object(factory)
>>> tracker.track_class(Employee)
>>> tracker.create_snapshot()
>>> populate_factory(factory)
>>> tracker.create_snapshot()
>>> tracker.stats.print_summary()
---- SUMMARY ------------------------------------------------------------------
                                         active      1.22 MB      average   pct
  Factory                                     1    344      B    344      B    0%
  __main__.Employee                           0      0      B      0      B    0%
                                         active      1.42 MB      average   pct
  Factory                                     1      4.75 KB      4.75 KB    0%
  __main__.Employee                        1000    195.38 KB    200      B   13%
-------------------------------------------------------------------------------
```

Basic Functionality

Instance Tracking

The purpose of instance tracking is to observe the size and lifetime of an object of interest. Creation and destruction timestamps are recorded and the size of the object is sampled when taking a snapshot.

To track the size of an individual object:

```
from pympler.classtracker import ClassTracker
tracker = ClassTracker()
obj = MyClass()
tracker.track_object(obj)
```

Class Tracking

Most of the time it’s cumbersome to track individual instances manually. Instead, all instances of a class can automatically be tracked with *track_class*:

```
tracker.track_class(MyClass)
```

All instances of *MyClass* (or a class that inherits from *MyClass*) created hereafter are tracked.

Tracked Object Snapshot

Tracking alone will not reveal the size of an object. The idea of the *ClassTracker* is to sample the sizes of all tracked objects at configurable instants in time. The *create_snapshot* function computes the size of all tracked objects:

```
tracker.create_snapshot('Before juggling with tracked objects')
...
tracker.create_snapshot('Juggling aftermath')
```

With this information, the distribution of the allocated memory can be apportioned to tracked classes and instances.

Print Statistics

The gathered data can be investigated with *print_stats*. This prints all available data. To filter and limit the output the more powerful "Off-line analysis" API can be used instead.

Advanced Functionality

Per-referent Sizing

It may not be enough to know the total memory consumption of an object. Detailed per-referent statistics can be gathered recursively up to a given resolution level. Resolution level 1 means that all direct referents of an object will be sized. Level 2 also include the referents of the direct referents, and so forth. Note that the member variables of an instance are typically stored in a dictionary and are therefore second order referents.

```
tracker.track_object(obj, resolution_level=2)
```

The resolution level can be changed if the object is already tracked:

```
tracker.track_change(obj, resolution_level=2)
```

The new setting will become effective for the next snapshot. This can help to raise the level of detail for a specific instance of a tracked class without logging all the class' instances with a high verbosity level. Nevertheless, the resolution level can also be set for all instances of a class:

```
tracker.track_class(MyClass, resolution_level=1)
```

Warning: Please note the per-referent sizing is very memory and computationally intensive. The recorded meta-data must be stored for each referent of a tracked object which might easily quadruplicate the memory footprint of the build. Handle with care and don't use too high resolution levels, especially if set via *track_class*.

Instantiation traces

Sometimes it is not trivial to observe where an object was instantiated. The *ClassTracker* can record the instantiation stack trace for later evaluation.

```
tracker.track_class(MyClass, trace=1)
```

This only works with tracked classes, and **not** with individual objects.

Background Monitoring

The *ClassTracker* can be configured to take periodic snapshots automatically. The following example will take 10 snapshots a second (approximately) until the program has exited or the periodic snapshots are stopped with *stop_periodic_snapshots*. Background monitoring also works if no object is tracked. In this mode, the *ClassTracker* will only record the total virtual memory associated with the program. This can be useful in combination with background monitoring to detect memory usage which is transient or not associated with any tracked object.

```
tracker.start_periodic_snapshots(interval=0.1)
```

Warning: Take care if you use automatic snapshots with tracked objects. The sizing of individual objects might be inconsistent when memory is allocated or freed while the snapshot is being taken.

Off-line Analysis

The more data is gathered by the *ClassTracker* the more noise is produced on the console. The acquired *ClassTracker* log data can also be saved to a file for off-line analysis:

```
tracker.stats.dump_stats('profile.dat')
```

The *Stats* class of the *ClassTracker* provides means to evaluate the collected data. The API is inspired by the Stats class of the Python profiler. It is possible to sort the data based on user preferences, filter by class and limit the output noise to a manageable magnitude.

The following example reads the dumped data and prints the ten largest Node objects to the standard output:

```
from pympler.classtracker_stats import ConsoleStats

stats = ConsoleStats()
stats.load_stats('profile.dat')
stats.sort_stats('size').print_stats(limit=10, clsname='Node')
```

HTML Statistics

The *ClassTracker* data can also be emitted in HTML format together with a number of charts (needs python-matplotlib). HTML statistics can be emitted using the *HtmlStats* class:

```
from pympler.classtracker_stats import HtmlStats
HtmlStats(tracker=tracker).create_html('profile.html')
```

However, you can also reprocess a previously generated dump:

```
from pympler.classtracker_stats import HtmlStats

stats = HtmlStats(filename='profile.dat')
stats.create_html('profile.html')
```

Limitations and Corner Cases

Inheritance

Class tracking allows to observe multiple classes that might have an inheritance relationship. An object is only tracked once. The tracking parameters of the most specialized tracked class control the actual tracking of an instance.

Shared Data

Data shared between multiple tracked objects won’t lead to overestimations. Shared data will be assigned to the first (evaluated) tracked object it is referenced from, but is only counted once. Tracked objects are evaluated in the order they were announced to the *ClassTracker*. This should make the assignment deterministic from one run to the next, but has two known problems. If the *ClassTracker* is used concurrently from multiple threads, the announcement order will likely change and may lead to random assignment of shared data to different objects. Shared data might also be assigned to different objects during its lifetime, see the following example:

```
class A():
  pass

from pympler.classtracker import ClassTracker
tracker = ClassTracker()

a = A()
tracker.track_object(a)
b = A()
tracker.track_object(b)
b.content = range(100000)
tracker.create_snapshot('#1')
a.notmine = b.content
tracker.create_snapshot('#2')
```

In the snapshot #1, *b*'s size will include the size of the large list. Then the list is shared with *a*. The snapshot *#2* will assign the list's footprint to *a* because it was registered before *b*.

If a tracked object *A* is referenced from another tracked object *B*, *A*'s size is not added to *B*'s size, regardless of the order in which they are sized.

Accuracy

ClassTracker uses the *sizer* module to gather size informations. Asizeof makes assumptions about the memory footprint of the various data types. As it is implemented in pure Python, there is no way to know how the actual Python implementation allocates data and lays it out in memory. Thus, the size numbers are not really accurate and there will always be a divergence between the virtual size of the Python process as reported by the OS and the sizes asizeof estimates.

Most recent C/Python versions contain a facility to report accurate size informations of Python objects. If available, asizeof uses it to improve the accuracy.

Morphing objects

Some programs instate the (anti-)pattern of changing an instance' class at runtime, for example to morph abstract objects into specific derivations during runtime. The pattern looks like the following in the code:

```
obj.__class__ = OtherClass
```

If the instance which is morphed is already tracked, the instance will continue to be tracked by the *ClassTracker*. If the target class is tracked but the instance is not, the instance will only be tracked if the constructor of the target class is called as part of the morphing process. The object will not be re-registered to the new class in the tracked object index. However, the new class is stored in the representation of the object as soon as the object is sized.

7.1.8 Identifying memory leaks

Muppy tries to help developers to identity memory leaks of Python applications. It enables the tracking of memory usage during runtime and the identification of objects which are leaking. Additionally, tools are provided which allow to locate the source of not released objects.

Muppy is (yet another) Memory Usage Profiler for Python. The focus of this toolset is laid on the identification of memory leaks. Let's have a look what you can do with muppy.

The muppy module

Muppy allows you to get hold of all objects,

```
>>> from pympler import muppy
>>> all_objects = muppy.get_objects()
>>> len(all_objects)                           # doctest: +SKIP
19700
```

or filter out certain types of objects.

```
>>> import types
>>> my_types = muppy.filter(all_objects, Type=types.ClassType)
>>> len(my_types)                                    # doctest: +SKIP
72
>>> for t in my_types:
...     print t
...                                               # doctest: +SKIP
UserDict.IterableUserDict
UserDict.UserDict
UserDict.DictMixin
os._Environ
sre_parse.Tokenizer
sre_parse.SubPattern
re.Scanner
string._multimap
distutils.log.Log
encodings.utf_8.StreamWriter
encodings.utf_8.StreamReader
codecs.StreamWriter
codecs.StreamReader
codecs.StreamReaderWriter
codecs.Codec
codecs.StreamRecoder
tokenize.Untokenizer
inspect.BlockFinder
sre_parse.Pattern
. . .
```

This result, for example, tells us that the number of lists remained the same, but the memory allocated by lists has increased by 8 bytes. The correct increase for a LP64 system (see 64-Bit_Programming_Models).

The summary module

You can create summaries

```
>>> from pympler import summary
>>> sum1 = summary.summarize(all_objects)
>>> summary.print_(sum1)                          # doctest: +SKIP
                       types |   # objects |   total size
============================ | =========== | ============
                        dict |         546 |    953.30 KB
```

(continues on next page)

(continued from previous page)

```
                          str |        8270 |    616.46 KB
                         list |         127 |    529.44 KB
                        tuple |        5021 |    410.62 KB
                         code |        1378 |    161.48 KB
                         type |          70 |     61.80 KB
           wrapper_descriptor |         508 |     39.69 KB
  builtin_function_or_method |         515 |     36.21 KB
                          int |         900 |     21.09 KB
            method_descriptor |         269 |     18.91 KB
                      weakref |         177 |     15.21 KB
           <class 'abc.ABCMeta |          16 |     14.12 KB
                          set |          48 |     10.88 KB
          function (__init__) |          81 |      9.49 KB
            member_descriptor |         131 |      9.21 KB
```

and compare them with other summaries.

```
>>> sum2 = summary.summarize(muppy.get_objects())
>>> diff = summary.get_diff(sum1, sum2)
>>> summary.print_(diff)                          # doctest: +SKIP
                          types |   # objects |   total size
=============================== | =========== | ============
                           list |        1097 |      1.07 MB
                            str |        1105 |     68.21 KB
                           dict |          14 |     21.08 KB
             wrapper_descriptor |         215 |     16.80 KB
                            int |         121 |      2.84 KB
                          tuple |          30 |      2.02 KB
              member_descriptor |          25 |      1.76 KB
                        weakref |          14 |      1.20 KB
              getset_descriptor |          15 |      1.05 KB
              method_descriptor |          12 |    864     B
  frame (codename: get_objects) |           1 |    488     B
     builtin_function_or_method |           6 |    432     B
     frame (codename: <module>) |           1 |    424     B
         classmethod_descriptor |           3 |    216     B
                           code |           1 |    120     B
```

The tracker module

Of course we don't have to do all these steps manually, instead we can use muppy's tracker.

```
>>> from pympler import tracker
>>> tr = tracker.SummaryTracker()
>>> tr.print_diff()                               # doctest: +SKIP
                                        types |   # objects |   total size
======================================= | =========== | ============
                                         list |        1095 |    160.78 KB
                                          str |        1093 |     66.33 KB
                                          int |         120 |      2.81 KB
                                         dict |           3 |    840     B
      frame (codename: create_summary) |           1 |    560     B
          frame (codename: print_diff) |           1 |    480     B
                frame (codename: diff) |           1 |    464     B
                 function (store_info) |           1 |    120     B
```

(continues on next page)

(continued from previous page)

```
                                            cell |              2 |      112      B
```

A tracker object creates a summary (that is a summary which it will remember) on initialization. Now whenever you call tracker.print_diff(), a new summary of the current state is created, compared to the previous summary and printed to the console. As you can see here, quite a few objects got in between these two invocations. But if you don't do anything, nothing will change.

```
>>> tr.print_diff()                                 # doctest: +SKIP
  types |   # objects |   total size
======= | =========== | ============
```

Now check out this code snippet

```
>>> i = 1
>>> l = [1,2,3,4]
>>> d = {}
>>> tr.print_diff()                                 # doctest: +SKIP
  types |   # objects |   total size
======= | =========== | ============
   dict |           1 |    280     B
   list |           1 |    192     B
```

As you can see both, the new list and the new dict appear in the summary, but not the 4 integers used. Why is that? Because they existed already before they were used here, that is some other part in the Python interpreter code makes already use of them. Thus, they are not new.

The refbrowser module

In case some objects are leaking and you don't know where they are still referenced, you can use the referrers browser. At first let's create a root object which we then reference from a tuple and a list.

```
>>> from pympler import refbrowser
>>> root = "some root object"
>>> root_ref1 = [root]
>>> root_ref2 = (root, )
```

```
>>> def output_function(o):
...     return str(type(o))
...
>>> cb = refbrowser.ConsoleBrowser(root, maxdepth=2, str_func=output_function)
```

Then we create a ConsoleBrowser, which will give us a referrers tree starting at *root*, printing to a maximum depth of 2, and uses *str_func* to represent objects. Now it's time to see where we are at.

```
>>> cb.print_tree()                                 # doctest: +SKIP
<type 'str'>-+-<type 'dict'>-+-<type 'list'>
             |               +-<type 'list'>
             |               +-<type 'list'>
             |
             +-<type 'dict'>-+-<type 'module'>
             |               +-<type 'list'>
             |               +-<type 'frame'>
             |               +-<type 'function'>
             |               +-<type 'list'>
```

(continues on next page)

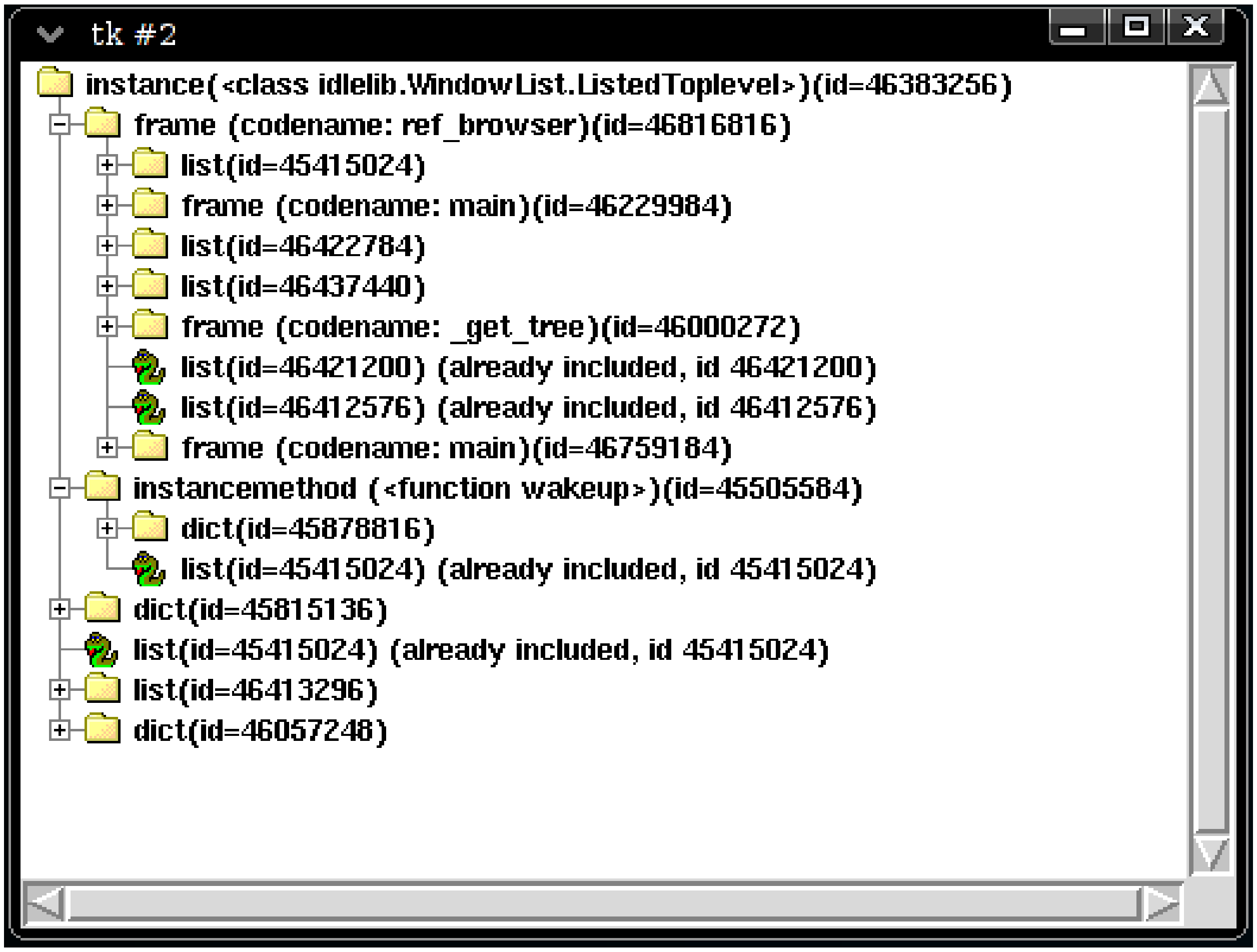
(continued from previous page)

```
             |                +-<type 'frame'>
             |                +-<type 'list'>
             |                +-<type 'function'>
             |                +-<type 'frame'>
             |
             +-<type 'list'>--<type 'dict'>
             +-<type 'tuple'>--<type 'dict'>
             +-<type 'dict'>--<class 'muppy.refbrowser.ConsoleBrowser'>
```

What we see is that the root object is referenced by the tuple and the list, as well as by three dictionaries. These dictionaries belong to the environment, e.g. the ConsoleBrowser we just started and the current execution context.

This console browsing is of course kind of inconvenient. Much better would be an InteractiveBrowser. Let's see what we got.

```
>>> from pympler import refbrowser
>>> ib = refbrowser.InteractiveBrowser(root)
>>> ib.main()
```



```
instance(<class idlelib.WindowList.ListedToplevel>)(id=46383256)
  frame (codename: ref_browser)(id=46816816)
    list(id=45415024)
    frame (codename: main)(id=46229984)
    list(id=46422784)
    list(id=46437440)
    frame (codename: _get_tree)(id=46000272)
    list(id=46421200) (already included, id 46421200)
    list(id=46412576) (already included, id 46412576)
    frame (codename: main)(id=46759184)
  instancemethod (<function wakeup>)(id=45505584)
    dict(id=45878816)
    list(id=45415024) (already included, id 45415024)
  dict(id=45815136)
  list(id=45415024) (already included, id 45415024)
  list(id=46413296)
  dict(id=46057248)
```

Now you can click through all referrers of the root object.

7.1.9 Tracking memory in Django

Introduction

Pympler includes a memory profile panel for Django that integrates with the Django Debug Toolbar. It shows process memory information and model instances for the current request.

Usage

Pympler adds a memory panel as a third party addon – it's not included in the Django Debug Toolbar. It can be added by overriding the *DEBUG_TOOLBAR_PANELS* setting in the Django project settings:

```
DEBUG_TOOLBAR_PANELS = (
    'debug_toolbar.panels.timer.TimerDebugPanel',
    'pympler.panels.MemoryPanel',
    )
```

Pympler also needs to be added to the *INSTALLED_APPS* in the Django settings:

```
INSTALLED_APPS = INSTALLED_APPS + ('debug_toolbar', 'pympler')
```

Known issues

Pympler doesn't correctly handle tracking calls from concurrent threads. In order to get accurate instance counts and sizes, it's recommended to only use single-threaded web servers for memory profiling, e.g.:

```
django-admin runserver --nothreading
```

Expose a memory-profiling panel to the Django Debug toolbar.

Shows process memory information (virtual size, resident set size) and model instances for the current request.

Requires Django and Django Debug toolbar:

https://github.com/jazzband/django-debug-toolbar

Pympler adds a memory panel as a third party addon (not included in the django-debug-toolbar). It can be added by overriding the *DEBUG_TOOLBAR_PANELS* setting in the Django project settings:

```
DEBUG_TOOLBAR_PANELS = (
    'debug_toolbar.panels.timer.TimerDebugPanel',
    'pympler.panels.MemoryPanel',
    )
```

Pympler also needs to be added to the *INSTALLED_APPS* in the Django settings:

```
INSTALLED_APPS = INSTALLED_APPS + ('debug_toolbar', 'pympler')
```

7.1.10 Library

Some functions of the library work on the entire object set of your running Python application. Expect some time-intensive computations.

Modules

pympler.asizeof

Introduction

This module exposes 9 functions and 2 classes to obtain lengths and sizes of Python objects (for Python 2.6 or later).

Earlier versions of this module supported Python versions down to Python 2.2. If you are using Python 2.5 or older, please consider downgrading Pympler to version 0.3.x.

Public Functions[1]

Function **asizeof** calculates the combined (approximate) size in bytes of one or several Python objects.

Function **asizesof** returns a tuple containing the (approximate) size in bytes for each given Python object separately.

Function **asized** returns for each object an instance of class **Asized** containing all the size information of the object and a tuple with the referents[2].

Functions **basicsize** and **itemsize** return the *basic*- respectively *itemsize* of the given object, both in bytes. For objects as `array.array`, `numpy.array`, `numpy.matrix`, etc. where the item size varies depending on the instance-specific data type, function **itemsize** returns that item size.

Function **flatsize** returns the *flat size* of a Python object in bytes defined as the *basic size* plus the *item size* times the *length* of the given object.

Function **leng** returns the *length* of an object, like standard function `len` but extended for several types. E.g. the **leng** of a multi-precision int (or long) is the number of `digits`[4]. The length of most *mutable* sequence objects includes an estimate of the over-allocation and therefore, the **leng** value may differ from the standard `len` result. For objects like `array.array`, `numpy.array`, `numpy.matrix`, etc. function **leng** returns the proper number of items.

Function **refs** returns (a generator for) the referents[2] of the given object.

Certain classes are known to be sub-classes of or to behave as `dict` objects. Function **adict** can be used to register other class objects to be treated like `dict`.

Public Classes[1]

Class **Asizer** may be used to accumulate the results of several **asizeof** or **asizesof** calls. After creating an **Asizer** instance, use methods **asizeof** and **asizesof** as needed to size any number of additional objects.

Call methods **exclude_refs** and/or **exclude_types** to exclude references to respectively instances or types of certain objects.

Use one of the **print_...** methods to report the statistics.

An instance of class **Asized** is returned for each object sized by the **asized** function or method.

Duplicate Objects

Any duplicate, given objects are sized only once and the size is included in the accumulated total only once. But functions **asizesof** and **asized** will return a size value respectively an **Asized** instance for each given object, including duplicates.

[1] The functions and classes in this module are not thread-safe.

[2] The *referents* of an object are the objects referenced *by* that object. For example, the *referents* of a `list` are the objects held in the `list`, the *referents* of a `dict` are the key and value objects in the `dict`, etc.

[4] See Python source file `.../Include/longinterp.h` for the C `typedef` of `digit` used in multi-precision int (or long) objects. The C `sizeof(digit)` in bytes can be obtained in Python from the int (or long) `__itemsize__` attribute. Function **leng** determines the number of `digits` of an int (or long) object.

Definitions[3]

The *length* of an objects like `dict`, `list`, `set`, `str`, `tuple`, etc. is defined as the number of items held in or allocated by the object. Held items are *references* to other objects, called the *referents*.

The *size* of an object is defined as the sum of the *flat size* of the object plus the sizes of any referents[2]. Referents are visited recursively up to the specified detail level. However, the size of objects referenced multiple times is included only once in the total *size*.

The *flat size* of an object is defined as the *basic size* of the object plus the *item size* times the number of allocated *items*, *references* to referents. The *flat size* does include the size for the *references* to the referents, but not the size of the referents themselves.

The *flat size* returned by function *flatsize* equals the result of function *asizeof* with options *code=True*, *ignored=False*, *limit=0* and option *align* set to the same value.

The accurate *flat size* for an object is obtained from function `sys.getsizeof()` where available. Otherwise, the *length* and *size* of sequence objects as `dicts`, `lists`, `sets`, etc. is based on an estimate for the number of allocated items. As a result, the reported *length* and *size* may differ substantially from the actual *length* and *size*.

The *basic* and *item size* are obtained from the `__basicsize__` respectively `__itemsize__` attributes of the (type of the) object. Where necessary (e.g. sequence objects), a zero `__itemsize__` is replaced by the size of a corresponding C type.

The overhead for Python's garbage collector (GC) is included in the *basic size* of (GC managed) objects as well as the space needed for `refcounts` (used only in certain Python builds).

Optionally, size values can be aligned to any power-of-2 multiple.

Size of (byte)code

The *(byte)code size* of objects like classes, functions, methods, modules, etc. can be included by setting option *code=True*.

Iterators are handled like sequences: iterated object(s) are sized like *referents*[2], but only up to the specified level or recursion *limit* (and only if function `gc.get_referents()` returns the referent object of iterators).

Generators are sized as *(byte)code* only, but the generated objects are never sized.

Old- and New-style Classes

All old- and new-style `class`, instance and `type` objects are handled uniformly such that (a) instance objects are distinguished from class objects and (b) instances of different old-style classes can be dealt with separately.

Class and type objects are represented as `<class ....* def>` respectively `<type ... def>` where the `*` indicates an old-style class and the `... def` suffix marks the *definition object*. Instances of classes are shown as `<class module.name*>` without the `... def` suffix. The `*` after the name indicates an instance of an old-style class.

Ignored Objects

To avoid excessive sizes, several object types are ignored[3] by default, e.g. built-in functions, built-in types and classes[5], function globals and module referents. However, any instances thereof and module objects will be sized when passed as given objects. Ignored object types are included unless option *ignored* is set accordingly.

[3] These definitions and other assumptions are rather arbitrary and may need corrections or adjustments.

[5] `Type``s and ``class``es are considered built-in if the ``__module__` of the type or class is listed in the private `_builtin_modules`.

In addition, many `__...__` attributes of callable objects are ignored[3], except crucial ones, e.g. class attributes `__dict__`, `__doc__`, `__name__` and `__slots__`. For more details, see the type-specific `_..._refs()` and `_len_...()` functions below.

Asizer

class `pympler.asizeof.`**`Asized`**(*size*, *flat*, *refs=()*, *name=None*)

Stores the results of an **asized** object in the following 4 attributes:

size – total size of the object (including referents)

flat – flat size of the object (in bytes)

name – name or `repr` of the object

refs – tuple containing an **Asized** instance for each referent

class `pympler.asizeof.`**`Asizer`**(***opts*)

Sizer state and options to accumulate sizes.

`asized`(**objs*, ***opts*)

Size each object and return an **Asized** instance with size information and referents up to the given detail level (and with modified options, see method **set**).

If only one object is given, the return value is the **Asized** instance for that object. The **Asized** size of duplicate and ignored objects will be zero.

`asizeof`(**objs*, ***opts*)

Return the combined size of the given objects (with modified options, see method **set**).

`asizesof`(**objs*, ***opts*)

Return the individual sizes of the given objects (with modified options, see method **set**).

The size of duplicate and ignored objects will be zero.

`exclude_refs`(**objs*)

Exclude any references to the specified objects from sizing.

While any references to the given objects are excluded, the objects will be sized if specified as positional arguments in subsequent calls to methods **asizeof** and **asizesof**.

`exclude_types`(**objs*)

Exclude the specified object instances and types from sizing.

All instances and types of the given objects are excluded, even objects specified as positional arguments in subsequent calls to methods **asizeof** and **asizesof**.

`print_profiles`(*w=0*, *cutoff=0*, ***print3options*)

Print the profiles above *cutoff* percentage.

The available options and defaults are:

w=0 – indentation for each line

cutoff=0 – minimum percentage printed

print3options – some keyword arguments, like Python 3+ print

`print_stats`(*objs=()*, *opts={}*, *sized=()*, *sizes=()*, *stats=3*, ***print3options*)

Prints the statistics.

The available options and defaults are:

w=0 – indentation for each line

objs=() – optional, list of objects

opts={} – optional, dict of options used

sized=() – optional, tuple of **Asized** instances returned

sizes=() – optional, tuple of sizes returned

stats=3 – print stats, see function **asizeof**

print3options – some keyword arguments, like Python 3+ print

`print_summary`(*w=0*, *objs=()*, ***print3options*)
Print the summary statistics.

The available options and defaults are:

w=0 – indentation for each line

objs=() – optional, list of objects

print3options – some keyword arguments, like Python 3+ print

`print_typedefs`(*w=0*, ***print3options*)
Print the types and dict tables.

The available options and defaults are:

w=0 – indentation for each line

print3options – some keyword arguments, like Python 3+ print

`set`(*above=None*, *align=None*, *code=None*, *cutoff=None*, *frames=None*, *detail=None*, *limit=None*, *stats=None*)
Set some sizing options. See also **reset**.

The available options are:

above – threshold for largest objects stats

align – size alignment

code – incl. (byte)code size

cutoff – limit large objects or profiles stats

detail – **Asized** refs level

frames – size or ignore frame objects

limit – recursion limit

stats – print statistics, see function **asizeof**

Any options not set remain unchanged from the previous setting.

`reset`(*above=1024*, *align=8*, *clip=80*, *code=False*, *cutoff=10*, *derive=False*, *detail=0*, *frames=False*, *ignored=True*, *infer=False*, *limit=100*, *stats=0*, *stream=None*, ***extra*)
Reset sizing options, state, etc. to defaults.

The available options and default values are:

above=0 – threshold for largest objects stats

align=8 – size alignment

code=False – incl. (byte)code size

cutoff=10 – limit large objects or profiles stats

derive=False – derive from super type

detail=0 – **Asized** refs level

frames=False – ignore frame objects

ignored=True – ignore certain types

infer=False – try to infer types

limit=100 – recursion limit

stats=0 – print statistics, see function **asizeof**

stream=None – output stream for printing

See function **asizeof** for a description of the options.

Public Functions

`pympler.asizeof.`**`adict`**`(`**classes*`)`

Install one or more classes to be handled as dict.

`pympler.asizeof.`**`asized`**`(`**objs*, ***opts*`)`

Return a tuple containing an **Asized** instance for each object passed as positional argument.

The available options and defaults are:

above=0 – threshold for largest objects stats

align=8 – size alignment

code=False – incl. (byte)code size

cutoff=10 – limit large objects or profiles stats

derive=False – derive from super type

detail=0 – Asized refs level

frames=False – ignore stack frame objects

ignored=True – ignore certain types

infer=False – try to infer types

limit=100 – recursion limit

stats=0 – print statistics

If only one object is given, the return value is the **Asized** instance for that object. Otherwise, the length of the returned tuple matches the number of given objects.

The **Asized** size of duplicate and ignored objects will be zero.

Set *detail* to the desired referents level and *limit* to the maximum recursion depth.

See function **asizeof** for descriptions of the other options.

`pympler.asizeof.`**`asizeof`**`(`**objs*, ***opts*`)`

Return the combined size (in bytes) of all objects passed as positional arguments.

The available options and defaults are:

above=0 – threshold for largest objects stats

align=8 – size alignment

clip=80 – clip `repr()` strings

code=False – incl. (byte)code size

cutoff=10 – limit large objects or profiles stats

derive=False – derive from super type

frames=False – ignore stack frame objects

ignored=True – ignore certain types

infer=False – try to infer types

limit=100 – recursion limit

stats=0 – print statistics

Set *align* to a power of 2 to align sizes. Any value less than 2 avoids size alignment.

If *all* is True and if no positional arguments are supplied. size all current gc objects, including module, global and stack frame objects.

A positive *clip* value truncates all repr() strings to at most *clip* characters.

The (byte)code size of callable objects like functions, methods, classes, etc. is included only if *code* is True.

If *derive* is True, new types are handled like an existing (super) type provided there is one and only of those.

By default certain base types like object, super, etc. are ignored. Set *ignored* to False to include those.

If *infer* is True, new types are inferred from attributes (only implemented for dict types on callable attributes as get, has_key, items, keys and values).

Set *limit* to a positive value to accumulate the sizes of the referents of each object, recursively up to the limit. Using *limit=0* returns the sum of the flat sizes of the given objects. High *limit* values may cause runtime errors and miss objects for sizing.

A positive value for *stats* prints up to 9 statistics, (1) a summary of the number of objects sized and seen and a list of the largests objects with size over *above* bytes, (2) a simple profile of the sized objects by type and (3+) up to 6 tables showing the static, dynamic, derived, ignored, inferred and dict types used, found respectively installed. The fractional part of the *stats* value (x 100) is the number of largest objects shown for (*stats*1.+) or the cutoff percentage for simple profiles for (*stats*=2.+). For example, *stats=1.10* shows the summary and the 10 largest objects, also the default.

See this module documentation for the definition of flat size.

`pympler.asizeof.`**`asizesof`**`(*objs, **opts)`

Return a tuple containing the size (in bytes) of all objects passed as positional arguments.

The available options and defaults are:

above=1024 – threshold for largest objects stats

align=8 – size alignment

clip=80 – clip `repr()` strings

code=False – incl. (byte)code size

cutoff=10 – limit large objects or profiles stats

derive=False – derive from super type

frames=False – ignore stack frame objects

ignored=True – ignore certain types

infer=False – try to infer types

limit=100 – recursion limit

stats=0 – print statistics

See function **asizeof** for a description of the options.

The length of the returned tuple equals the number of given objects.

The size of duplicate and ignored objects will be zero.

`pympler.asizeof.`**`basicsize`**`(`*obj*, ***opts*`)`

Return the basic size of an object (in bytes).

The available options and defaults are:

derive=False – derive type from super type

infer=False – try to infer types

save=False – save the object's type definition if new

See this module documentation for the definition of *basic size*.

`pympler.asizeof.`**`flatsize`**`(`*obj*, *align=0*, ***opts*`)`

Return the flat size of an object (in bytes), optionally aligned to the given power of 2.

See function **basicsize** for a description of other available options.

See this module documentation for the definition of *flat size*.

`pympler.asizeof.`**`itemsize`**`(`*obj*, ***opts*`)`

Return the item size of an object (in bytes).

See function **basicsize** for a description of the available options.

See this module documentation for the definition of *item size*.

`pympler.asizeof.`**`leng`**`(`*obj*, ***opts*`)`

Return the length of an object (in items).

See function **basicsize** for a description of the available options.

`pympler.asizeof.`**`refs`**`(`*obj*, ***opts*`)`

Return (a generator for) specific *referents* of an object.

See function **basicsize** for a description of the available options.

pympler.classtracker

The *ClassTracker* is a facility delivering insight into the memory distribution of a Python program. It can introspect memory consumption of certain classes and objects. Facilities are provided to track and size individual objects or all instances of certain classes. Tracked objects are sized recursively to provide an overview of memory distribution between the different tracked objects.

Classes

class pympler.classtracker.**ClassTracker**(*stream=None*)

clear()
Clear all gathered data and detach from all tracked objects/classes.

create_snapshot(*description=”*, *compute_total=False*)
Collect current per instance statistics and saves total amount of memory associated with the Python process.

If *compute_total* is *True*, the total consumption of all objects known to *asizeof* is computed. The latter might be very slow if many objects are mapped into memory at the time the snapshot is taken. Therefore, *compute_total* is set to *False* by default.

The overhead of the *ClassTracker* structure is also computed.

Snapshots can be taken asynchronously. The function is protected with a lock to prevent race conditions.

detach_all()
Detach from all tracked classes and objects. Restore the original constructors and cleanse the tracking lists.

detach_all_classes()
Detach from all tracked classes.

detach_class(*cls*)
Stop tracking class ‘cls’. Any new objects of that type are not tracked anymore. Existing objects are still tracked.

start_periodic_snapshots(*interval=1.0*)
Start a thread which takes snapshots periodically. The *interval* specifies the time in seconds the thread waits between taking snapshots. The thread is started as a daemon allowing the program to exit. If periodic snapshots are already active, the interval is updated.

stop_periodic_snapshots()
Post a stop signal to the thread that takes the periodic snapshots. The function waits for the thread to terminate which can take some time depending on the configured interval.

track_class(*cls*, *name=None*, *resolution_level=0*, *keep=False*, *trace=False*)
Track all objects of the class *cls*. Objects of that type that already exist are *not* tracked. If *track_class* is called for a class already tracked, the tracking parameters are modified. Instantiation traces can be generated by setting *trace* to True. A constructor is injected to begin instance tracking on creation of the object. The constructor calls *track_object* internally.

Parameters

- **cls** – class to be tracked, may be an old-style or a new-style class
- **name** – reference the class by a name, default is the concatenation of module and class name
- **resolution_level** – The recursion depth up to which referents are sized individually. Resolution level 0 (default) treats the object as an opaque entity, 1 sizes all direct referents individually, 2 also sizes the referents of the referents and so forth.
- **keep** – Prevent the object’s deletion by keeping a (strong) reference to the object.
- **trace** – Save instantiation stack trace for each instance

track_object(*instance*, *name=None*, *resolution_level=0*, *keep=False*, *trace=False*)
Track object ‘instance’ and sample size and lifetime information. Not all objects can be tracked; trackable

objects are class instances and other objects that can be weakly referenced. When an object cannot be tracked, a *TypeError* is raised.

Parameters

- **`resolution_level`** – The recursion depth up to which referents are sized individually. Resolution level 0 (default) treats the object as an opaque entity, 1 sizes all direct referents individually, 2 also sizes the referents of the referents and so forth.
- **`keep`** – Prevent the object's deletion by keeping a (strong) reference to the object.

pympler.classtracker_stats

Provide saving, loading and presenting gathered *ClassTracker* statistics.

Classes

class `pympler.classtracker_stats.`**`Stats`**(*tracker=None*, *filename=None*, *stream=None*)

Presents the memory statistics gathered by a *ClassTracker* based on user preferences.

`__init__`(*tracker=None*, *filename=None*, *stream=None*)

Initialize the data log structures either from a *ClassTracker* instance (argument *tracker*) or a previously dumped file (argument *filename*).

Parameters

- **`tracker`** – ClassTracker instance
- **`filename`** – filename of previously dumped statistics
- **`stream`** – where to print statistics, defaults to `sys.stdout`

`dump_stats`(*fdump*, *close=True*)

Dump the logged data to a file. The argument *file* can be either a filename or an open file object that requires write access. *close* controls if the file is closed before leaving this method (the default behaviour).

`load_stats`(*fdump*)

Load the data from a dump file. The argument *fdump* can be either a filename or an open file object that requires read access.

`reverse_order`()

Reverse the order of the tracked instance index *self.sorted*.

`sort_stats`(**args*)

Sort the tracked objects according to the supplied criteria. The argument is a string identifying the basis of a sort (example: 'size' or 'classname'). When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, `sort_stats('name', 'size')` will sort all the entries according to their class name, and resolve all ties (identical class names) by sorting by size. The criteria are fields in the tracked object instances. Results are stored in the `self.sorted` list which is used by `Stats.print_stats()` and other methods. The fields available for sorting are:

'classname'	the name with which the class was registered
'name'	the classname
'birth'	creation timestamp
'death'	destruction timestamp

'size' the maximum measured size of the object

'tsize' the measured size during the largest snapshot

'repr' string representation of the object

Note that sorts on size are in descending order (placing most memory consuming items first), whereas name, repr, and creation time searches are in ascending order (alphabetical).

The function returns self to allow calling functions on the result:

```
stats.sort_stats('size').reverse_order().print_stats()
```

class pympler.classtracker_stats.**ConsoleStats**(*tracker=None*, *filename=None*, *stream=None*)

Presentation layer for *Stats* to be used in text-based consoles.

__init__(*tracker=None*, *filename=None*, *stream=None*)

Initialize the data log structures either from a *ClassTracker* instance (argument *tracker*) or a previously dumped file (argument *filename*).

Parameters

- **tracker** – ClassTracker instance
- **filename** – filename of previously dumped statistics
- **stream** – where to print statistics, defaults to `sys.stdout`

print_stats(*clsname=None*, *limit=1.0*)

Write tracked objects to stdout. The output can be filtered and pruned. Only objects are printed whose classname contain the substring supplied by the *clsname* argument. The output can be pruned by passing a *limit* value.

Parameters

- **clsname** – Only print objects whose classname contain the given substring.
- **limit** – If *limit* is a float smaller than one, only the supplied percentage of the total tracked data is printed. If *limit* is bigger than one, this number of tracked objects are printed. Tracked objects are first filtered, and then pruned (if specified).

print_summary()

Print per-class summary for each snapshot.

class pympler.classtracker_stats.**HtmlStats**(*tracker=None*, *filename=None*, *stream=None*)

Output the *ClassTracker* statistics as HTML pages and graphs.

__init__(*tracker=None*, *filename=None*, *stream=None*)

Initialize the data log structures either from a *ClassTracker* instance (argument *tracker*) or a previously dumped file (argument *filename*).

Parameters

- **tracker** – ClassTracker instance
- **filename** – filename of previously dumped statistics
- **stream** – where to print statistics, defaults to `sys.stdout`

create_html(*fname*, *title='ClassTracker Statistics'*)

Create HTML page *fname* and additional files in a directory derived from *fname*.

pympler.garbagegraph

Garbage occurs if objects refer too each other in a circular fashion. Such reference cycles cannot be freed automatically and must be collected by the garbage collector. While it is sometimes hard to avoid creating reference cycles, preventing such cycles saves garbage collection time and limits the lifetime of objects. Moreover, some objects cannot be collected by the garbage collector.

Reference cycles can be visualized with the help of graphviz.

Classes

class `pympler.garbagegraph.`**`GarbageGraph`**(*reduce=False*, *collectable=True*)

The `GarbageGraph` is a `ReferenceGraph` that illustrates the objects building reference cycles. The garbage collector is switched to debug mode (all identified garbage is stored in *gc.garbage*) and the garbage collector is invoked. The collected objects are then illustrated in a directed graph.

Large graphs can be reduced to the actual cycles by passing `reduce=True` to the constructor.

It is recommended to disable the garbage collector when using the `GarbageGraph`.

```
>>> from pympler.garbagegraph import GarbageGraph, start_debug_garbage
>>> start_debug_garbage()
>>> l = []
>>> l.append(l)
>>> del l
>>> gb = GarbageGraph()
>>> gb.render('garbage.eps')
True
```

`__init__`(*reduce=False*, *collectable=True*)

Initialize the GarbageGraph with the objects identified by the garbage collector. If *collectable* is true, every reference cycle is recorded. Otherwise only uncollectable objects are reported.

`render`(*filename*, *cmd='dot'*, *format='ps'*, *unflatten=False*)

Render the graph to *filename* using graphviz. The graphviz invocation command may be overridden by specifying *cmd*. The *format* may be any specifier recognized by the graph renderer ('-Txxx' command). The graph can be preprocessed by the *unflatten* tool if the *unflatten* parameter is True. If there are no objects to illustrate, the method does not invoke graphviz and returns False. If the renderer returns successfully (return code 0), True is returned.

An *OSError* is raised if the graphviz tool cannot be found.

`split`()

Split the graph into sub-graphs. Only connected objects belong to the same graph. *split* yields copies of the Graph object. Shallow copies are used that only replicate the meta-information, but share the same object list `self.objects`.

```
>>> from pympler.refgraph import ReferenceGraph
>>> a = 42
>>> b = 'spam'
>>> c = {a: b}
>>> t = (1,2,3)
>>> rg = ReferenceGraph([a,b,c,t])
>>> for subgraph in rg.split():
...   print (subgraph.index)
0
1
```

write_graph(*filename*)
Write raw graph data which can be post-processed using graphviz.

print_stats(*stream=None*)
Log annotated garbage objects to console or file.

Parameters **stream** – open file, uses sys.stdout if not given

Functions

pympler.garbagegraph.**start_debug_garbage**()
Turn off garbage collector to analyze *collectable* reference cycles.

pympler.garbagegraph.**end_debug_garbage**()
Turn garbage collection on and disable debug output.

pympler.muppy

Functions

pympler.muppy.**get_objects**(*remove_dups=True*, *include_frames=False*)
Return a list of all known objects excluding frame objects.

If (outer) frame objects shall be included, pass *include_frames=True*. In order to prevent building reference cycles, the current frame object (of the caller of get_objects) is ignored. This will not prevent creating reference cycles if the object list is passed up the call-stack. Therefore, frame objects are not included by default.

Keyword arguments: remove_dups – if True, all duplicate objects will be removed. include_frames – if True, includes frame objects.

pympler.muppy.**get_size**(*objects*)
Compute the total size of all elements in objects.

pympler.muppy.**get_diff**(*left*, *right*)
Get the difference of both lists.

The result will be a dict with this form {'+': [], '-': []}. Items listed in '+' exist only in the right list, items listed in '-' exist only in the left list.

pympler.muppy.**sort**(*objects*)
Sort objects by size in bytes.

pympler.muppy.**filter**(*objects*, *Type=None*, *min=-1*, *max=-1*)
Filter objects.

The filter can be by type, minimum size, and/or maximum size.

Keyword arguments: Type – object type to filter by min – minimum object size max – maximum object size

pympler.muppy.**get_referents**(*object*, *level=1*)
Get all referents of an object up to a certain level.

The referents will not be returned in a specific order and will not contain duplicate objects. Duplicate objects will be removed.

Keyword arguments: level – level of indirection to which referents considered.

This function is recursive.

pympler.process

This module queries process memory allocation metrics from the operating system. It provides a platform independent layer to get the amount of virtual and physical memory allocated to the Python process.

Different mechanisms are implemented: Either the process stat file is read (Linux), the *ps* command is executed (BSD/OSX/Solaris) or the resource module is queried (Unix fallback). On Windows try to use the win32 module if available. If all fails, return 0 for each attribute.

Windows without the win32 module is not supported.

```
>>> from pympler.process import ProcessMemoryInfo
>>> pmi = ProcessMemoryInfo()
>>> print ("Virtual size [Byte]: " + str(pmi.vsz)) # doctest: +ELLIPSIS
Virtual size [Byte]: ...
```

Classes

class pympler.process.**_ProcessMemoryInfo**

Stores information about various process-level memory metrics. The virtual size is stored in attribute *vsz*, the physical memory allocated to the process in *rss*, and the number of (major) pagefaults in *pagefaults*. On Linux, *data_segment*, *code_segment*, *shared_segment* and *stack_segment* contain the number of Bytes allocated for the respective segments. This is an abstract base class which needs to be overridden by operating system specific implementations. This is done when importing the module.

update()

Refresh the information using platform instruments. Returns true if this operation yields useful values on the current platform.

pympler.process.**is_available**()

Convenience function to check if the current platform is supported by this module.

pympler.refbrowser

Tree-like exploration of object referrers.

This module provides a base implementation for tree-like referrers browsing. The two non-interactive classes ConsoleBrowser and FileBrowser output a tree to the console or a file. One graphical user interface for referrers browsing is provided as well. Further types can be subclassed.

All types share a similar initialisation. That is, you provide a root object and may specify further settings such as the initial depth of the tree or an output function. Afterwards you can print the tree which will be arranged based on your previous settings.

The interactive browser is based on a TreeWidget implemented in IDLE. It is available only if you have Tcl/Tk installed. If you try to instantiate the interactive browser without having Tkinter installed, an ImportError will be raised.

Classes

class pympler.refbrowser.**RefBrowser**(*rootobject*, *maxdepth=3*, *str_func=<function _repr>*, *repeat=True*, *stream=None*)

Base class to other RefBrowser implementations.

This base class provides means to extract a tree from a given root object and holds information on already known objects (to avoid repetition if requested).

get_tree()
Get a tree of referrers of the root object.

class pympler.refbrowser.**ConsoleBrowser**(**args*, ***kwargs*)
RefBrowser that prints to the console (stdout).

print_tree(*tree=None*)
Print referrers tree to console.

keyword arguments tree – if not None, the passed tree will be printed. Otherwise it is based on the rootobject.

class pympler.refbrowser.**FileBrowser**(*rootobject*, *maxdepth=3*, *str_func=<function _repr>*, *repeat=True*, *stream=None*)
RefBrowser implementation which prints the tree to a file.

print_tree(*filename*, *tree=None*)
Print referrers tree to file (in text format).

keyword arguments tree – if not None, the passed tree will be printed.

class pympler.refbrowser.**InteractiveBrowser**(*rootobject*, *maxdepth=3*, *str_func=<function gui_default_str_function>*, *repeat=True*)
Interactive referrers browser.

The interactive browser is based on a TreeWidget implemented in IDLE. It is available only if you have Tcl/Tk installed. If you try to instantiate the interactive browser without having Tkinter installed, an ImportError will be raised.

main(*standalone=False*)
Create interactive browser window.

keyword arguments standalone – Set to true, if the browser is not attached to other windows

pympler.refgraph

This module exposes utilities to illustrate objects and their references as (directed) graphs. The current implementation requires 'graphviz' to be installed.

Classes

class pympler.refgraph.**ReferenceGraph**(*objects*, *reduce=False*)
The ReferenceGraph illustrates the references between a collection of objects by rendering a directed graph. That requires that 'graphviz' is installed.

```
>>> from pympler.refgraph import ReferenceGraph
>>> a = 42
>>> b = 'spam'
>>> c = {a: b}
>>> gb = ReferenceGraph([a,b,c])
>>> gb.render('spam.eps')
True
```

__init__(*objects*, *reduce=False*)
Initialize the ReferenceGraph with a collection of *objects*.

render (*filename*, *cmd='dot'*, *format='ps'*, *unflatten=False*)

Render the graph to *filename* using graphviz. The graphviz invocation command may be overridden by specifying *cmd*. The *format* may be any specifier recognized by the graph renderer ('-Txxx' command). The graph can be preprocessed by the *unflatten* tool if the *unflatten* parameter is True. If there are no objects to illustrate, the method does not invoke graphviz and returns False. If the renderer returns successfully (return code 0), True is returned.

An *OSError* is raised if the graphviz tool cannot be found.

split ()

Split the graph into sub-graphs. Only connected objects belong to the same graph. *split* yields copies of the Graph object. Shallow copies are used that only replicate the meta-information, but share the same object list `self.objects`.

```
>>> from pympler.refgraph import ReferenceGraph
>>> a = 42
>>> b = 'spam'
>>> c = {a: b}
>>> t = (1,2,3)
>>> rg = ReferenceGraph([a,b,c,t])
>>> for subgraph in rg.split():
...   print (subgraph.index)
0
1
```

write_graph (*filename*)

Write raw graph data which can be post-processed using graphviz.

pympler.summary

A collection of functions to summarize object information.

This module provides several function which will help you to analyze object information which was gathered. Often it is sufficient to work with aggregated data instead of handling the entire set of existing objects. For example can a memory leak identified simple based on the number and size of existing objects.

A summary contains information about objects in a table-like manner. Technically, it is a list of lists. Each of these lists represents a row, whereas the first column reflects the object type, the second column the number of objects, and the third column the size of all these objects. This allows a simple table-like output like the following:

types	# objects	total size
<type 'dict'>	2	560
<type 'str'>	3	126
<type 'int'>	4	96
<type 'long'>	2	66
<type 'list'>	1	40

Another advantage of summaries is that they influence the system you analyze only to a minimum. Working with references to existing objects will keep these objects alive. Most of the times this is no desired behavior (as it will have an impact on the observations). Using summaries reduces this effect greatly.

output representation

The output representation of types is defined in summary.representations. Every type defined in this dictionary will be represented as specified. Each definition has a list of different representations. The later a representation appears in this list, the higher its verbosity level. From types which are not defined in summary.representations the default str() representation will be used.

Per default, summaries will use the verbosity level 1 for any encountered type. The reason is that several computations are done with summaries and rows have to remain comparable. Therefore information which reflect an objects state, e.g. the current line number of a frame, should not be included. You may add more detailed information at higher verbosity levels than 1.

functions

`pympler.summary.`**`summarize`**(*objects*)

Summarize an objects list.

Return a list of lists, whereas each row consists of:: [str(type), number of objects of this type, total size of these objects].

No guarantee regarding the order is given.

`pympler.summary.`**`get_diff`**(*left*, *right*)

Get the difference of two summaries.

Subtracts the values of the right summary from the values of the left summary. If similar rows appear on both sides, the are included in the summary with 0 for number of elements and total size. If the number of elements of a row of the diff is 0, but the total size is not, it means that objects likely have changed, but not there number, thus resulting in a changed size.

`pympler.summary.`**`print_`**(*rows*, *limit=15*, *sort='size'*, *order='descending'*)

Print the rows as a summary.

Keyword arguments: limit – the maximum number of elements to be listed sort – sort elements by 'size', 'type', or '#' order – sort 'ascending' or 'descending'

pympler.tracker

The tracker module allows you to track changes in the memory usage over time.

Using the SummaryTracker, you can create summaries and compare them with each other. Stored summaries can be ignored during comparison, avoiding the observer effect.

The ObjectTracker allows to monitor object creation. You create objects from one time and compare with objects from an earlier time.

Classes

`class` `pympler.tracker.`**`SummaryTracker`**(*ignore_self=True*)

Helper class to track changes between two summaries taken.

Detailed information on single objects will be lost, e.g. object size or object id. But often summaries are sufficient to monitor the memory usage over the lifetime of an application.

On initialisation, a first summary is taken. Every time *diff* is called, a new summary will be created. Thus, a diff between the new and the last summary can be extracted.

Be aware that filtering out previous summaries is time-intensive. You should therefore restrict yourself to the number of summaries you really need.

diff(*summary1=None*, *summary2=None*)
Compute diff between to summaries.

If no summary is provided, the diff from the last to the current summary is used. If summary1 is provided the diff from summary1 to the current summary is used. If summary1 and summary2 are provided, the diff between these two is used.

print_diff(*summary1=None*, *summary2=None*)
Compute diff between to summaries and print it.

If no summary is provided, the diff from the last to the current summary is used. If summary1 is provided the diff from summary1 to the current summary is used. If summary1 and summary2 are provided, the diff between these two is used.

store_summary(*key*)
Store a current summary in self.summaries.

class pympler.tracker.**ObjectTracker**
Helper class to track changes in the set of existing objects.

Each time you invoke a diff with this tracker, the objects which existed during the last invocation are compared with the objects which exist during the current invocation.

Please note that in order to do so, strong references to all objects will be stored. This means that none of these objects can be garbage collected. A use case for the ObjectTracker is the monitoring of a state which should be stable, but you see new objects being created nevertheless. With the ObjectTracker you can identify these new objects.

get_diff(*ignore=()*)
Get the diff to the last time the state of objects was measured.

keyword arguments ignore – list of objects to ignore

print_diff(*ignore=()*)
Print the diff to the last time the state of objects was measured.

keyword arguments ignore – list of objects to ignore

pympler.web

This module provides a web-based memory profiling interface. The Pympler web frontend exposes process information, tracker statistics, and garbage graphs. The web frontend uses Bottle, a lightweight Python web framework. Bottle is packaged with Pympler.

The web server can be invoked almost as easily as setting a breakpoint using *pdb*:

```
from pympler.web import start_profiler
start_profiler()
```

Calling `start_profiler` suspends the current thread and executes the Pympler web server, exposing profiling data and various facilities of the Pympler library via a graphic interface.

Functions

pympler.web.**start_profiler**(*host='localhost'*, *port=8090*, *tracker=None*, *stats=None*, *debug=False*, ***kwargs*)
Start the web server to show profiling data. The function suspends the Python application (the current thread) until the web server is stopped.

The only way to stop the server is to signal the running thread, e.g. press Ctrl+C in the console. If this isn't feasible for your application use *start_in_background* instead.

During the execution of the web server, profiling data is (lazily) cached to improve performance. For example, garbage graphs are rendered when the garbage profiling data is requested and are simply retransmitted upon later requests.

The web server can display profiling data from previously taken snapshots when *tracker* or *stats* is specified. The former is useful for profiling a running application, the latter for off-line analysis. Requires existing snapshots taken with `create_snapshot()` or `start_periodic_snapshots()`.

Parameters

- **`host`** – the host where the server shall run, default is localhost
- **`port`** – server listens on the specified port, default is 8090 to allow coexistance with common web applications
- **`tracker`** – *ClassTracker* instance, browse profiling data (on-line analysis)
- **`stats`** – *Stats* instance, analyze *ClassTracker* profiling dumps (useful for off-line analysis)

`pympler.web.`**`start_in_background`**`(`***kwargs*`)`

Start the web server in the background. A new thread is created which serves the profiling interface without suspending the current application.

For the documentation of the parameters see *start_profiler*.

Returns the created thread object.

7.1.11 Pympler Tutorials

- *Tutorial - Track Down Memory Leaks* - This tutorial shows you ways in which *muppy* can be used to track down memory leaks.
- *Tutorial - Tracking Class Instances in SCons* - A tutorial illustrating how to use the *ClassTracker* facility.

Table of Content

Tutorial - Track Down Memory Leaks

This tutorial shows you ways in which *muppy* can be used to track down memory leaks. From my experience, this can be done in 3 steps, each answering a different question.

1. Is there a leak?
2. What objects leak?
3. Where does it leak?

IDLE

My first real-life test for *muppy* was IDLE, which is "the Python IDE built with the Tkinter GUI toolkit." It offers the following features:

- coded in 100% pure Python, using the Tkinter GUI toolkit
- cross-platform: works on Windows and Unix (on Mac OS, there are currently problems with Tcl/Tk)

- multi-window text editor with multiple undo, Python colorizing and many other features, e.g. smart indent and call tips
- Python shell window (a.k.a. interactive interpreter)
- debugger (not complete, but you can set breakpoints, view and step)

Because it is integrated in every Python distribution, runs locally and provides easy interactive feedback, it was a nice first candidate to test the tools of muppy.

The task was to check if IDLE leaks memory, if so, what objects are leaking, and finally, why are they leaking.

Preparations

IDLE is part of every Python distribution and can be found at `Lib/idlelib`. The modified version which makes use of muppy can be found at http://code.google.com/p/muppy/source/browse/trunk#trunk/playground/idlelib.

With IDLE having a GUI, I also wanted to be able to interact with muppy through the GUI. This can be done in `Lib/idlelib/Bindings.py` and `Lib/idlelib/PyShell.py`. For details, please refer to the modified version mentioned above.

Task 1: Is there a leak?

At first, we need to find out if there are any objects leaking at all. We will have a look at the objects, invoke an action, and look at the objects again.

```
from pympler import tracker

self.memory_tracker = tracker.SummaryTracker()
self.memory_tracker.print_diff()
```

The last step is repeated after each invocation. Let's start with something simple which should not leak. We will check the Windows resize. You can invoke it in the menu at *Windows->Zoom Height*.

At first call *print_diff* till it has calibrated. That is, the first one or two times, you will get some output because there is still something going on in the background. But then you should get this:

```
types |   # objects |   total size
====== | =========== | ============
```

Which means nothing has changed since the last invocation of *print_diff*. Now let's call *Windows->Zoom Height* and invoke *print_diff* again.:

```
             types |   # objects |   total size
================== | =========== | ============
              dict |           1 |      280     B
              list |           1 |      176     B
  _sre.SRE_Pattern |           1 |       88     B
             tuple |           1 |       80     B
               str |           0 |        7     B
```

Seems as this requires some of the above mentioned objects. Let's repeat it.:

```
 types |   # objects |   total size
====== | =========== | ============
```

Okay, nothing changed, so nothing is leaking. But we see that often, the first call to a function creates some objects, which then exist on a second invocation.

Next, we try something different. We will open a new window. Let's have a look at the Path Browser at *File->Path Browser*.:

```
                                                types |   # objects |   total size
===================================================== | =========== | ============
                                                 dict |          18 |     14.26 KB
                                                tuple |         146 |     13.17 KB
                                                 list |           2 |     11.67 KB
                                                  str |          97 |      7.85 KB
                                                 code |          46 |      5.52 KB
                                             function |          45 |      5.40 KB
                                             classobj |           9 |    864     B
                    instancemethod (<function wakeup>) |           3 |    240     B
                  instancemethod (<function __call__>) |           3 |    240     B
                 instance(<class Tkinter.CallWrapper>) |           3 |    216     B
                                               module |           3 |    168     B
  instance(<class idlelib.WindowList.ListedToplevel>) |           1 |     72     B
```

Let's repeat it.:

```
                                                types |   # objects |   total size
===================================================== | =========== | ============
                                                 dict |           5 |      2.17 KB
                                                 list |           0 |    384     B
                                                  str |           5 |    259     B
                    instancemethod (<function wakeup>) |           3 |    240     B
                  instancemethod (<function __call__>) |           3 |    240     B
                 instance(<class Tkinter.CallWrapper>) |           3 |    216     B
  instance(<class idlelib.WindowList.ListedToplevel>) |           1 |     72     B
```

Mh, still some new objects. Repeating this procedure several times will reveal that here indeed we have a leak.

Task 2: What objects leak?

So let's have a closer look at the diff. We see 5 new *dicts* and *strings*, a bit more memory usage by *lists*, 3 *wakeup* and *__call__* instance methods, 3 *CallWrapper* and 1 *ListedToplevel*. We know the standard types, but the last couple of objects seem to be from IDLE.

We ignore the standard type objects for now. It is more likely that these are only children of some other instances which are causing the leak.

We start with the *ListedTopLevel* object. One invocation of *File->Path Browser* and one more of this type looks like this object is not garbage collected, although it should have been. Searching for *ListedTopLevel* in *idlelib/* reveals that is the base class to all window objects of IDLE. We can assume that opening the Path Browser, a new window object is created, but closing the window does not remove the reference.

Next, we take a look at the *wakeup* instance method of which we have three more on each invocation. Searching the code, we find it to be defined in *idlelib/WindowList.py*. This piece of code is used to give users of IDLE a list of currently open windows. Every time a new window is created, it will be added to the *Windows* menu, from where the user can select any open window. *wakeup* is the method which will bring the selected window up front. Adding a window calls menu.add_command, linking menu and the wakeup command together.

```
menu.add_command(label=title, command=window.wakeup)
```

So we are getting closer. Only *__call__* and *Tkinter.CallWrapper* are left. As the name indicates, the latter is located in the Tkinter module, which is part of the standard library of Python. So let's dive into it. The CallWrapper docstring states:

```
Internal class. Stores function to call when some user defined Tcl function is
called e.g. after an event occurred.
```

Also, CallWrapper contains a method called *__call__*, which is used to invoke the stored function call. A CallWrapper is created by the method *_register* which then creates a command (Tk speak) and adds it's name to a list called *self._tclCommands*.

So what do we know so far? Every time a Path Browser is opened, a window is created, but not deleted when closed again. It has something to do with the *wakeup* method of the window. This method is wrapped as a Tcl command and then linked to the window list menu. Also, we have traced this wrapping back to Tkinter library, where names of the function wrappers are stored in an attribute called *_tclCommands*.

This brings us to the third question.

Task 3: Where is the leak?

_tclCommands stores the names of all commands linked to a widget. The base class for interior widgets (of which the menu is one), has a method called *destroy* which:

```
Delete all Tcl commands created for this widget in the Tcl
interpreter.
```

as well as a method *deletecommand* which deletes a single Tcl command. Both remove commands as by there name. Among them, we find our CallWrappers' *__call__* used to wrap the wakeup of the Path Browser window.

So we should expect at least either one to be invoked when a window is closed (best would be the invocation of only deletecommand). This would also go in line with *menu.add_command* we identified *above*. And indeed, in *idlelib/EditorWindow.py*, *menu.delete* is called. So where is the problem?

We return to *Tkinter.py* and have a closer look at *delete* method:

```
def delete(self, index1, index2=None):
    """Delete menu items between INDEX1 and INDEX2 (not included)."""
    self.tk.call(self._w, 'delete', index1, index2)
```

Mh, it looks like the menu item is deleted, but what about the attached command? Let's ask the Web for "tkinter deletecommand". Turns out that somebody some years ago filed a bug (see bugreport) which states:

```
Tkinter.Menu.delete does not delete the commands
defined for the entries it deletes. Those objects
will be retained until the menu itself is deleted.
[..]
the command function will still be referenced and
kept in memory - until the menu object itself is
destroyed.
```

Well, this seems to be the root of our memory leak. Let's adapt the *delete* method a bit, so that the associated commands are deleted as well:

```
def delete(self, index1, index2=None):
    """Delete menu items between INDEX1 and INDEX2 (not included)."""
    if index2 is None:
        index2 = index1
    cmds = []
    (num_index1, num_index2) = (self.index(index1), self.index(index2))
    if (num_index1 is not None) and (num_index2 is not None):
        for i in range(num_index1, num_index2 + 1):
            if 'command' in self.entryconfig(i):
                c = str(self.entrycget(i, 'command'))
                if c in self._tclCommands:
                    cmds.append(c)
    self.tk.call(self._w, 'delete', index1, index2)
    for c in cmds:
        self.deletecommand(c)
```

Now we restart IDLE, calibrate our tracker and do another round of *print_diff*. After the first time the Path Browser is opened we get this:

```
     types |   # objects |   total size
========== | =========== | ============
     tuple |         146 |     13.17 KB
      dict |          13 |     12.01 KB
      list |           2 |     11.26 KB
       str |          92 |      7.59 KB
      code |          46 |      5.52 KB
  function |          45 |      5.40 KB
  classobj |           9 |    864     B
    module |           3 |    168     B
```

Okay, still some objects created, but no more instances and instance methods. Let's do it again.:

```
  types |   # objects |   total size
======= | =========== | ============
```

Yes, this looks definitely better. The memory leak is gone.

The problem is fixed for Python versions 2.5 and higher so updated installations will not face this leak.

Tutorial - Tracking Class Instances in SCons

This tutorial demonstrates the class tracking facility to profile and optimize a non-trivial program. SCons is a next-generation build system with a quite elaborate architecture and thus an interesting candidate for profiling attempts.

Before we begin, it should be identified what shall be tracked, i.e. what classes we want to connect to and whose instances are to be sized and profiled. In this tutorial, the effect of a patch is analyzed that tries to size-optimize the very heart of SCons - the `Node` class. Naturally, we will connect to the `Node` base class and its sub-classes. It makes sense to put the profiling data in context and track additional classes that are believed to contribute significantly to the total memory consumption.

Installing hooks into SCons

The first step is to find the proper spots for connecting to the classes that shall be tracked, taking snapshots, and printing the gathered profile data. SCons has a simple memory profiling tool that we will override. The SCons MemStats class provides all we need:

```
from pympler.classtracker import ClassTracker

class MemStats(Stats):
    def __init__(self):
        Stats.__init__(self)
        classes = [SCons.Node.Node, SCons.Node.FS.Base, SCons.Node.FS.File,
                   SCons.Node.FS.Dir, SCons.Executor.Executor]
        self.tracker = ClassTracker()
        for c in classes:
            self.tracker.track_class(c)
    def do_append(self, label):
        self.tracker.create_snapshot(label)
    def do_print(self):
        stats = self.tracker.stats
        stats.print_summary()
        stats.dump_stats('pympler.stats')
```

When SCons starts, `MemStats` is instantiated and the *ClassTracker* is connected to a number of classes. SCons has predefined spots where it invokes its statistics facilities with `do_append` being called. This is where snapshosts will be taken of all objects tracked so far.

Because of the large number of instances, only a summary is printed to the console via `stats.print_summary()` and the profile data is dumped to a file in case per-instance profile information is needed later.

Test run

Time for a test. In the following examples, SCons builds a non-trivial program with a fair number of nodes. Running SCons via `scons --debug=memory` will print the gathered data to the console:

```
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: `.' is up to date.
scons: done building targets.
---- SUMMARY ------------------------------------------------------------------
before reading SConscript files:          active      4.17 MB      average   pct
  SCons.Executor.Executor                      7      7.53 KB      1.08 KB    0%
  SCons.Node.FS.Base                           1      9.30 KB      9.30 KB    0%
  SCons.Node.FS.Dir                            6     17.77 KB      2.96 KB    0%
  SCons.Node.FS.File                           1      2.91 KB      2.91 KB    0%
  SCons.Node.Node                              0      0     B      0     B    0%
after reading SConscript files:           active     13.06 MB      average   pct
  [...]
before building targets:                  active     13.41 MB      average   pct
  [...]
after building targets:                   active     34.77 MB      average   pct
  SCons.Executor.Executor                   1311      3.57 MB      2.79 KB   10%
  SCons.Node.FS.Base                        1102      4.84 MB      4.50 KB   13%
  SCons.Node.FS.Dir                          108      5.67 MB     53.72 KB   16%
  SCons.Node.FS.File                        2302     10.45 MB      4.65 KB   30%
  SCons.Node.Node                              1     84.93 KB     84.93 KB    0%
-------------------------------------------------------------------------------
```

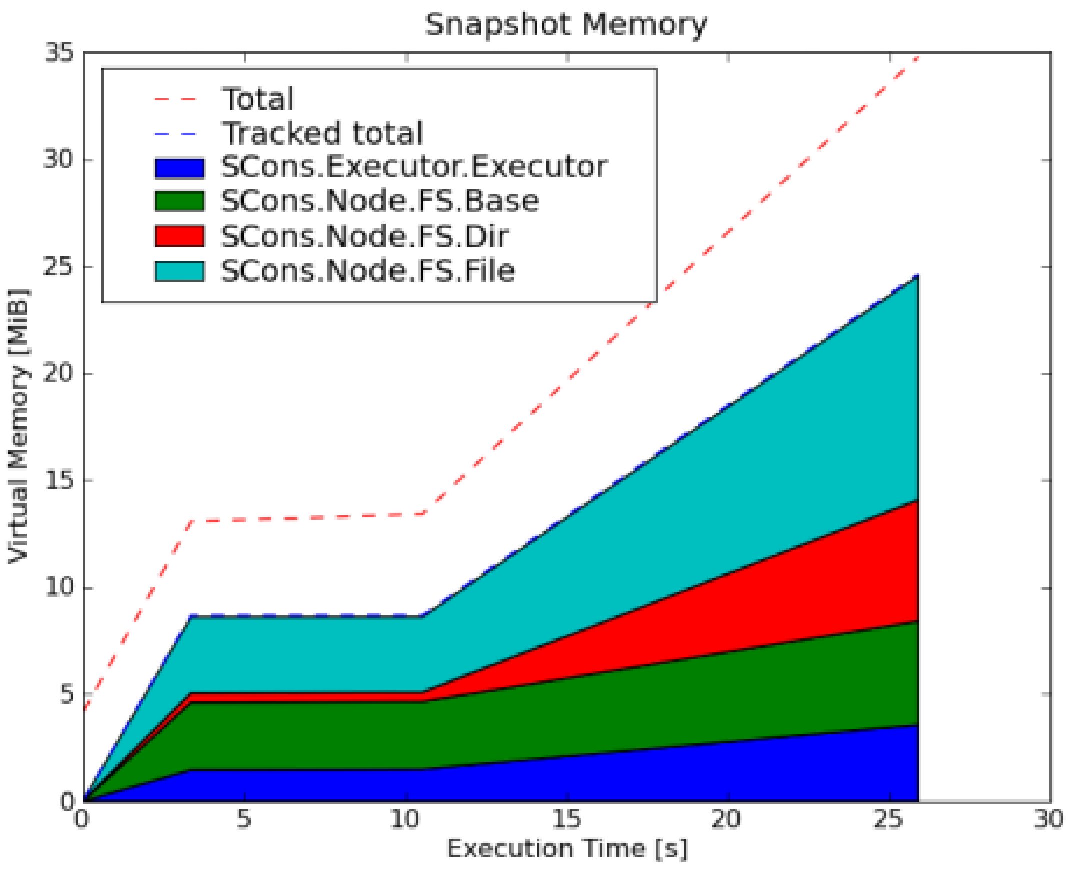
Making sense of the data

The console output may give a brief overview how much memory is allocated by instances of the individual tracked classes. A more appealing and well arranged representation of the data can be generated with the `HtmlStats` class. The dump generated previously can be loaded and a set of HTML pages can be emitted:

```python
from pympler.classtracker_stats import HtmlStats

stats = HtmlStats()
stats.load_stats('pympler.stats')
stats.create_html('pympler.html')
```

If `matplotlib` is installed, charts will be embedded in the HTML output:



At first sight it might seem suspicious that the tracked classes appear to be the sole contributors to the total memory footprint of the application. Because the tracked objects are sized recursively, referenced objects which are not tracked themselves are added to the referrers account. Thus, a root object's size will include the size of every leaf unless the leaf is also tracked by the *ClassTracker*.

Optimization attempt

After applying the patch by Jean Brouwers, SCons is rerun under the supervision of the *ClassTracker*. The differences in the last snapshot show that the changes indeed reduce the memory footprint of `Node` instances:

```
$ scons --debug=memory
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: `.' is up to date.
scons: done building targets.
---- SUMMARY ----------------------------------------------------------------
[...]
after building targets:                 active     32.41 MB      average   pct
  SCons.Executor.Executor                 1311      3.50 MB      2.73 KB   10%
  SCons.Node.FS.Base                      1102      4.29 MB      3.98 KB   13%
  SCons.Node.FS.Dir                        108      5.52 MB     52.30 KB   17%
  SCons.Node.FS.File                      2302      8.82 MB      3.92 KB   27%
  SCons.Node.Node                            1     84.32 KB     84.32 KB    0%
-----------------------------------------------------------------------------
```

The total measured memory footprint dropped from 34.8MB to 32.4MB, `File` nodes' average size from 4.6KB to 3.9KB.

Summary

This tutorial illustrated how applications can be profiled with the *ClassTracker* facility. It has been shown how the memory impact of changes can be quantified.

7.1.12 Related Work

Pympler is a merger of several approaches toward memory profiling of Python applications. This page lists other known tools. If you know yet another one or find the description is not correct you can create a new issue at http://code.google.com/p/pympler/issues.

asizeof

Asizeof is a pure-Python module to estimate the size of objects by Jean Brouwers. This implementation has been published previously on aspn.activestate.com. It is possible to determine the size of an object and its referents recursively up to a specified level. asizeof is also distributed with muppy and allows the usage of muppy with Python versions prior to Python 2.6.

asizeof has become a part of Pympler.

URL: http://code.activestate.com/recipes/546530/

Heapmonitor

"The Heapmonitor is a facility delivering insight into the memory distribution of SCons. It provides facilities to size individual objects and can track all objects of certain classes." It was developed in 2008 by Ludwig Haehne.

URL: http://www.scons.org/wiki/LudwigHaehne/HeapMonitor

Heapmonitor has become a part of Pympler.

Heapy

Heapy was part of the Master thesis by Sverker Nilsson done in 2006. It is part of the umbrella project guppy. Heapy has a very mathematical approach as it works in terms of sets, partitions, and equivalence relations. It allows to gather information about objects at any given time, but only objects starting from a specific root object. Type information for standard objects is supported by default and type information for non-standard object types can be added through an interface.

URL: http://guppy-pe.sourceforge.net

Meliae

"This project is similar to heapy (in the 'guppy' project), in its attempt to understand how memory has been allocated.

Currently, its main difference is that it splits the task of computing summary statistics, etc of memory consumption from the actual scanning of memory consumption. It does this, because I often want to figure out what is going on in my process, while my process is consuming huge amounts of memory (1GB, etc). It also allows dramatically simplifying the scanner, as I don't allocate python objects while trying to analyze python object memory consumption."

Meliae is being developed by John A Meinel since 2009. It is well suited for offline analysis of full memory dumps.

URL: https://launchpad.net/meliae

muppy

"Muppy [..] enables the tracking of memory usage during runtime and the identification of objects which are leaking. Additionally, tools are provided which allow to locate the source of not released objects." It was developed in 2008 by Robert Schuppenies.

muppy has become a part of Pympler.

Python Memory Validator

A commercial Python memory validator which uses the Python Reflection API.

URL: http://www.softwareverify.com/python/memory/index.html

PySizer

PySizer was a Google Summer of Code 2005 project by Nick Smallbone. It relies on the garbage collector to gather information about existing objects. The developer can create a summary of the current set of objects and then analyze the extracted data. It is possible to group objects by criteria like object type and apply filtering mechanisms to the sets of objects. Using a patched CPython version it is also possible to find out where in the code a certain object was created. Nick points out that "the interface is quite sparse, and some things are clunky". The project is deprecated and the last supported Python version is 2.4.

URL: http://pysizer.8325.org/

Support Tracking Low-Level Memory Usage in CPython

This is an experimental implementation of CPython-level memory tracking by Brett Cannon. Done in 2006, it tackles the problem at the core, the CPython interpreter itself. To trace the memory usage he suggests to tag every memory allocation and de-allocation. All actions involving memory take a *const char ** argument that specifies what the memory is meant for. Thus every allocation and freeing of memory is explicitly registered. On the Python level the total memory usage as well as "a dict with keys as the string names of the types being tracked and values of the amount of memory being used by the type" are available.

URL: http://svn.python.org/projects/python/branches/bcannon-sandboxing/PEP.txt

7.1.13 Glossary

`asizeof` Name of a formerly separate package. Now integrated into Pympler.

`HeapMonitor` Name of a formerly separate package. Now integrated into Pympler.

`muppy` Name of a formerly separate package. Now integrated into Pympler.

`pymple, to` Obtain detailed insight in the size and the lifetime of Python objects.

`pymple, a` Undesirable or unexpected runtime behavior like memory bloat.

`summary` A summary contains information about objects in a summarized format. Instead of having data of every object, information are grouped by object type. Each object type is represented by a row, whereas the first column reflects the object type, the second column the number of objects of this type, and the third column the size of all of these objects. The output looks like the following:

types	# objects	total size
<type 'dict'>	2	560
<type 'str'>	3	126
<type 'int'>	4	96
<type 'long'>	2	66
<type 'list'>	1	40

7.1.14 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

0.9 - 2020-10-14

Added

- Python 3.9 support – By tirkarthi (#105)
- Compatibility with Django 3.x – By Lance Moore (#108)

Removed

- Python 3.4 support

Fixed

- Include size of data when sizing Numpy slices – Rported by sinorga (#111), fixed by Jean Brouwers
- Fix KeyError when sizing dicts in certain scenarios – Reported by MrSanZhi (#114), fixed by Jean Brouwers

0.8 - 2019-11-12

Added

- Python 3.8 support
- Compatibility with Django Debug Toolbar 2.x – Reported by John Carter (#96)

Removed

- Python 3.3 support
- Compatibility with Django Debug Toolbar 1.x

Fixed

- Include dicts which aren't tracked by garbage collector in summary diff – Reported by Dave Johansen (#97)
- Fix formatting of Python 3 class names in summary diff – Reported by laundmo (#98)

0.7 - 2019-04-05

Added

- Added `asizeof` options `above` and `cutoff` to specify minimal size and the number of large objects to be printed
- The `Asizer` class has a new property `ranked` returning the number of ranked objects.
- New `Asizer` method `exclude_objs` can be used to exclude objects from being sized, profiled and ranked.

Changed

- The `asizeof` option `stats` has been enhanced to include the list of the 100 largest objects, ranked by total size.

Fixed

- Fix TypeError raised in certain scenarios – Reported by James Hirschorn (#72), fixed by Jean Brouwers
- Fix TypeError when creating snapshots with classtracker in certain scenarios – Reported by rtadewald (#79), fixed by Jean Brouwers

0.6 - 2018-09-01

Added

- Python 3.7 support

Changed

- Update asizeof module to version 18.07.08. Includes more accurate sizing of objects with slots. – By Jean Brouwers

Removed

- Python 2.6 and 3.2 support

Fixed

- Fix KeyError when using Django memory panel in certain scenarios – Reported by Mark Davidoff (#55), fixed by Pedro Tacla Yamada
- Fix Debug Toolbar - Remove all jQuery variables from the global scope – By the5fire (#66)
- Fix process import error when empty lines found in /proc/self/status – Reported by dnlsng (#67)
- Return more accurate size of objects with slots – Reported by Ivo Anjo (#69), fixed by Jean Brouwers

0.5 - 2017-03-23

Added

- Add support for Python 3.5 and Python 3.6

Changed

- Improved runtime performance of summary differ – By Matt Perpick (#42)
- Include values when sizing named tuples – Reported by Paul Ellenbogen (#35), fixed by Chris Klaiber
- Update bottle.py to 0.12.13

Removed

- Drop Python 2.5 and Python 3.1 support

0.4.3 - 2016-03-31

Added

- Add Django 1.9 support for DDT panel – By Benjy (#30)

Fixed

- Handle untracked classes in tracker statistics – By gbtami (#33)
- Handle colons in process names gracefully – By Dariusz Suchojad (#26)
- Support types without `__flags__` attribute in muppy (#24)
- Fix documentation errors (#32, #28, #25) – By gbtami, Matt, Lawrence Hudson

0.4.2 - 2015-07-26

Fixed

- Include private variables within slots when sizing recursively – GitHub issue #20 report and fix by ddodt
- Fix NameError in memory panel – GitHub issue #21 reported by relekang

0.4.1 - 2015-04-15

Changed

- Replace Highcharts with Flot (#17)

0.4 - 2015-02-03

Added

- Added memory panel for django-debug-toolbar
- Format tracker statistics without printing – GitHub issue #2 reported and implemented by Andrei Sosnin
- Added close method to ClassTracker
- Support for Python 3.4

Changed

- Track instance counts of tracked classes without snapshots
- Upgrade to Highcharts 3 and jQuery 1.10

Removed

- Dropped support for Python 2.4

Fixed

- Include size of closure variables – GitHub issue #8 reported and implemented by Craig Silverstein
- Fix tkinter import on Python 3 – GitHub issue #4 reported by pedru-de-huere

- Fix `StreamBrowser.print_tree` when called without arguments – GitHub issue #5 reported by pedru-de-huere
- Fix sizing of named tuples – GitHub issue #10 reported by ceridwen

0.3.1 - 2013-02-16

- Fix class tracker graph data formatting – Issue 48 reported by Berwyn Hoyt
- Improve web class tracker documentation – Issue 49 reported by Berwyn Hoyt
- Update links to GitHub and PyPi

0.3.0 - 2012-12-29

- Support for Python 3.3

0.2.2 - 2012-11-24

- Work around array sizing bug in Python 2.6-3.2 – Issue 46 reported by Matt
- Fix import when python is run with optimization `-OO` – Issue 47 reported by Kunal Parmar

0.2.1 - 2011-11-13

- Fix static file retrieval when installed via easy_install
- Show class tracker instantiation traces and referent trees in web interface
- New style for web interface

0.2

The second release is one of several steps to better integrate the different sub-systems of Pympler. All modules now directly reside in the pympler namespace which simplifies the import of Pympler modules. Pympler 0.2 introduces a web interface to facilitate memory profiling. Pympler now fully supports Python 3.x. This release also adds several modules replacing the *Heapmonitor* module with the new *class tracker* facility.

- Introduce web frontend
- Split Heapmonitor into several new modules
- New `process` module to obtain memory statistics of the Python process
- Improved garbage illustration which can directly render directed graphs using graphviz

0.1

This initial release is the first step to unify three separate Python memory profiling tools. We aim to create a place-to-go for Python developers who want to monitor and analyze the memory usage of their applications. It is just the first step towards a further integration. There is still lots of work that needs to be done and we stress that the API is subject to change. Any feedback you want to give us, wishes, bug reports, or feature requests please send them to **pympler-dev@googlegroups.com**.

7.1.15 Copyright

The Pympler software and sample code is licensed under the Apache License, Version 2.0. The asizeof module is licensed under a different license (see *asizeof license*):

```
                              Apache License
                        Version 2.0, January 2004
                     http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
```

(continues on next page)

(continued from previous page)

```
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
```

(continues on next page)

(continued from previous page)

```
        of the following places: within a NOTICE text file distributed
        as part of the Derivative Works; within the Source form or
        documentation, if provided along with the Derivative Works; or,
        within a display generated by the Derivative Works, if and
        wherever such third-party notices normally appear. The contents
        of the NOTICE file are for informational purposes only and
        do not modify the License. You may add Your own attribution
        notices within Derivative Works that You distribute, alongside
        or as an addendum to the NOTICE text from the Work, provided
        that such additional attribution notices cannot be construed
        as modifying the License.

    You may add Your own copyright statement to Your modifications and
    may provide additional or different license terms and conditions
    for use, reproduction, or distribution of Your modifications, or
    for any such Derivative Works as a whole, provided Your use,
    reproduction, and distribution of the Work otherwise complies with
    the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
```

(continues on next page)

(continued from previous page)

```
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.
```

asizeof license

The asizeof module is licensed under the BSD license.

```
      Copyright (c) 2002-2008 -- ProphICy Semiconductor, Inc.
                       All rights reserved.
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

- Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer in
  the documentation and/or other materials provided with the
  distribution.

- Neither the name of ProphICy Semiconductor, Inc. nor the names
  of its contributors may be used to endorse or promote products
  derived from this software without specific prior written
  permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

Python Module Index

Index