Memory management in Python - how it works and is it worth messing with it?

Is it possible to have a memory leak in Python? What is the performance cost of garbage collector? When can I run into problems with memory and how can I solve them?

Maciej Pytel

codilime
CREATING VALUE

02/03/2015

About me

Software developer @ CodiLime. I code (mainly) in Python. I like to know how stuff works "inside".

Disclaimer

Which Python?

This talk is about Python 2.7.
By ”Python” I mean CPython.

Examples

`https://github.com/MaciekPytel/python-memory-talk`

1. Just a bit of theory

2. Why is it worth knowing this stuff? What (and how) can we do?
    - Example: RESTful server
    - Memory leaks

3. A few useful tools

4. More theory, or how it all works?

Section 1

Just a bit of theory

Reference counting

- Every object has a reference counter.
- When the counter reaches 0 the object is *immediately* deallocated.
- What about reference cycles?

Garbage collector

Mission: delete objects in unreacheable reference cycles.
Solution: garbage collector.

- Automatically run by Python interpreter every once in a while.
- Doesn't explicitly deallocate any objects - just breaks reference cycles.
- Freezes the process for garbage collection.
- Expensive: cost is (at least) linear with regard to the number of references in program.

Generational GC

Weak generational hypothesis

Most objects live either for a very short or a very long time.

Can this help us optimise GC?

Generational GC

- Divide objects into 3 generations (0, 1, 2).
- Each newly allocated object is added to generation 0.
- Long living objects are promoted to higher generations.
- Generation 0 is made of relatively few objects with relatively large chance of being ready for deallocation (according to weak generational hypothesis).

Generational GC

- n-th generation garbage collection analyses objects of generation 0..n.
- Any object that survived n-th generation collection is promoted to generation n + 1 (technically min(2, n + 1)).
- 0-th generetion collection is cheap and done often.
- Higher generation collections are more expensive and performed much less often.

Weak references

- Weak reference is a reference which doesn't increase reference counter of the object it points to.
- Existence of weak reference doesn't prevent deallocation of its target.
- In Python provided by *weakref* module.
- Dereferencing weak reference to deallocated object yields None.

Section 2

Why is it worth knowing this stuff? What (and how) can we do?

Subsection 1

Example: RESTful server

Experiment

- Server:
  - Flask-RESTful + gunicorn.
  - Domain made of 2 classes: Task and Subtask.
  - Natural one-to-many relationship (Each Task has multiple Subtasks).
  - Simple API supporting creation, retrieval and deletion of Tasks and Subtasks.
- Test: run a bunch of clients sending requests in a loop and measure server performance.

Task

```
class Task(object):
    def __init__(self, name):
        self.name = name
        self._subtasks = {}

    def add_subtask(self, name):
        self._subtasks[name] = Subtask(name, self)

    def get_subtask(self, name):
        return self._subtasks.get(name, None)
```

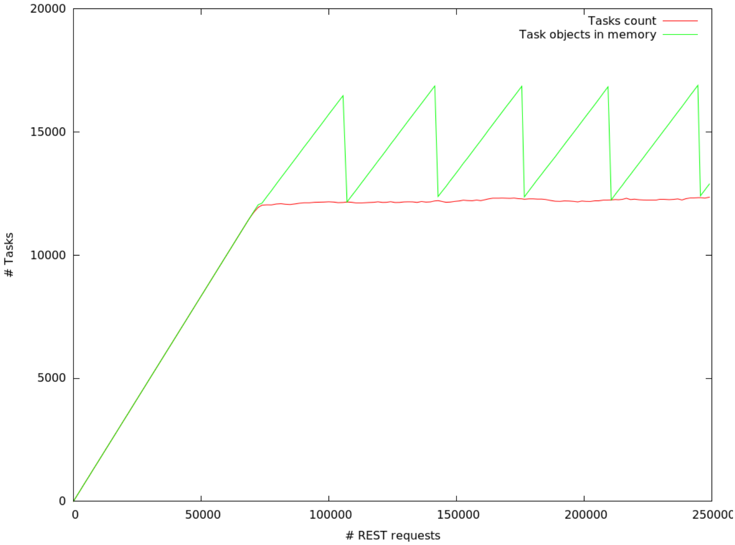
Subtask version 1

```
class Subtask(object):
    def __init__(self, name, task):
        self.name = name
        self._task = task

    def get_task(self):
        return self._task
```

REST API

- **GET /tasks/{task_id}**/ - return all Subtasks corresponding to a given Task
- **PUT /tasks/{task_id}**/ - create a Task with a given id
- **DELETE /tasks/{task_id}**/ - delete a Task and all corresponding Subtasks
- **GET /subtasks/{subtask_id}**/ - return a Subtask and its parent Task
- **PUT /subtasks/{task_id}/{subtask_id}** - create a Task with a given id and assign it to a Task

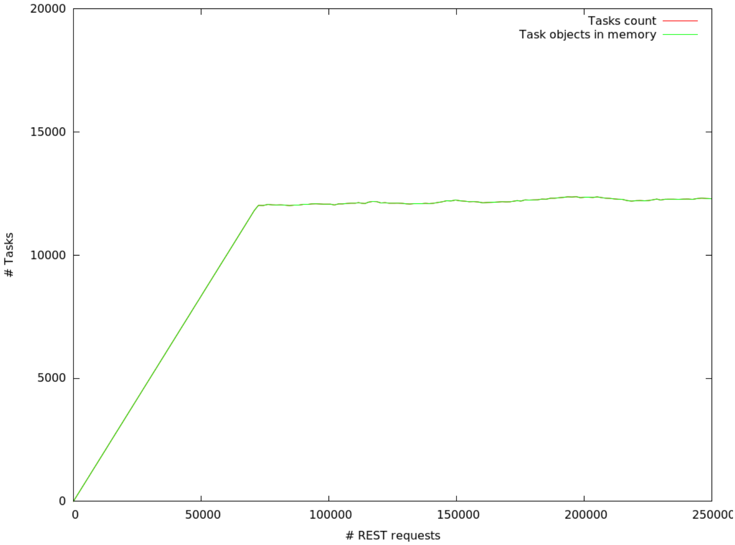
Results


Tasks count
Task objects in memory
20000
15000
10000
5000
0
Tasks
0
50000
100000
150000
200000
250000
REST requests

Subtask version 2

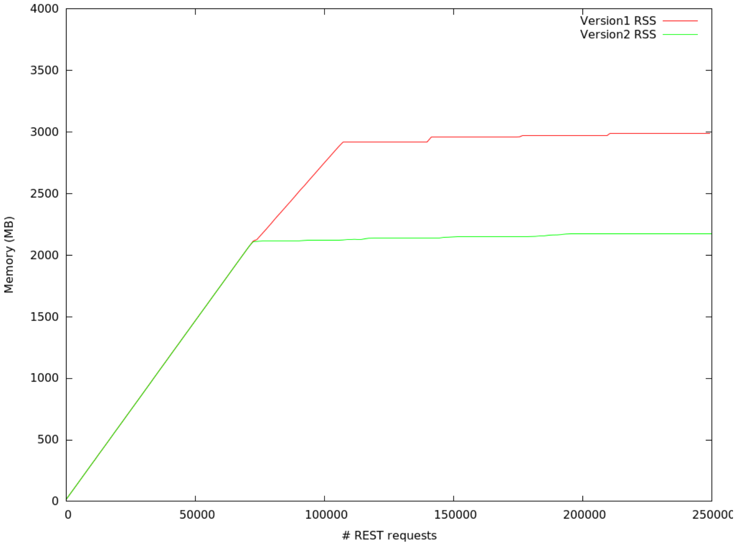
```
import weakref

class Subtask(object):
    def __init__(self, name, task):
        self.name = name
        self._task = weakref.ref(task)

    def get_task(self):
        return self._task()
```

Results?



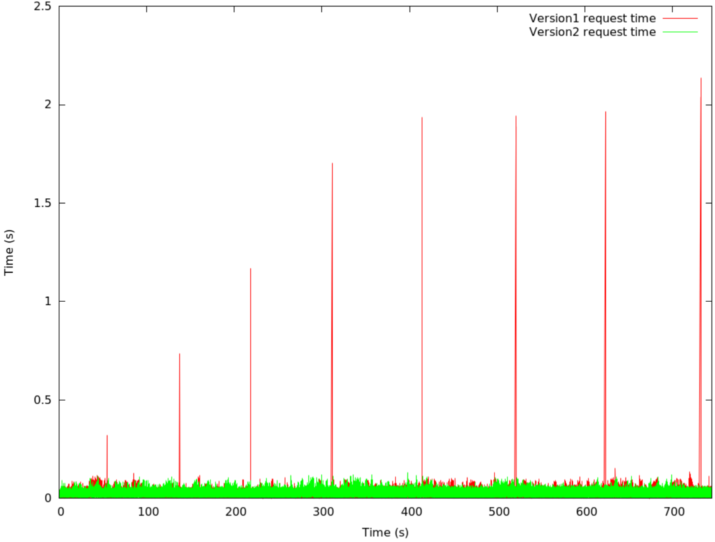
Tasks count
Task objects in memory
20000
15000
10000
5000
0
Tasks
0
50000
100000
150000
200000
250000
REST requests

Results



Version1 RSS
Version2 RSS
Memory (MB)
4000
3500
3000
2500
2000
1500
1000
500
0
0
50000
100000
150000
200000
250000
REST requests

Results



Version1 request time
Version2 request time
2.5
2
1.5
1
0.5
0
Time (s)
0
100
200
300
400
500
600
700
Time (s)

What have we learned?

- Relying on GC is expensive.
- Specifically 2nd generation GC takes a long time (2-2.5s in example server).
- Lower generation collections are significantly cheaper (no noticeable impact on request time for client).
- 2nd generation collection is run very rarely.
- Python is able to reuse freed memory, but it doesn't seem to return it to OS.

Freeing memory

- Python uses its own memory allocator (we'll skip the details).
- Allocator can return the memory to OS*, but:
  - Memory is allocated in blocks and only released once all objects in a block are deallocated (fragmentation!).
  - Many objects are returned to pool for reuse (ex. tuples).
  - Memory allocated for integers will never be freed or used for anything else.
  - Same with floats.
  - ...

Conclusion

If our program is having memory problems we may:

- Remove references to objects as soon as we don't need them anymore.
- Avoid *unnecessary* reference cycles.
- In most cases when they are necessary we can use weakrefs.
- Or explicitly break the cycle when we don't need the object anymore (ex. by defining and calling *delete* method).
- *xrange* instead of *range* in long loops makes a difference. Seriously, use generators.

Subsection 2

Memory leaks

Memory leaks

There are a few possible reasons for leaks:

1. Memory leak in c/c++ module.
2. Memory allocated for integers or floats will never be freed or used for anything else.
3. ”Forgotten” references.
4. Garbage collector won’t free *any* object in a cycle where at least a single object defines a finalizer (__*del*__).

"Forgotten" references

There are a few places where unexpected references can hide and keep your objects alive:

1. sys.exc_info() returns information about last exception handled in this stack frame. Part of this information is a traceback object containing whole stack state at the time of exception. If you catch an exception in your main loop: Oops.
2. Closures, functools.partial, etc.

Finalizer (__del__)

- If an object defines __del__ method it will be called just before object deallocation.
- Object can create a reference to itself in __del__ - this will prevent deallocation.
- **GC will never deallocate an object defining __del__!**
- In other words the whole cycle containing such an object will never be broken.
- **Warning:** __del__ is used in many existing libraries (including Python standard library!).

Finalizer

As we can see finalizers can cause problems. How do we deal with them?

- Don't use finalizers. Context manager (”with”) is almost always a better solution.
- If you really need a finalizer - make sure it's never a part of reference cycle (think decomposition).
- As a last resort: weakref.ref takes an optional callback parameter. This callback will be called after the object is deallocated. Problem: once the callback gets invoked the object no longer exists, so we need to keep the necessary state externally.

weakref based finalizer

```
class FileWrapper(object):
    _weakrefs = set()

    @classmethod
    def _delegated_close(cls, file_object, w):
        file_object.close()
        cls._weakrefs.remove(w)

    def __init__(self, name, mode):
        self._f = open(name, mode)
        self._weakrefs.add(weakref.ref(
            self,
            functools.partial(self._delegated_close,
                self._f)))
```

Section 3

A few useful tools

pdb

Debugger, part of Python standard library. Allows you to inspect the state of your program, execute code and evaluate expressions in any stack frame. The basic Python debugging tool.

Python standard library module.

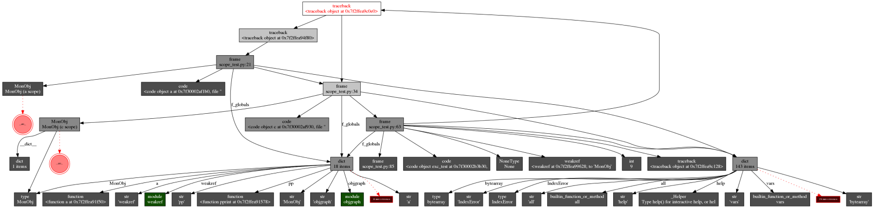
- **gc.collect(generation=2)** - trigger GC manually.
- **gc.enable()** / **gc.disable()** - enable/disable automatic GC.
- **gc.garbage** - a list of all objects that should be deleted, but define __del__ method and are a part of reference cycle.
- You can set debugging flags, ex. to print GC statistics with variable level of details.
- Can give you a full list of objects referring a given object.

objgraph

Simple library for analysys and visulisation of reference graph. Very useful when looking for memory leaks.

- Available via pip.
- Visualise reference graphs.
- Count objects in memory by type.
- Also display change in number of objects of given type in time.
- Sufficient to diagnose 99% of memory problems.

objgraph



traceback
<traceback object at 0x7f2ffea9c0e0>
traceback
<traceback object at 0x7f2ffea94f80>
frame
scope_test.py:21
MonObj
MonObj (a scope)
code
<code object a at 0x7f30002af1b0, file "
frame
scope_test.py:34
f_globals
MonObj
MonObj (c scope)
code
<code object c at 0x7f30002af930, file "
f_globals
frame
scope_test.py:63
__dict__
dict
1 items
f_globals
dict
18 items
frame
scope_test.py:85
code
<code object exc_test at 0x7f30002b3b30,
NoneType
None
weakref
<weakref at 0x7f2ffea99628; to 'MonObj'
int
9
traceback
<traceback object at 0x7f2ffea9c128>
dict
143 items
MonObj
a
weakref
pp
objgraph
bytearray
IndexError
all
help
vars
type
MonObj
function
<function a at 0x7f2ffea91f50>
str
'weakref'
module
weakref
str
'pp'
function
<function pprint at 0x7f2ffea91578>
str
'MonObj'
str
'objgraph'
module
objgraph
str
'a'
type
bytearray
str
'IndexError'
type
IndexError
str
'all'
builtin_function_or_method
all
str
'help'
_Helper
Type help() for interactive help, or hel
str
'vars'
builtin_function_or_method
vars
str
'bytearray'

guppy/heapy

- Memory profiler.
- Incredible number of available functions: analyse paths in reference graph, reference graph spanning trees (!), grouping objects by various equality relationships (ex. class, module the originates from), ...
- Provides hooks for tracking objects defined in C module.
- Steep learning curve, not the best documentation and lack of good tutorials.
- For this 1% of situations when objgraph simply isn't enough :)

Section 4

More theory, or how it all works?

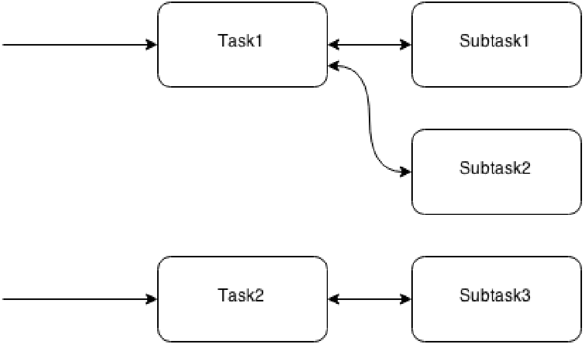
Generations

- Each generation is a doubly-linked list.
- Each newly allocated object is added to generation 0.
- Before n-th generation GC lower generations' lists are merged to n-th generation list.
- After n-th generation GC is done surviving objects' list is merged to generation n + 1.

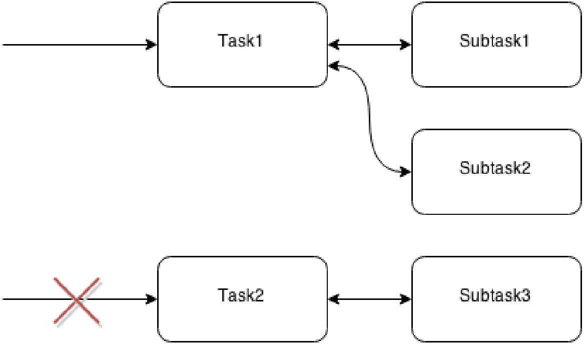
GC algorithm

1. (For generation 1 and 2) Merge lower generation into current generation (doubly-linked list merge).
2. Iterate over objects on the list. Check all outgoing references (non-recursively) - if they point to other objects on the list decrease target's refcount.
3. Iterate over objects on the list. Move any object with refcount equal to 0 to a new *unreacheable* list.
4. Iterate over objects remaining on the original list. Check all outgoing references and move any target on *unreacheable* list back to the original list.
5. Restore original refcounts on all objects.
6. Merge original list to generation n + 1.
7. For each object on *unreacheable* list remove all references originating from that object.

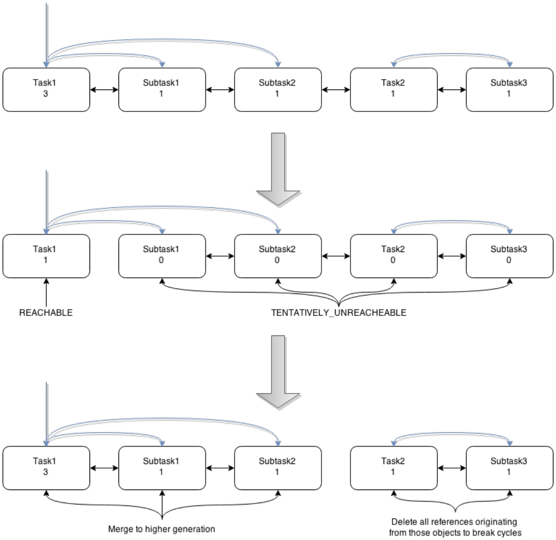
Example



Example



GC - visualisation



Notes

- The above is the idea of the algorithm. The real implementation is a bit more complex.
- Details in Python source /Modules/gcmodule.c - very well commented.

Quote

Programmers waste enormous amounts of time thinking about the speed of noncritical parts of their programs (...). We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

- Donald Knuth