Python Garbage Collection

From Digi Developer

Contents

Introduction to Python Memory Management

Python's memory allocation and deallocation method is automatic. The user does not have to preallocate or deallocate memory by hand as one has to when using dynamic memory allocation in languages such as C or C++. Python uses two strategies for memory allocation **reference counting** and **garbage collection**.

Prior to Python version 2.0, the Python interpreter only used reference counting for memory management. Reference counting works by counting the number of times an object is referenced by other objects in the system. When references to an object are removed, the reference count for an object is decremented. When the reference count becomes zero the object is deallocated.

Reference counting is extremely efficient but it does have some caveats. One such caveat is that it cannot handle *reference cycles.* A reference cycle is when there is no way to reach an object but its reference count is still greater than zero. The easiest way to create a reference cycle is to create an object which refers to itself as in the example below:

```
def make_cycle():
    l = [ ]
    l.append(l)

make_cycle()
```

Because `make_cycle()` creates an object `l` which refers to itself, the object `l` will not automatically be freed when the function returns. This will cause the memory that `l` is using to be held onto until the Python garbage collector is invoked.

Automatic Garbage Collection of Cycles

Because reference cycles are take computational work to discover, garbage collection must be a scheduled activity. Python schedules garbage collection based upon a threshold of object allocations and object deallocations. When the number of allocations minus the number of deallocations are greater than the threshold number, the garbage collector is run. One can inspect the threshold for new objects (objects in Python known as *generation 0* objects) by loading the gc module and asking for garbage collection thresholds:

```
import gc
print "Garbage collection thresholds: %r" % gc.get_threshold()
```

```
Garbage collection thresholds: (700, 10, 10)
```

Here we can see that the default threshold on the above system is 700. This means when the number of allocations vs. the number of deallocations is greater than 700 the automatic garbage collector will run.

Automatic garbage collection will not run if your Python device is running out of memory; instead your application will throw exceptions, which must be handled or your application crashes. This is aggravated by the fact that the automatic garbage collection places high weight upon the NUMBER of free objects, not on how large they are. Thus any portion of your code which frees up large blocks of memory is a good candidate for running manual garbage collection.

Manual Garbage Collection

For some programs, especially long running server applications or embedded applications running on a Digi Device automatic garbage collection may not be sufficient. Although an application should be written to be as free of reference cycles as possible, it is a good idea to have a strategy for how to deal with them. Invoking the garbage collector manually during opportune times of program execution can be a good idea on how to handle memory being consumed by reference cycles.

The garbage collection can be invoked manually in the following way:

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

`gc.collect()` returns the number of objects it has collected and deallocated. You can print this information in the following way:

```
import gc
collected = gc.collect()
print "Garbage collector: collected %d objects." % (collected)
```

If we create a few cycles, we can see manual collection work:

```
import sys, gc

def make_cycle():
    l = { }
    l[0] = l

def main():
    collected = gc.collect()
    print "Garbage collector: collected %d objects." % (collected)
    print "Creating cycles..."
    for i in range(10):
        make_cycle()
    collected = gc.collect()
    print "Garbage collector: collected %d objects." % (collected)

if __name__ == "__main__":
    ret = main()
    sys.exit(ret)
```

In general there are two recommended strategies for performing manual garbage collection: time-based and event-based garbage collection. Time-based garbage collection is simple: the garbage collector is called on a fixed time interval. Event-based garbage collection calls the garbage collector on an event. For example, when a user disconnects from the application or when the application is known to enter an idle state.

Recommendations

Which garbage collection technique is correct for an application? It depends. The garbage collector should be invoked as often as necessary to collect cyclic references without affecting vital application performance. Garbage collection should be a part of your Python application design process.

- Do not run garbage collection too freely, as it can take considerable time to evaluate every memory object within a large system. For example, one team having memory issues tried calling gc.collect() between every step of a complex start-up process, increasing the boot time by 20 times (2000%). Running it more than a few times per day - without specific design reasons - is likely a waste of device resources.

- Run manual garbage collection after your application has completed start up and moves into steady-state operation. This frees potentially huge blocks of memory used to open and parse file, to build and modify object lists, and even code modules never to be used again. For example, one application reading XML configuration files was consuming about 1.5MB of temporary memory during the process. Without manual garbage collection, there is no way to predict when that 1.5MB of memory will be returned to the python memory pools for reuse.

- Run manual garbage collection after infrequently run sections of code which use and then free large blocks of memory. For example, consider running garbage collection after a once-per-day task which evaluates thousands of data points, creates an XML 'report', and then sends that report to a central office via FTP or SMTP/email. One application doing such daily reports was creating over 800K worth of temporary sorted lists of historical data. Piggy-backing gc.collect() on such daily chores has the nice side-effect of running it once per day for 'free'.

- Consider manually running garbage collection either before or after timing-critical sections of code to prevent garbage collection from disturbing the timing. As example, an irrigation application might sit idle for 10 minutes, then evaluate the status of all field devices and make adjustments. Since delays during system adjustment might affect field device battery life, it makes sense to manually run garbage collection as the gateway is entering the idle period AFTER the adjustment process - or run it every sixth or tenth idle period. This insures that garbage collection won't be triggered automatically during the next timing-sensitive period.

Further References and Reading

- Official Python documentation on the gc module: http://docs.python.org/library/gc.html
- An article on cyclic references and Python garbage collection: http://arctrix.com/nas/python/gc/
- Wikipedia's entry on Garbage Collection: http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

Retrieved from "http://www.digi.com/wiki/developer/index.php/Python_Garbage_Collection"

Categories: General Python | Good Python Suggestions
