Garbage Collection

Python 3.x 3.5–3.6

Improvements requested:

Examples

Reuse of primitive objects

An interesting thing to note which may help optimize your applications is that primitives are actually also refcounted under the hood. Let's take a look at numbers; for all integers between -5 and 256, Python always reuses the same object:

```
>>> import sys
>>> sys.getrefcount(1)
797
>>> a = 1
>>> b = 1
>>> sys.getrefcount(1)
799
```

Note that the refcount increases, meaning that `a` and `b` reference the same underlying object when they refer to the `1` primitive. However, for larger numbers, Python actually doesn't reuse the underlying object:

```
>>> a = 999999999
>>> sys.getrefcount(999999999)
3
>>> b = 999999999
>>> sys.getrefcount(999999999)
3
```

Because the refcount for `999999999` does not change when assigning it to `a` and `b` we can infer that they refer to two different underlying objects, even though they both are assigned the same primitive.

Effects of the del command

Removing a variable name from the scope using `del v`, or removing an object from a collection using `del v[item]` or `del[i:j]`, or removing an attribute using `del v.name`, or any other way of removing references to an object, *does not* trigger any destructor calls or any memory being freed in and of itself. Objects are only destructed when their reference count reaches zero.

```
>>> import gc
>>> gc.disable()  # disable garbage collector
>>> class Track:
        def __init__(self):
            print("Initialized")
        def __del__(self):
            print("Destructed")
>>> def bar():
    return Track()
>>> t = bar()
Initialized
>>> another_t = t  # assign another reference
>>> print("...")
...
>>> del t          # not destructed yet - another_t still refers to it
>>> del another_t  # final reference gone, object is destructed
Destructed
```

Forcefully deallocating objects

You can force deallocate objects even if their refcount isn't 0 in both Python 2 and 3.

Both versions use the `ctypes` module to do so.

WARNING: doing this *will* leave your Python environment unstable and prone to crashing without a traceback! Using this method could also introduce security problems (quite unlikely) Only deallocate objects you're sure you'll never reference again. Ever.

Python 3.x ≥3.0

```
import ctypes
deallocated = 12345
```

```
ctypes.pythonapi._Py_Dealloc(ctypes.py_object(deallocated))
```

Python 2.x ≥2.3

```
import ctypes, sys
deallocated = 12345
(ctypes.c_char * sys.getsizeof(deallocated)).from_address(id(deallocated))[:4] = '\x00' * 4
```

After running, any reference to the now deallocated object will cause Python to either produce undefined behavior or crash - without a traceback. There was probably a reason why the garbage collector didn't remove that object...

If you deallocate `None`, you get a special message - `Fatal Python error: deallocating None` before crashing.

Garbage Collector for Reference Cycles

The only time the garbage collector is needed is if you have a *reference cycle*. The simples example of a reference cycle is one in which A refers to B and B refers to A, while nothing else refers to either A or B. Neither A or B are accessible from anywhere in the program, so they can safely be destructed, yet their reference counts are 1 and so they cannot be freed by the reference counting algorithm alone.

```
>>> import gc; gc.disable()  # disable garbage collector
>>> class Track:
        def __init__(self):
            print("Initialized")
        def __del__(self):
            print("Destructed")
>>> A = Track()
Initialized
>>> B = Track()
Initialized
>>> A.other = B
>>> B.other = A
>>> del A; del B  # objects are not destructed due to reference cycle
>>> gc.collect()  # trigger collection
Destructed
Destructed
4
```

A reference cycle can be arbitrary long. If A points to B points to C points to ... points to Z which points to A, then neither A through Z will be collected, until the garbage collection phase:

```
>>> objs = [Track() for _ in range(10)]
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
>>> for i in range(len(objs)-1):
...     objs[i].other = objs[i + 1]
...
>>> objs[-1].other = objs[0]  # complete the cycle
>>> del objs                  # no one can refer to objs now - still not destructed
>>> gc.collect()
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
20
```

Reference Counting

The vast majority of Python memory management is handled with reference counting.

Every time an object is referenced (e.g. assigned to a variable), its reference count is automatically increased. When it is dereferenced (e.g. variable goes out of scope), its reference count is automatically

decreased.

When the reference count reaches zero, the object is **immediately destroyed** and the memory is immediately freed. Thus for the majority of cases, the garbage collector is not even needed.

```
>>> import gc; gc.disable()  # disable garbage collector
>>> class Track:
        def __init__(self):
            print("Initialized")
        def __del__(self):
            print("Destructed")
>>> def foo():
        Track()
        # destructed immediately since no longer has any references
        print("---")
        t = Track()
        # variable is referenced, so it's not destructed yet
        print("---")
        # variable is destructed when function exits
>>> foo()
Initialized
Destructed
---
Initialized
---
Destructed
```

To demonstrate further the concept of references:

```
>>> def bar():
        return Track()
>>> t = bar()
Initialized
>>> another_t = t  # assign another reference
>>> print("...")
...
>>> t = None          # not destructed yet - another_t still refers to it
>>> another_t = None  # final reference gone, object is destructed
Destructed
```

Viewing the refcount of an object

```
>>> import sys
>>> a = object()
>>> sys.getrefcount(a)
2
>>> b = a
>>> sys.getrefcount(a)
3
>>> del b
>>> sys.getrefcount(a)
2
```

Forcing garbage collection

Using following command you can enforce garbage collection

```
import gc
gc.collect()
```

Reference counting is implementation detail

The fact that the garbage collection is considered as an implementation detail and good python code should not rely on it.

for example:

In the following code

```
>>> f = open("test.txt")
>>> del f
```

Under Cpython (the standard implementation) the file is close as soon as the `del` line is executed as the reference count of file object reach zero and it is eventually garbaged collected. However, one should not

reference count of file object reach zero and it is eventually garbaged collected. However, one should not rely on this behavior, in other implementation such as pypy, the file close will happen at a future time which can be at the end of the program.

The right way to guarantee the implicit file close is to use a context managers :

```
>>> with open("test.txt") as f:
...     pass
...     # do something with f
>>> #now the f object still exist, but it is closed
```

Syntax

Parameters

Remarks

At its core, Python's garbage collector (as of 3.5) is a simple reference counting implementation. Every time you make a reference to an object (for example, `a = myobject`) the reference count on that object (myobject) is incremented. Every time a reference gets removed, the reference count is decremented, and once the reference count reaches `0` , we know that nothing holds a reference to that object and we can deallocate it!

One common misunderstanding about how Python memory management works is that the `del` keyword frees objects memory. This is not true. What actually happens is that the `del` keyword merely decrements the objects refcount, meaning that if you call it enough times for the refcount to reach zero the object may be garbage collected (even if there are actually still references to the object available elsewhere in your code).

Python aggresively creates or cleans up objects the first time it needs them If I perform the assignment a = object(), the memory for object is allocated at that time (cpython will sometimes reuse certain types of object, eg. lists under the hood, but mostly it doesn't keep a free object pool and will perform allocation when you need it). Similarly, as soon as the refcount is decremented to 0, GC cleans it up.

Generational Garbage Collection

In the 1960's John McCarthy discovered a fatal flaw in refcounting garbage collection when he implemented the refcounting algorithm used by Lisp: What happens if two objects refer to each other in a cyclic reference? How can you ever garbage collect those two objects even if there are no external references to them if they will always refer to eachother? This problem also extends to any cyclic data structure, such as a ring buffers or any two consecutive entries in a doubly linked list. Python attempts to fix this problem using a slightly interesting twist on another garbage collection algorithm called **Generational Garbage Collection** .

In essence, any time you create an object in Python it adds it to the end of a doubly linked list. On occasion Python loops through this list, checks what objects the objects in the list refer too, and if they're also in the list (we'll see why they might not be in a moment), further decrements their refcounts. At this point (actually, there are some heuristics that determine when things get moved, but let's assume it's after a single collection to keep things simple) anything that still has a refcount greater than 0 gets promoted to another linked list called "Generation 1" (this is why all objects aren't always in the generation 0 list) which has this loop applied to it less often. This is where the generational garbage collection comes in. There are 3 generations by default in Python (three linked lists of objects): The first list (generation 0) contains all new objects; if a GC cycle happens and the objects are not collected, they get moved to the second list (generation 1), and if a GC cycle happens on the second list and they are still not collected they get moved to the third list (generation 2). The third generation list (called "generation 2", since we're zero indexing) is garbage collected much less often than the first two, the idea being that if your object is long lived it's not as likely to be GCed, and may never be GCed during the lifetime of your application so there's no point in wasting time checking it on every single GC run. Furthermore, it's observed that most objects are garbage collected relatively quickly. From now on, we'll call these "good objects" since they die young. This is called the "weak generational hypothesis" and was also first observed in the 60s.

A quick aside: unlike the first two generations, the long lived third generation list is not garbage collected on a regular schedule. It is checked when the ratio of long lived pending objects (those that are in the third generation list, but haven't actually had a GC cycle yet) to the total long lived objects in the list is greater than 25%. This is because the third list is unbounded (things are never moved off of it to another list, so they only go away when they're actually garbage collected), meaning that for applications where you are creating lots of long lived objects, GC cycles on the third list can get quite long. By using a ratio we achieve "amortized linear performance in the total number of objects"; aka, the longer the list, the longer GC takes, but the less often we perform GC (here's the original 2008 proposal for this heuristic by Martin von Löwis for futher reading). The act of performing a garbage collection on the third generation or "mature" list is called "full garbage collection".

So the generational garbage collection speeds things up tremdously by not requiring that we scan over objects that aren't likely to need GC all the time, but how does it help us break cyclic references? Probably not very well, it turns out. The function for actually breaking these reference cycles starts out like this :

```
/* Break reference cycles by clearing the containers involved.  This is
 * tricky business as the lists can be changing and we don't know which
 * objects may be freed.  It is possible I screwed something up here.
 */
static void
delete_garbage(PyGC_Head *collectable, PyGC_Head *old)
```

The reason generational garbage collection helps with this is that we can keep the length of the list as a separate count; each time we add a new object to the generation we increment this count, and any time we move an object to another generation or dealloc it we decrement the count. Theoretically at the end of a GC cycle this count (for the first two generations anyways) should always be 0. If it's not, anything in the list that's left over is some form of circular reference and we can drop it. However, there's one more problem here: What if the leftover objects have Python's magic method `__del__` on them? `__del__` is called any time a Python object is destroyed. However, if two objects in a circular reference have `__del__` methods, we can't be sure that destroying one won't break the others `__del__` method. For a contrived example, imagine we wrote the following:

```
class A(object):
    def __init__(self, b=None):
        self.b = b

    def __del__(self):
        print("We're deleting an instance of A containing:", self.b)

class B(object):
    def __init__(self, a=None):
        self.a = a

    def __del__(self):
        print("We're deleting an instance of B containing:", self.a)
```

and we set an instance of A and an instance of B to point to one another and then they end up in the same garbage collection cycle? Let's say we pick one at random and dealloc our instance of A first; A's `__del__` method will be called, it will print, then A will be freed. Next we come to B, we call its `__del__` method, and oops! Segfault! A no longer exists. We could fix this by calling everything that's left over's `__del__` methods first, then doing another pass to actually dealloc everything, however, this introduces another, issue: What if one objects `__del__` method saves a reference of the other object that's about to be GCed and has a reference to us somewhere else? We still have a reference cycle, but now it's not possible to actually GC either object, even if they're no longer in use. Note that even if an object is not part of a circular data structure, it could revive itself in its own `__del__` method; Python does have a check for this and will stop GCing if an objects refcount has increased after its `__del__` method has been called.

CPython deals with this is by sticking those un-GC-able objects (anything with some form of circular reference and a `__del__` method) onto a global list of uncollectable garbage and then leaving it there for all eternity:

```
/* list of uncollectable objects */
static PyObject *garbage = NULL;
```