What's Eating My Memory?

Andrew Bennetts
andrew@bemusement.org
Bazaar team, Canonical

The problem

So you're worried about how much memory your program is using...

You take the usual steps...

1. Remove your __del__ methods

2. Sprinkle __slots__ everywhere

~~3. Avoid cycles~~

Ok __del__ and __slots__, but your program still consumes 2GB of memory?

Anatomy of a PyObject

```
typedef struct _object {
    PyObject_HEAD
} PyObject;

typedef struct {
    PyObject_VAR_HEAD
} PyVarObject;
```

```
/* PyObject_HEAD defines the initial segment
of every PyObject. */
#define PyObject_HEAD                           \
    _PyObject_HEAD_EXTRA                        \
    Py_ssize_t ob_refcnt;                       \
    struct _typeobject *ob_type;
```

```
#define PyObject_VAR_HEAD                       \
    PyObject_HEAD                               \
    Py_ssize_t ob_size; /* Number of items
                           in variable part */
```

What the heck does that mean?

Some Python object sizes:

(on 32-bit CPython 2.6)

type	size (bytes)
int	12
str	24
tuple	24 + len(t) * 4
dict, len < 4	132
dict, len < 22	1672
instance of “`class C(object): pass`”	32

How can you find these numbers?

Answer 1:

`sys.getsizeof(`*`thing`*`)`

Answer 2:

Just instantiate 100000 of something and see how much memory it takes!

An aside:

Memory stats on Linux

```
$ cat /proc/PID/status | grep Vm
VmPeak:     8484 kB
VmSize:     8484 kB
VmLck:         0 kB
VmHWM:      4372 kB
VmRSS:      4372 kB
VmData:     1680 kB
VmStk:       100 kB
VmExe:      1920 kB
VmLib:      3964 kB
VmPTE:        32 kB
```

Goodies built into Python

```
gc.get_referrers

gc.get_objects

sys.getsizeof

sys.getrefcount
```

These can give a good approximation of “which kinds of objects are most used?”, etc.

```
def mostRefs(n=30):
    d = {}
    for obj in gc.get_objects():
        if type(obj) is types.InstanceType:
            cls = obj.__class__
        else:
            cls = type(obj)
        d[cls] = d.get(cls, 0) + 1
    counts = [(x[1],x[0]) for x in d.items()]
    counts = sorted(counts)[-n:]
    return reversed(counts)

def printCounts(counts=None, file=None):
    if counts is None: counts = mostRefs()
    if file is None: file = sys.stdout
    for c, obj in counts:
        file.write("%s %s\n" % (c, obj))
```

Sadly, sys and gc modules have some blind spots

Besides, who wants to write this stuff from scratch?

Meliae

https://launchpad.net/meliae

Two parts: scanner and processor

scanner — dump memory details to disk

processor — analyse those dumps

Designed for use from Python prompt (or pdb):

```
> /home/andrew/code/bzr/bzrlib/inventory.py(1501)__init__()->None
-> self.root_id = None
(Pdb) from meliae import scanner
(Pdb) pp self
<bzrlib.inventory.CHKInventory object at 0xb0ce1ac>
(Pdb) scanner.get_recursive_size(self)
(854591, 47393254)
(Pdb) scanner.dump_all_objects('filename.json')
(Pdb)
```

```
>>> from meliae import loader
>>> om = loader.load('/tmp/bzr-branch.bzr-dev.meliae')
loaded line 903085, 903086 objs, 107.7 / 107.7 MiB read in 35.8s
>>> om.summarize()
Total 903086 objects, 249 types, Total size = 50.5MiB (52987209 bytes)
 Index   Count   %      Size   % Cum     Max Kind
     0  221788  24  18699847  35  35 1878706 str
     1    8790   0  13457328  25  60 1573000 dict
     2  226283  25   8097956  15  75     276 tuple
     3  292078  32   3504936   6  82      12 bzrlib._static_tuple_c.StaticTuple
     4     213   0   2926384   5  88 1048688 set
     5    1019   0   2631000   4  93 1339048 list
     6  120034  13   1440408   2  95      12 int
     7    1476   0    661248   1  97     448 type
     8    8268   0    562224   1  98      68 code
     9    8194   0    458864   0  98      56 function
    10    4988   0    139664   0  99      28 _LazyGroupCompressFactory
    11    1723   0     68920   0  99      40 weakref
    12    1163   0     41868   0  99      36 wrapper_descriptor
    13    1171   0     37472   0  99      32 builtin_function_or_method
    14     790   0     25280   0  99      32 getset_descriptor
    15      36   0     18240   0  99     592 StgDict
    16      42   0     16444   0  99     544 frame
    17     494   0     15808   0  99      32 method_descriptor
    18     299   0     13156   0  99      44 _LRUNode
    19     203   0     12532   0  99     348 unicode
>>>
```

```
>>> om[s.summaries[0].max_address]
str(3068231688 1878706B
'f\x92\x02chkinventory:\nsearch_key_name: hash-255-
way\nroot_id: TREE_ROOT\nparent_id_basename_to_file_id:
sha')
>>> om[s.summaries[1].max_address]
dict(172895468 1573000B 63308refs)
>>> om[s.summaries[1].max_address].p
[]
>>> om[s.summaries[1].max_address].c[:3]
[bzrlib._static_tuple_c.StaticTuple(181272096 12B 1refs),
bzrlib._static_tuple_c.StaticTuple(210597672 12B 2refs),
bzrlib._static_tuple_c.StaticTuple(174057584 12B 1refs)]
```

[Address, size, children, parents, content]

heapy

http://guppy-pe.sourceforge.net/#Heapy

Powerful... but poorly documented

```
>>> from guppy import hpy
>>> hp = hpy()
>>> h = hp.heap()
>>> h
Partition of a set of 1449133 objects. Total size = 102766644 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of
class)
     0 985931  68 46300932  45  46300932  45 str
     1  24681   2 22311624  22  68612556  67 dict of
pkgcore.ebuild.ebuild_src.package
     2  49391   3 21311864  21  89924420  88 dict (no owner)
     3 115974   8  3776948   4  93701368  91 tuple
     4 152181  11  3043616   3  96744984  94 long
     5  36009   2  1584396   2  98329380  96 weakref.KeyedRef
     6  11328   1  1540608   1  99869988  97 dict of
pkgcore.ebuild.ebuild_src.ThrowAwayNameSpace
     7  24702   2   889272   1 100759260  98 types.MethodType
     8  11424   1   851840   1 101611100  99 list
     9  24681   2   691068   1 102302168 100
pkgcore.ebuild.ebuild_src.package
<54 more rows. Type e.g. '_.more' to view.>
```

```
>>> h.get_rp(40)
Reference Pattern by <[dict of] class>.
 0: _ --- [-] 14 (dict (no owner) | list | str | types.FrameType | types.Gen...
 1: a      [-] 3 dict (no owner): 0x8c11f34*2, 0x8c1bd54*2, 0x8c1f854*2
 2: aa ---- [-] 1 list: 0x833c504*18
 3: a3       [-] 1 dict of django.db.backends.mysql.base.DatabaseWrapper: 0x...
 4: a4 ------ [-] 1 dict (no owner): 0x83a65d4*2
 5: a5         [R] 1 guppy.heapy.heapyc.RootStateType: 0xb787c7a8L
 6: a3b ----- [-] 1 django.db.backends.mysql.base.DatabaseWrapper: 0x8356a34
 7: a3ba       [S] 7 dict of module: ..db, ..models, ..query, ..transaction...
 8: b ---- [S] 1 types.FrameType: <<lambda> at 0x8b16ecc>
 9: c      [-] 2 list: 0x833c504*18, 0xb7dafe6cL*5
<Type e.g. '_.more' for more.>
```

Also has support for taking differences between heaps, sorting the data, grouping the data...

...and probably much more, if you can understand the manual.

Has had portability problems :(

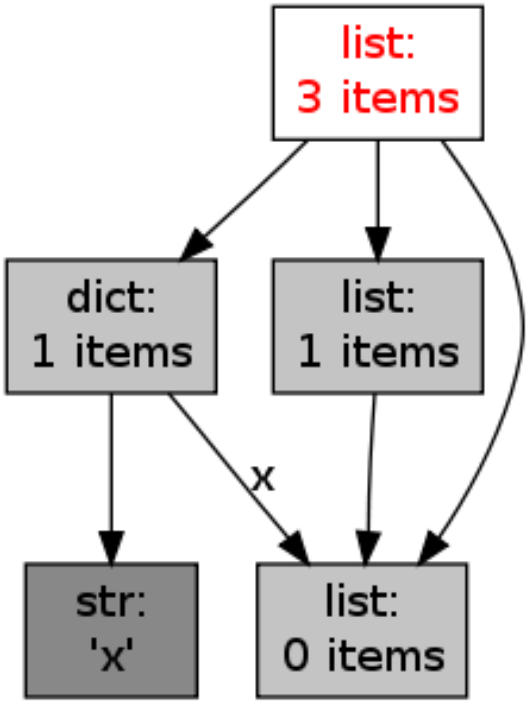
objgraph

(tired of text yet?)

http://mg.pov.lt/objgraph/

```
>>> import objgraph
>>> objgraph.show_refs([obj])
```

```
>>> import objgraph
>>> x = []
>>> y = [x, [x], dict(x=x)]
>>> objgraph.show_refs([y], filename='sample.png')
Graph written to /tmp/tmp3onTjF.dot (5 nodes)
Image generated as sample.png
```



Can also graph backrefs, define filters, highlights, and more.

valgrind

Try this if you suspect a C extension is leaking

For best results, compile your own Python following the instructions in `Misc/README.valgrind`

Typical use:

```
valgrind python foo.py
```

Thanks!

Links, slides — http://bemusement.org/pycon2010

(soon, I promise!)