Heapy

a memory profiler and debugger for Python

Sverker Nilsson
sverker.is@home.se

June 2, 2006

Goal

- Make a tool for the Python programming language
- Support memory debugging and optimization
- Provide data not available directly in Python
- Manage complexity of large programs
- Design to generalise well to new situations

The engineer wishes

- To make programs that run in limited memory
- Especially long running and embedded systems
- Avoiding guesswork by accurate observations
- Using knowledge to make wise optimizations

The problem

- Automatic memory management is not automatic
- Garbage collection frees unreferenced objects only
- Referenced objects may still be useless to keep
- Complex programs are easier to make using GC
- Tools needed to understand memory behaviour
- Has been a lack of such tools for Python

Questions raised

- HOW much memory is used by objects?
- WHAT objects are of most interest?
- WHY are objects retained in memory?

HOW much memory is used by objects?

- No built in support for this in Python
- Requires code to look into objects at implementation (C)level
- Heapy provides this code for predefined and user defined Python objects
- Special problems with objects from extension modules
- An interface is defined so extension modules can supply functions for sizing and other information about their types

WHAT objects are of interest?

- All objects in memory may be of general interest, except those used only for analysis purposes
- Of special interest are objects that use much memory, either because they are big or there are many of them,
- and objects that are no longer of any use to the program --- *memory leaks*,
- and objects that refer to other objects, keeping them in memory perhaps unnecessarily

WHY are objects retained in memory?

- Is there any good reason?
- If not, there is still some reason but a bad one
- Objects are generally retained because they are referenced by other objects
- The referrers and their relations can tell if objects are retained for a good reason or not
- The reference graph may be too big and complex to understand directly
- To manage complexity, summarizing views exist such as reference pattern and paths to root

Memory leaks

- Memory that is allocated but is no more used
- Problem for long running applications and when memory is sparse
- Can occur even with automatic garbage collection
- Garbage collection frees objects when they can not possibly be used anymore i.e. when there are no references left
- Leaking objects are referenced but still of no use

Finding memory leaks

- A symptom is often that memory usage tends to increase with time
- Often a critical section can be identified where memory usage increases unexpectantly
- An example is opening and closing a window when one expects all objects used by the window to be freed after it is closed
- Comparing memory population before and after the critical section may reveal the leaking objects

Memory profiling

- To get an overview and find critical sections where memory leaks are likely to occur
- Shows memory usage of objects grouped by different criteria
- Shows memory usage as it evolves with time

Different kinds of memory profiling

- A *constructor profile* classifies cells according to the kinds of values they represent
- A *retainer profile* classifies cells by information about the active components that retain access to the cells
- A *producer profile* classifies cells by the program components that created them
- A *lifetime profile* classifies cells by the cell's eventual lifetime
- *Lag, drag and void* include *usage* information

Constructor profiling

- Classifies objects by type or class
- Type is a built in attribute of Python objects, eg a predefined type (list, int etc) or user defined
- Class is the same as Type in "new style" objects

Retainer profiling

- Retainer edge classification – consists of a set of edge descriptions such as attribute name, indices or keys
- Retainer classification – consists of a set of classifications of the retainers themselves

Retainer edge profiling example

```
>>> (h.heap()&str).byvia
Partition of a set of 14205 objects. Total size = 845464 bytes.
 Index  Count   %     Size   % Cumulative  % Referred Via:
     0   1510  11   156240  18     156240  18 '.co_code'
     1   1511  11    99432  12     255672  30 '.co_filename'
...
```

- 1510 strings referred via 'co_code' attribute
- 1511 strings referred via 'co_filename' attribute
- One filename for each code object is suspect
- Obvious optimization possibility
- The code objects could share file names

Example optimization suggested by retainer edge profiling

- Code objects could share file name strings
- Optimization was introduced in Python 2.4
- Could possibly been found quicker using profiling

```
>>> (h.heap()&str).byvia
Partition of a set of 13082 objects. Total size = 935964 bytes.
 Index  Count   %     Size   % Cumulative  % Referred Via:
     0   2605  20   288004  31     288004  31 '.co_code'
...
    11     55   0     3312   0     729420  78 '.co_filename'
...
```

Other profiling

- Not implemented in Heapy 0.1
- A *producer profile* classifies cells by the program components that created them
- *Lifetime profiler* classifies each object according to its eventual lifetime
- *Lag, drag and void* include *usage* information
- Drag is the time after last use until an object may actually be freed – can find leaking objects
- Can not be generated continuously, only after the program has finished, with special instrumentation

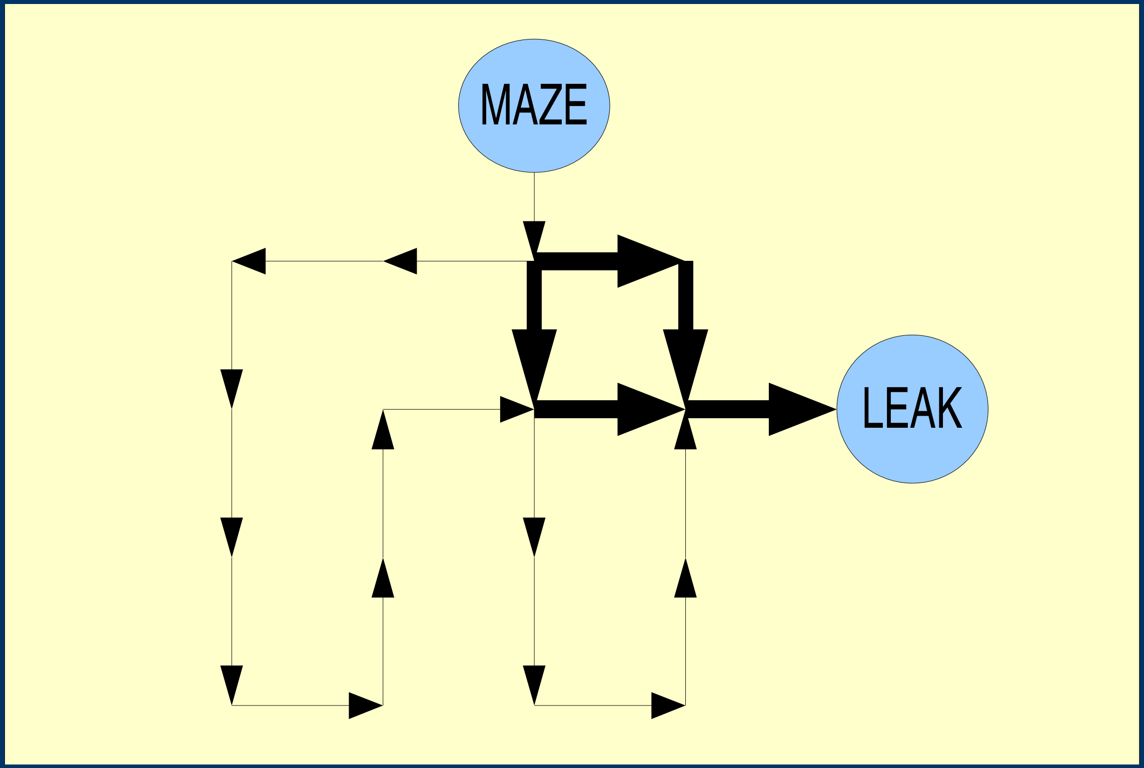
The WHY question revisited
Why are objects retained in memory?

- Sometimes answered directly by profiling
- Otherwise have to look into the reference graph
- The entire graph could be overwhelming
- Need for different summarizing views
- Path from root analysis
- Reference pattern

Path from root analysis

- Assumes a root from which objects can be reached
- A path is a walk visiting any node at most once
- A single path may tell why an object is retained
- But there may be astronomical numbers of paths
- Finding the *interesting* paths among all paths may be practically impossible to do manually
- The *shortest* paths are often much fewer than all the paths and of special interest by themselves
- If the shortest paths are not enough, it is possible to find longer paths

Shortest paths from root example

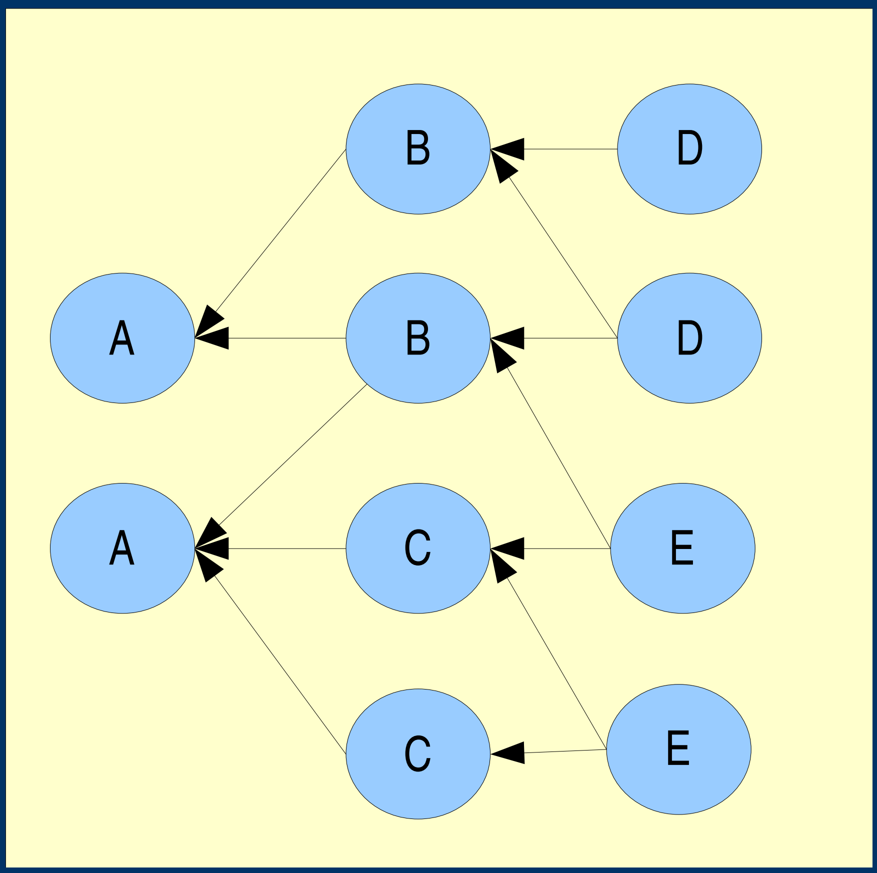


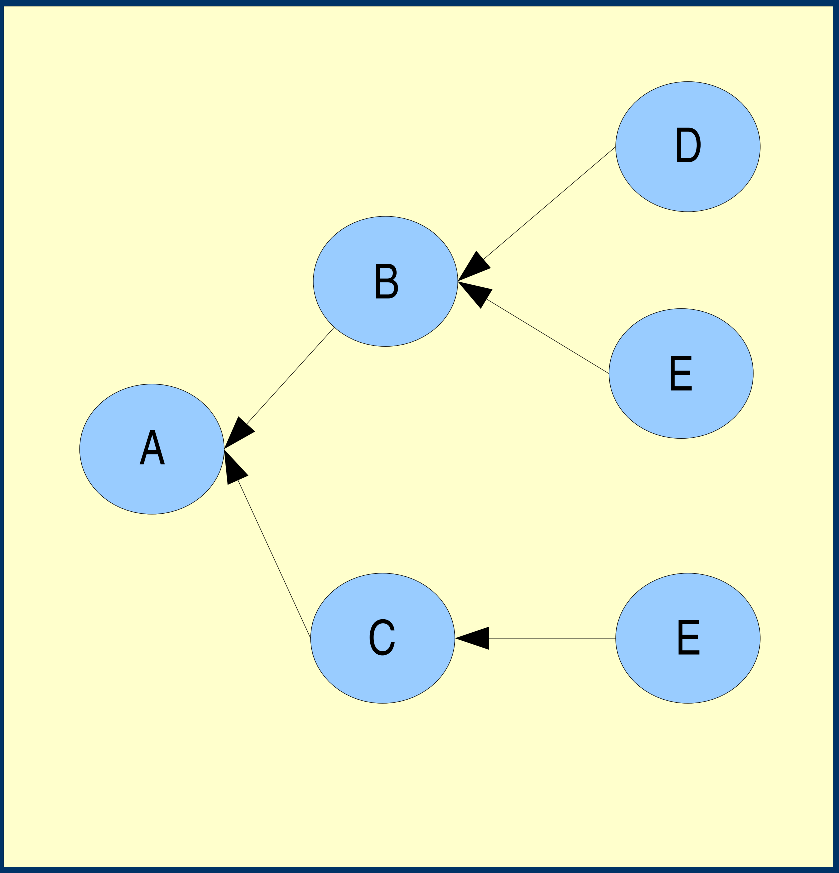
- Shortest paths: **S.E.E & E.S.E**
- Longest: W.W.S.S.S.E.N.N.E.S.S.E.N.N.E
- Other: W.W.S.S.S.E.N.N.E.E.E

Reference pattern

- Another way to manage complexity and tell why objects are retained in memory
- Simplifies the reference graph when there is much repetition in the data structures
- Treats retainer objects of the same kind as one unit
- The reference pattern is itself a graph
- Presented as a spanning tree

Reference pattern example





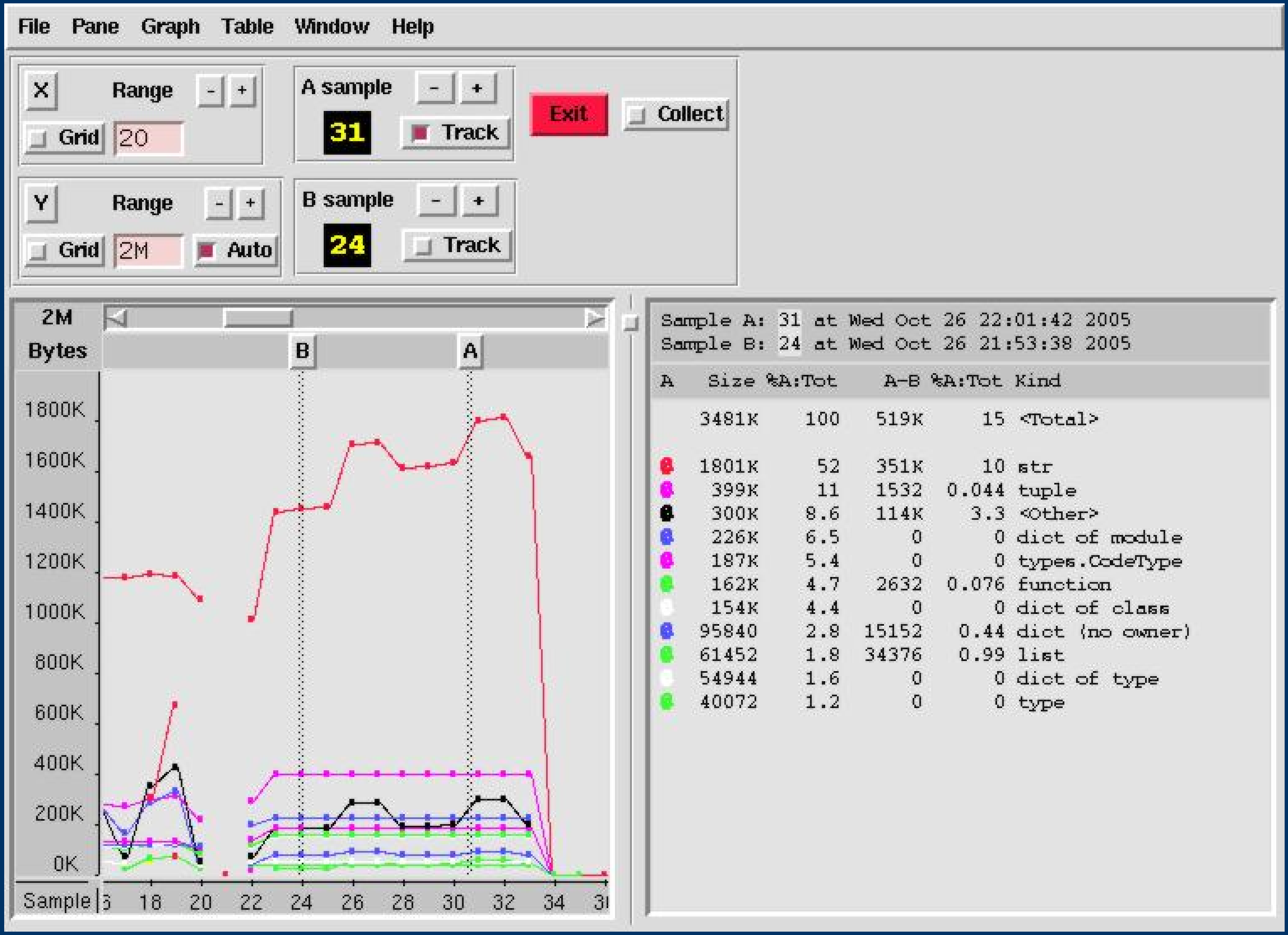
- Reference graph

Reference pattern

Other design concepts

- Universal sets – unify different set representations
- Identity sets – address based object wrappers
- Kind objects – symbolic sets
- Equivalence relations – classification definitions
- Remote monitor – separates observer from target
- Profile browser – shows graphical time series

Profile browser example



Example, finding & sealing a memory leak

- The target was a GUI application
- The critical section was open – close of a window
- Remote monitor enabled transparent observation
- Snapshot was taken before the open operation
- Window was then opened and closed
- New snapshot was taken and compared to old one
- The difference was a set of leaked objects
- Shortest paths and reference pattern show context
- The leak cause was found in library widget code
- Repair could be tested directly from the monitor
- Finally the source code could be fixed and tested

Summary of main features

- Information not available directly in Python is provided such as object sizes and relations
- Various memory profilers are designed to help finding unknown optimization possibilities
- Leaking objects can be extracted by comparing different memory population snapshots
- Reference patterns and shortest reference paths can help tell why objects are kept in memory
- Accurate observation using special C techniques
- Concepts such as sets and equivalence relations are intended to generalize well to new situations

Future work

- More kinds of profiling, some of which may rely on modifying the Python virtual machine
- Improved reference patterns for complex cases
- Automatic validation of expected memory usage
- Readily support common extension modules
- More tests, examples and documentation
- Make sure to work in various operating systems
- Theoretical model building and analysis, maybe using concepts from cognitive science such as distributed cognition

THANK YOU

- Heapy is released under an Open Source license
- Tested with C Python 2.3 – 2.4
- Known to compile so far in Linux
- Source code is available for download
- `http://guppy-pe.sourceforge.net`