Master Thesis

# Heapy

## A Memory Profiler and Debugger for Python

by

**Sverker Nilsson**

LITH-IDA-EX--06/043--SE

2006-06-02

**Avdelning, institution**
Division, department

Institutionen för datavetenskap

Department of Computer
and Information Science

**Datum**
Date

2006-06-02

Linköpings universitet

**Språk**
Language

[ ] Svenska/Swedish
[X] Engelska/English

[ ]

**Rapporttyp**
Report category

[ ] Licentiatavhandling
[X] Examensarbete
[ ] C-uppsats
[ ] D-uppsats
[ ] Övrig rapport

**ISBN** —

**ISRN** LITH-IDA-EX--06/043--SE

**Serietitel och serienummer** **ISSN**
Title of series, numbering —

**URL för elektronisk version**

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-7247

**Titel**
Title

# Heapy: A Memory Profiler and Debugger for Python

**Författare**
Author

Sverker Nilsson

**Sammanfattning**
Abstract

Excessive memory use may cause severe performance problems and system crashes. Without appropriate tools, it may be difficult or impossible to determine why a program is using too much memory. This applies even though Python provides automatic memory management — garbage collection can help avoid many memory allocation bugs, but only to a certain extent due to the lack of information during program execution. There is still a need for tools helping the programmer to understand the memory behaviour of programs, especially in complicated situations. The primary motivation for Heapy is that there has been a lack of such tools for Python.

The main questions addressed by Heapy are how much memory is used by objects, what are the objects of most interest for optimization purposes, and why are objects kept in memory. *Memory leaks* are often of special interest and may be found by comparing snapshots of the heap population taken at different times. *Memory profiles*, using different kinds of classifiers that may include *retainer* information, can provide quick overviews revealing optimization possibilities not thought of beforehand. *Reference patterns* and *shortest reference paths* provide different perspectives of object access patterns to help explain why objects are kept in memory.

**Nyckelord**
Keywords

memory profiling, memory leaks, debugging, optimization, testing, quality assurance, Python

Linköpings universitet
Department of Computer and Information Science

Master Thesis

# Heapy

## A Memory Profiler and Debugger for Python

by

**Sverker Nilsson**
*sn@sncs.se*

LITH-IDA-EX--06/043--SE

2006-06-02

Supervisor: Peter Fritzson
Examiner: Peter Fritzson

**Abstract**

Excessive memory use may cause severe performance problems and system crashes. Without appropriate tools, it may be difficult or impossible to determine why a program is using too much memory. This applies even though Python provides automatic memory management — garbage collection can help avoid many memory allocation bugs, but only to a certain extent due to the lack of information during program execution. There is still a need for tools helping the programmer to understand the memory behaviour of programs, especially in complicated situations. The primary motivation for Heapy is that there has been a lack of such tools for Python.

The main questions addressed by Heapy are how much memory is used by objects, what are the objects of most interest for optimization purposes, and why are objects kept in memory. *Memory leaks* are often of special interest and may be found by comparing snapshots of the heap population taken at different times. *Memory profiles*, using different kinds of classifiers that may include *retainer* information, can provide quick overviews revealing optimization possibilities not thought of beforehand. *Reference patterns* and *shortest reference paths* provide different perspectives of object access patterns to help explain why objects are kept in memory.

# Contents

This page is intentionally left blank.

# Chapter 1

# Introduction

This report presents background, design, implementation, rationale and some use cases for Heapy version 0.1, a toolset and library for the Python programming language [3] providing object and heap memory sizing, profiling and other analysis.

The general aim of Heapy is to support debugging and optimization regarding memory related issues in Python programs. Such issues can make a program use too much memory, making it slow by itself as well as slowing down an entire computer, if it can run at all. These problems can occur especially in long running applications or when memory is extra limited such as in embedded systems.

## 1.1 Background

Although Python provides garbage collection (a general term for "algorithms for automatic dynamic memory management" [16]), which can reduce memory usage by deallocating objects when they can no longer be used in the program, there is a limit to what can be done by such general algorithms during program execution. The program still needs to be written so that unused objects may be removed by the garbage collector and so that the remaining objects are memory efficient.

Although programs could perhaps in theory be written to be memory efficient enough, based just on some static design criteria, it could take extraordinary efforts and would be especially hard when independent packages are combined to form complex systems. In practice, we need to be able to look into running programs to find out about problems, their causes and possible solutions. To do this, however, requires some support from the programming language, either directly or via add on modules. The primary motivation for Heapy is that there has been a lack of such support in Python.

The question is, what is the information needed and how is it best presented. There are some extreme alternatives. One alternative would be to make a raw dump of the memory contents. This would give all possible information about the heap content (except information that could only be collected by special run time instrumentation) but would usually be too much

to manually look through. Another extreme variant would be to have such an advanced analysis system that it could present exact advice on what to change in the program, for example as a patch list. Though perhaps possible under some circumstances, such a system is not within reach within the current development time constraint.

## 1.2 Problem definition

It is probably not practically possible to generate a single view of memory usage that will always provide the programmer with all information needed. For best results, the programmer needs to be involved interactively to select views based on previously retrieved information. The question is still, what kinds of information to provide. One way to define the problem could be to try to answer three general questions:

1. How much memory is used by an object?

   This is a basic information point, which is not available directly from Python. It may be taken to mean either the individual memory size of a separate object or the memory used by an object and its 'subobjects' in some sense; for example the objects that would be freed as soon as a root object is freed.

2. What are the interesting objects?

   In general, all objects in memory may be of interest, except those that are only used by Heapy itself. Of special interest may be such objects that are no longer of use to the program ('leaking' objects) or objects that use a major part of memory and may be optimized or removed, perhaps to be recreated on demand.

3. Why are objects in memory?

   If there is no good reason why some objects are retained in memory, they should better be removed. If an object is retained for no good reason, there must still be a reason albeit a bad one. This may be that there is a reference chain to the object so the garbage collector can not remove it. Such reference chains may indicate how to update the code so that the object may be freed.

## 1.3 Design approach

How Heapy addresses these 3 questions is summarized as follows.

1. Size of individual objects is provided by C routines for builtin standard types as well as user types defined in Python. Non-standard extension module types can be sized via function tables. Size of composite objects including subobjects is provided by a special algorithm.

2. It seems difficult or impossible to find all allocated objects in memory, separated out from objects used only by the analysis library. In practice, it should often be sufficient to find the objects reachable from a root object -- these are the objects that can actually be created and used by a Python program. Heapy provides a method to find these objects separated out from analysis-only objects. There is also a method to find some 'unreachable' objects which is useful in special cases involving extension modules.

   An assumption can then be made that interesting objects tend to have something in common. Objects that are allocated after some point in time may be 'leaked' objects, and can be extracted by comparing to a reference set. Interesting objects may be of a certain type or be referred to in certain ways. Heapy provides a variety of ways to classify objects. The result of classification can be presented as a table where each row summarizes data for one kind of objects. Often the interesting objects are those that use most memory; these are in the first row of such a table.

3. Reference chains may be presented in the form of the shortest paths from a root, where each edge indicates the relationship between two objects. To accomplish this, special functions in the C library are used to find out how a parent object of a particular type is referring to a child object, an operation also called 'retainer edge classification'. Another aspect of the reference graph may be presented in the form of a 'reference pattern', summarizing a reference graph by treating objects of one kind as a unit. The definition of which objects are of the same kind is provided by a suitable choice of classifier.

The problem situation also has other aspects such as:

- How is the system to be used
- How portable to be across platforms and versions
- How is it to be released

Much of the functionality is provided directly as a library. It can thus be used from a program as well as from the interactive Python command line. In the easiest case, the library can be used directly from within the application thread. For more advanced usage, there is support to have a separate thread used only for analysis purposes, which uses a special technique to provide a cleaner separation between application data and analysis data. A monitor program can be used to communicate with several analysed processes.

To separate the implementation from the API (application programming interface) some general concepts are used. A session context object provides an environment containing top level methods and data. A universal set abstraction combines object sets and symbolic sets (kinds and equivalence relations) with common algebraic set operations. Object sets provide a number of attributes that present their content from different perspectives. Such attributes provide the information to help answer the previously stated HOW, WHAT and WHY questions.

The presentation is generally text based and designed to provide practical accessibility in most situations (even Emacs buffers); showing only a limited number of rows at a time, without depending on an external pager. There is also a GUI (graphical user interface) based browser that can show a time sequence of classified heap data sets as a graph together with a table detailing the data at a specific time or the difference between two points in time.

A constraint for the current design was that it should be possible to use an unmodified Python interpreter. Though it excludes some information possibilities, it makes the system more generally usable. The current version can be used with an unmodified C Python, back to version 2.3. It can not be used with Jython or other non-C Python versions. It requires Tkinter if the graphical browser is to be used.

Heapy is released under an Open Source licence (MIT) and is available for download via the Guppy-PE homepage [1]. Guppy-PE is a project serving as an umbrella for some new and experimental programming environment related tools.

## 1.4 Related work

There are many memory profiling and debugging systems for languages other than Python, and some of these are referred to in Chapter 2. Two tools for Python have recently come to my attention. PySizer [4] is a "memory usage profiler for Python code. ... The idea is to take a snapshot of memory use at some time, and then use the functions of the profiler to find information about it.". The Python Memory Validator [5] "provides automatic memory usage analysis of applications as they run ... with information provided by the Python Profiling API.". Although both of these systems seem to provide interesting concepts, they are too recent to have influenced the current design and I will not speculate about their relative merits in this report.

## 1.5 Scope of this report

Chapter 2. provides background information, with references to existing work on ways of finding memory leaks, on various kinds of memory profiling, and on such things as *paths from root* and *reference patterns* to tell why objects are kept in memory.

Chapter 3. dives into Python internals to address some of the technical issues concerning how to get information about object memory sizes and object relationships, and how to find the objects in the heap that belong to the application, separated out from the objects that are used only for analysis purposes.

Chapter 4. is a design and implementation overview and rationale.

Chapter 5. is a tour with examples to introduce the basic concepts of sets, kinds and equivalence relations. It concludes with an example of how to get an overview of memory usage which in this case happens to reveal an optimization possibility involving more than

a thousand objects.

Chapter 6. is an example showing how to find and seal a memory leak in a GUI program which in this case revealed (arguably) a bug in a library widget.

Chapter 7. is the conclusion, discussion and suggested future work.

# Chapter 2

# Background

This chapter gives some background to the WHAT and WHY questions:

- WHAT are the interesting objects
- WHY are objects in memory

## 2.1 Memory leaks

One possible refinement of the WHAT question, is to say that we are interested in objects kept in memory due to 'memory leaks'. A memory leak can be defined as follows.

> Memory is *leaked* when an allocated object is not freed, but is never used again. The central question for a memory leak detector is: Given a time $t$ in the run of a program, and an object $o$, have object $o$ been *leaked*? It is generally impossible to determine at time $t$. ([15], as quoted in [14] p. 3)

Though the quote is concerned with compiled languages such as C, it applies to Python programs as well. Python provides memory management using garbage collection, a general term for "algorithms for automatic dynamic memory management" [16]. Common for such algorithms is that they implement what can be efficiently done during program execution. Generally, such a system works by automaticallly reclaiming the storage of objects as it becomes certain that they can not longer be used by the program, i.e. when there is no reference to the object. But there is no way for such an algorithm to tell beforehand, which of the still accessible objects will actually not be used in the future.

It may be possible to find leaked objects automatically by running the program to completion and keep track of when objects are actually used. Although it is out of the scope for the current report, as it would require modifying the Python interpreter which as mentioned in the introduction was to be avoided, it could be an interesting future development. An example is the heap profiler described in [19]. Memory for an object is classified as *lag* from

the time of its allocation to the first use of the object. The *drag* period is the time after last use until the object actually may be deallocated by the GC. Finally, memory can be classified as *void*, which means that an object was allocated that was never used at all. Of these classifications, the *drag* or *void* periods tend to be the most informative because they are associated with leaking objects. Although the time from last use could be of interest by itself, a heap profile that records lag, drag and void periods can not in general be presented continuosly but only after the program has run to completion. The implementation requires special instrumentation of the virtual machine, to record the time for each object use.

Though it could be effective, such an automated leak detection scheme could fail to find leaks that should nevertheless be consider actual leaks. Although the quoted definition of a memory leak is clear, it depends on the meaning of the word 'use'. Suppose for example there is a tree implementing some sorted data structure. Perhaps we actually 'use' only some of the objects but have to walk through a majority of the objects to find those objects actually used. The automatic leak detector would record uses of the objects we are just walking past. So they would not be reported as leaks. But in reality, they could be removed since they are not used for any other purpose than just walking through to find the objects actually used by the application.

Apart from automated leak detection schemes, there are also schemes that require interaction with the user, who is required to have some idea about what may be leaking, and when it may happen. For example, one such scheme is described as follows:

> Our approach to memory leaks is based on the observation that many memory leaks occur during well-defined operations which are supposed to release all of their temporary objects upon completion. If we let the programmer tell us the boundaries in time of such an operation, we can use this information to greatly simplify the discovery and diagnosis of memory leaks. [17]

The system presented in [17] lets the programmer define a "critical section" where the leak is believed to occur. The system can then take a snapshot of the heap population at the beginning and end of the critical section. Using this information, it can determine what objects were allocated during this section and were not deallocated after it.

It is noted that not all memory leaks are of the simple kind being so easy to detect. The objects found may be a mix of leaking and legitimatively retained objects. However, the method is said to nevertheless often be effective — the leaking objects could be identified by other means, such as via the reference pattern.

A similar approach to memory leak detection is used in KBDB [21], a heap inspector for Scheme programs. Another system, OptimizeIt for Java, uses a similar scheme as described in [9]. The systems seem to use quite different technical implementations, because of different underlying memory allocation schemes and varying compromises between implementation effort, time & space requirements and observation accuracy.

## 2.2 Memory profiling

To find out where to look for memory leaks, we generally need some overview of memory usage to start with. It should be possible to look at the memory usage of different groups of objects, to see what kinds of objects are most worthwhile to optimize or get rid of. It should be possible to look at memory usage as it evolves with time, to verify that it stays constant or find the places where it increases. Obtaining such overall information is often called heap profiling or memory profiling.

Though the information may be grouped by any suitable criterium, it is common to start with a classification based on 'type' or 'class'. These words have specific meanings in different programming languages. In Python, I think the situation may be described as follows.

Each Python object in the C implementation has a field, ob_type, which points to the type of the object. The type is also a Python object; it has the type 'type'. It is the type object that is returned by the type() function. Each object also has a __class__ attribute (as viewed from a Python program). It is often the same as the type of the object, but there is one exception. There is a special type, for which __class__ differs from the type. Objects of this type, InstanceType, have a __class__ attribute which is a user defined class, a so called 'old style' class.

When not talking specifically about a Python type or class, but about some classification in general, it would be confusing to say type or class and I will rather use the word 'kind'.

In summary, this is how these words should be used in Heapy.

- The type is the result of the type() function.
- The class is the value of the __class__ attribute. It is identical to the type unless the type is InstanceType; then the class is a user defined 'old style' class object.
- The word kind means the result of some classification in general.

There are a variety of heap profilers described in the published literature. Some depend on tracking usage patterns and would require quite deep modifications to the Python core, like the one already mentioned for Haskell [19]; there is a somewhat similar one for Java described in [22]. Some other classification schemes may be described as follows (paraphrased from [19, p. 1]):

- A *producer profile* classifies cells by the program components that created them.
- 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 cells.
- A *lifetime profile* classifies cells by the cells' eventual lifetime.

Of these, the producer profile would require special instrumentation of the Python interpreter to record the call stack when allocating objects, and is therefore outside the current scope of Heapy. Such modifications have however been implemented in PySizer [4].

The constructor profile seems in Python to be equivalent to a classification by class or type.

Two kinds of retainer profiles have been implemented in Heapy: one based on the set of retainer classifications (using for example the Rcs classifier), and one based on the set of edges from retainers (the Via classifier). Such information may be expensive to measure and takes some extra work to implement. But it may be worthwhile, as is noted in the following conclusion:

> Sometimes [memory faults or leaks] can be found by careful scrutiny of source code in the light of static heap profiles showing cell producers and constructions. But heap profiling by *retainer* so often points directly to an offending function that it seems well-worth the extra implementation effort required. [20, p. 618]

The paper also mentions a number of tweaks to make the retainer profile more informative, basically by not including all possible retainers so as to provide aggregation. This is, in principle, implemented in Heapy by providing a free choice of retainer classifiers.

Finally, the lifetime profiler classifies objects using information from the future, but it should be possible to implement it for Python even though the results would not be available continuosly. For best result it would probably require a modified Python core with hooks into the allocation and deallocation functions.

## 2.3 The WHY question

Having found some leaking objects is just a first step. We need to somehow find out why they are leaking and what can be done about it. One common method is based on the referrer objects. These are the objects that directly refer to the leaking objects. Investigating the referrers can help tell why the leaking objects are retained. If not the case becomes clear directly, such as when using a retainer profile mentioned previously, finding the referrers of the referrers again and so on may ultimately show the required picture. One may wonder how the referrers may be found in the first place - they are not available directly since they are normally of no use to a program.

In KBDB [21], each object is equipped with an extra slot, which contains a back pointer. There is only one back pointer, and if there are more than one referrer just one (arbitrary) referrer will be stored. The disadvantage of this approach seems to be the extra space and code needed for the back pointer, and that it is still not handling all referrers. An advantage of the approach is that it is fast to follow the pointers, so that long chains of referrers can potentially be built quickly.

In Python, there is a function `get_referrers` that finds the referrers of a set of objects. This function searches all possible referrer objects in the heap to find those that refer to the

objects in question. The disadvantage of this is that it can be slow especially when there are many objects in the heap. The advantage of the approach is that no extra space is taken in the heap for back pointers, and that the function can be built into a standard GC-supporting Python interpreter.

Since a constraint for Heapy (as mentioned in the introduction, at least in the initial version) was to not require a special Python runtime, it was not possible to add an extra field to the object implementation. To speed up finding the referrers of objects (especially to avoid quadratic time complexity for retainer profiling), a separate table is used which is initialized in a single heap traversal.

The referrer information may be made use of in various ways. In addition to the retainer profiling mentioned previously, the literature reviewed also mentions

*holder chain links*, *path to root*, *reference graphs*, *reduced reference graphs* and *reference patterns*.

Holder chain links are described in [21] as "representing the pointers chain from the holder to the inspected object". This concept seems similar to path to root analysis. With this kind of analysis, it is possible to calculate:

> **a single root** Only a single garbage collector root will be found. When searching for a memory leak, this option is often appropriate since any path to a garbage collector root will prevent the instance from being garbage collected.
>
> **up to a certain number of roots** A specified maximum number of roots will be found and displayed. If a single root is not sufficient, try displaying one root more at a time until you get a useful result.
>
> **all roots** All paths to garbage collector roots will be found and displayed. This analysis takes much longer than the single root option and can use a lot of memory. *(Quoted from [12, p.126].)*

From the description, it seems a bit unclear for the single root case, if only a single path is found, since there can be many paths to a single root. However, being simple, I take it that the single root implies a single path as well. One may furthermore wonder, how long it would take to generate the "all roots" case, if taken to mean all paths, in the typical or worst case. It is also an open question, how often a single path is really appropriate. From the amount of other analysis methods found in the literature, one may suspect that the situation may often be of a more complex nature.

An alternative could be to display the reference graph itself; as described in [12]:

> The reference graph shows the incoming and outgoing references of all instances of classes and arrays which are contained in the current object set.

## 2.4 Managing complexity

One may suspect that manually looking through the reference graph will become overwhelming in complex situations. A better approach is perhaps to use the "reduced reference graph", as described in [8]:

> Reduced reference graph provides a transitive closure of the full reference graph, to display only references that should be removed in order to free the object for garbage collection.

It seems unclear what this quote means, since the transitive closure would have much more edges than the original reference graph, but I note it anyway for reference. There is a picture of a reduced reference graph in [9] and a description that may provide some clues to how it is generated.

A *reference pattern* [17] seems to be another way to reduce the complexity of the reference graph. It is noted in the cited paper that a memory leak often consists of a large number of interrelated objects. It is also noted that "most programs with large data spaces have much repetition in their data structures". It is often possible to take advantage of this repetition to generate and display a reference pattern, which is smaller and simpler than the reference graph itself. In the reference pattern, objects from the original reference graph are combined to form larger clusters, based on a combination of their type and the clusters they refer to. Using the reference pattern, it is claimed that the programmer can often get enough information about the context of the leaking object, even in complex situations, to be able to determine a program change that may improve the situation.

A way to further reduce the complexity of a reference pattern may be to use the concept of *immediate dominators* [7]. The immediate dominators is a subset of the referrers, where only the referrers that can not be reached via any other referrer are included. When building the reference pattern, the hypothesis is that using the immediate dominators instead of the (immediate) referrers, could reduce the number of nodes and edges, without removing essential information. Some experiments have verified this in Heapy, though it may be hard to prove in general.

The main features implemented in Heapy to handle the WHY objective are *paths from root* and *reference pattern*.

# Chapter 3

# Diving into Python Internals

## 3.1 Python

There are a number of different Python implementations. This report is concerned primarily with the variant written in C, sometimes called CPython.

A program written in Python does not have access to all internal data of the underlying interpreter. This protects it from accidentally crashing but it can also not get all information that would be useful for memory debugging and profiling. To find this information we may use C code, compiled into an extension module. Extension modules are library modules that can be dynamically loaded into Python, and used from Python programs much as if they were actual Python programs. Extension modules are quite common and well supported. Care is taken by the Python core developers to maintain compatible interfaces for different revisions of CPython.

Information about object sizes can be provided from C. Each object contains a type field. The type may contain enough information so that the size of the object can be calculated. However, this is not always the case.

For example, the list type allocates an extra memory area so that it can expand dynamically. It does not indicate the size of this area in a standard way. One will have to write a special function to figure out how much memory is allocated to a list object.

While it is possible to write such functions for the predefined types, there is a special problem with objects defined in extension modules.

Extension modules often define their own types. Though they must be compatible with Python types, they can allocate memory freely. It is then not generally known how much memory such a type allocates. It is however possible to write special functions to calculate this.

In addition to the object sizes, it is also useful to know about their relationships. An object can contain pointers to other objects. In the current CPython implementation, such objects with pointers often need to participate in cyclic garbage collection. They must provide a method to traverse all objects they refer to. This traversal function can be used for

informational purposes. However, just knowing what objects are referred to does not tell the whole story. The missing information is what way the object is referred. This can tell the name of the attribute to access it with, or from lists or dicts what index to index it with. Such information is useful to see why objects are kept in memory. This information must be provided by special functions. Some of these could in principle be written in Python, but currently they are all written in C since it was more regular and faster.

## 3.2 Finding objects

This is about the objective: What are the interesting objects.

We don't know yet so much about what is 'interesting'. For a start, we may ask a simpler question. The question is simply put, so it should reasonably be possible to ask it and answer. The simple question may be put so: 'What are the objects in memory?'.

Using a Python program to ask questions about a running Python program has some tricky considerations. One is that the program used for testing may itself be allocating objects. Almost nothing is done in Python without allocating objects. A simple counter will allocate a new integer object each time it is incremented (unless the value is among the some 100 preallocated integers).

One would however want to be able to use a Python library to ask questions about an application, without changing the reported memory situation of the application. Otherwise it would be disturbingly difficult to compare the memory situation at two different times, to see what has changed. We want to see clearly that nothing has changed. But if the Python library used for analysis keeps adding objects, it would disturb the measurements.

So when asking this simple question: 'What are the objects in memory?', it is actually already modified. It does not mean that we really want to see all objects in memory, rather it means the objects for the particular application. Or somewhat equivalently, if there is only one application, we would like to see all objects in memory, except those objects that are allocated only for asking the questions and reporting the results.

It is such a separation requirement, that makes it tricky to find the relevant objects in memory.

An obvious variant would be to associate some label objects, that could tell that it was not to be counted as an application object. But this seems hard to do using a standard Python interpreter:

- It would require something to be done for each and every object allocation, when done from the context of the analysis library. But to do this automatically would require it used a special interpreter. This would require a modified Python core, or perhaps it could be done with some special tricks such as rewriting the byte code of all the modules used by the analysis library. It seems complicated anyway, compared to the tricks actually used described later on.

Is it so important to not require a modified Python? It will make the Heapy system more generally usable. And it will be easier to maintain the system, since it doesn't include distributing updated Python cores. And though it might be possible to have the Standard Python distribution include such a modification, it might be difficult to get it accepted, since if for nothing else, it might add some nanoseconds to the object allocation time, even when not used. And it would take some years until such a version became universally used. On the other hand, despite the problems there could also be advantages with such an approach, it might for example actually help making the system more universally used once the initial obstacles had been overcome. Perhaps this is a strategy to consider for future versions.

But in any case, let us see what can be done with an unmodified Python.

A variant that may seem obvious is: Transfer the information of objects to a completely separate process. There, all the analysis may take place without any risk of disturbing the analysed process. An approach like this is taken in Jinsight [17]. The contents of the application process heap may be dumped, a 'snapshot' taken and stored in a file. A separate process may then load these snapshots and analyse them and compare snapshots taken at different times. This is a possibility but:

- Wouldn't we still want to have the possibility to get memory related information, from *within* the Python running the program itself? And wouldn't we want to be able to ask about what objects are currently allocated? It seems that *only* being able to do it from a completely separate process, would be to give up on obviously desired functionality. So, if we can do it in the running Python process, it would be extra (double if not more) work if we would need to do it in some completely different way in a separate process.
- Dumping the memory would also take some program, depending on the form. This program could also be disturbing the application. So we have in any case required special means to be taken to not disturb the application.
- Dumping all the objects may take a lot of space and be slow.

There may be other reasons to dump data than just isolation from application. One such reason may be to have a record to go back to. But such data may not necessarily contain complete information about all objects, it would often suffice with a statistical summary.

To go back to the current problem, which was to do with the isolation.

We want to be working with some analysis library in Python, in the same process as the application, and not having to dump all objects from memory. Then the system could be used like an internal debugger without having to start an external process.

It seems it should be possible, by structuring the data of the analysis library in a certain way, to have it tell its own data from those of the application. It could put some labels on some of its data to tell that it belongs to itself. (It would be hard to put labels on all data, however, since there is no place left in small objects like ints, and they are so often allocated so it would be a pain to do something extra each time such an objects is allocated.)

To have such a scheme work, it will take some modification of what is regarded to be the contents of the memory. What are the objects in memory? It turns out, the way we find those objects, has a crucial impact on the possibility of sorting them out from the objects of the analyser itself.

We can idealise the situation. We imagine that there is a root, consisting of one object. All other objects in memory can be reached, directly or indirectly, from the root, via pointers from one object to another. It is possible to walk through such a graph to find all objects. (I sometimes call this a heap traversal.) It is then possible to look at some special labels that some objects may have. Such labels have been put there by the analysis library on some of the objects that belong to it. It is then possible for the heap traversal to stop at such objects, and not continue traversing the objects they refer to. If the only incoming objects, from other objects, have been marked, it is possible to exclude all data that belong to the analysis library.

This works in many situations. The requirement is:

- The user doing an analysis only creates objects via the analysis library. All new objects stored (eg in the main module) are of such a special labeled kind.

This will exclude many objects from blurring the sight of the view of the heap. This scheme is implemented in Heapy and it makes it possible to use it directly as a library, from the interactive command prompt or from test programs. The heap view reported is not disturbed (much if at all) by objects used internally in the Heapy library. This may sometimes be good enough, but there are also cases where this scheme fails:

- What if the user does something that requires him to create his own objects, be it lists or whatever, to support the analysis. These are not marked and they may be reached without passing any specially marked objects. So they will be counted when the heap is measured.
- What if the user is at the interactive prompt. Each time a command is given it is compiled by Python to a bytecode form. This will allocate some objects and then some objects are needed to execute the bytecode. Though these objects will disappear as soon the command has been executed, they will be allocated while the command is being executed, and will be found and reported by a command to find the objects in the heap.

Though there may be means to get around such problems on a case by case basis, there is a general solution which gives us a chance to dive into Python internals again.

## 3.3 Interpreter structures

In the bottom of Python internal data, there is an interpreter structure. Usually only one is used but new may be created. It is possible to start a new thread using a new interpreter.

There is no support to do this from Python, but it is possible from C and Heapy added a gateway to do it from Python. Such a new interpreter will:

- Have separate module storage. Variables stored in eg __main__ in one interpreter is stored in separate data. (However, extension module data is shared, it is of no importance here. The Heapy analysis library doesn't use any global variables in extension or otherwise modules.)
- Share a common physical address space. So pointers can point between objects allocated from different interpreters.
- Share common data related to garbage collection and memory allocation.
- Have separate standard input and output files. So a print statement in a new interpreter can be directed to go to another output file, for example a socket.
- Objects when allocated are not in general tied to a particular interpreter.
- Each interpreter has a separate list of threads. The threads point back to the interpreter. This is used to get access to the right module storage and other data local to the interpreter structure.
- Each thread has a list of active frames. A frame is allocated each function call and deallocated when it returns. Frames are used to store local variables.

The idea then, is to run the analysis library in an interpreter structure separate from the application being analysed. If the user can work in this separate structure, its data can be separated:

- The objects are only stored in modules of the analysing interpreter, or in a frame belonging to a thread of the analysing interpreter.
- Care is taken not to introduce pointers from the analysed program, back into data of the analysing session. If there really must be such a pointer, it must be isolated by means of going through an object marked to belong to the analysing library session.

The result of 'objects in memory' can then exclude the objects in the analysis session, with a suitable choice of 'root'. The root should be taken to include the interpreter structure of the application, but exclude the analysing interpreter. The traversal will thus find objects only in the application. If there are any back-pointers, since they must be labeled they are not followed.

## 3.4 Simplified assumption

The underlying assumption is that 'all objects in memory' can be found via a traversal from some root objects (ie interpreter structures). Though the assumption is often good enough, there are cases where it is false and it may make an important difference. Thus we will again dive into Python internals.

## 3.5 The memory allocation in Python

Combining reference counting and cycle detecting garbage collection.

Each object has a reference counting field. This counts how many references there are to the object. It must be incremented (by C code) when creating a new reference to the object, and decremented when the reference goes away. When it reaches zero, the object is deallocated. This works fine in many cases, and one advantage over other schemes is that objects get deallocated soon after the last reference disappears, so the memory can be reused quickly. This is good for cache locality.

The standard problem with reference counting (see for example [16]) has to do with cyclic references. The reference counts of the objects involved in cycles will never reach zero, so the objects will be kept in memory for ever, even if there are no references from outside.

This was quite a problem in earlier versions of Python. Programs had often to be severily contrived to avoid cycles. Luckily, a solution was found. There is now a cycle detection garbage collector, in addition to the reference counting scheme. It will periodically (after some large number of objects have been allocated, and not deallocated again) search through the allocated objects to find groups of cyclic objects that can be deallocated.

To find the objects that may be involved in cycles, they must have been registered in a special way. When such an object is allocated, it is inserted in a global list. Most kinds of objects that contain any pointers to other objects may be involved in cycles and are thus registered. (But not all, for example objects of code type have pointers to constant objects but they can not be cyclic, so they don't need to participate in cyclic GC and can save the overhead involved.)

The garbage collection, be it by reference count or cycle detection, is thus NOT based on a single (or small set of) root objects. This breaks the assumption previously made for Heapy purposes, that the objects in memory could be found by traversal from a root.

For an object to be usable at all, there must of course be some kind of reference to it. But this reference may take whatever form imaginable. References may be hidden in any kind of C data structures, in global data in extension modules and any kind of external libraries.

## 3.6 No general procedure

This means it seems impossible to find all allocated objects with a general procedure.(At least without going further into low level special libraries such as asking the memory allocator

itself; this being intricate and less portable.)

At least those objects that may participate in cycles, must have been registered so they can be found in the list maintaned by the garbage collector. And some other objects, though not registered and not reachable from a root, may be found by following references from the registered objects. Although there is no guarantee to find all objects, this is what can currently be done in practice.

But the problem of using such an approach to find the objects in memory, is that it breaks the previously mentioned isolation of application objects from objects used by the analysis library. Since the objects are not found by following pointers from a root, but rather in a single list containing all objects, the special labeled analysis objects can not be used to isolate unlabeled analysis objects from the view. It would require all analysis objects be labeled which would be practically impossible as mentioned before.

## 3.7 The compromise

The solution attempted in Heapy is a compromise. It considers most objects of importance to be indeed reachable from a single root. Only in special cases we need to look at other, unreachable objects. It is then assumed, that the analysis library itself does not allocate unreachable objects that would blur the picture. Unreachable objects can then be found from the global GC list of objects, and the objects they reference, and separated out by the property of not being reachable from the root. However, care must be taken to not make them reachable again, since they can not then be separated out anymore. If they are not separated out, they would include both application objects and objects used only for analysis, which would blur the picture because only the application objects are of interest.

For analysing object usage of Python programs, it should usually be sufficient to look at reachable objects only. But in some cases it is desirable to look for unreachable objects, typically to find special kinds of memory leaks involving extension modules. It is possible to get data about unreachable objects, via a special method (heapu).

# Chapter 4

# Design, implementation and rationale

## 4.1 General Concepts

Heapy is built from the following main components or concepts:

- Session context -- Top level methods and data.
- Universal set -- Different but compatible set representations.
- Set of objects -- Standard object container.
- Kind objects -- Symbolic sets, often an equivalence class.
- Equivalence relation -- Defines how to classify and partition.
- Path -- A list of graph edges showing how objects are referred, used to tell how to fix memory leaks.
- Reference pattern -- A condensed reference graph, used to tell how to fix memory leaks.
- Remote monitor -- Separates the debugging process from the target process.
- Memory profiling -- Collecting a series of samples of memory usage by different kinds of objects.
- Profile browser -- Used to browse a graphically displayed memory profile to find problematic events and trends.

### 4.1.1 Session context

A Heapy session context is an object that provides the top level Heapy methods, and contains data for configuration and other purposes such as caching common operations and storing reference heap populations. It avoids using global variables. To use Heapy, a session context

object must first be created. Normally only one session context needs to be used but there can in principle be any number of them. A Heapy session context is an instance of a general session context abstraction which is provided via the Guppy Glue system.

### 4.1.2 Universal set

Another important kind of objects provided by Heapy is based on a universal set abstraction, of which there are two main variants: Sets containing actual objects and symbolic sets representing kinds of objects.

The word kind is used here to avoid the confusion likely to occur when using words like type and class, because these words are often used with narrowly established meanings in programming languages in general and in Python in particular.

### 4.1.3 Set of objects

Methods, such as heap() and iso(), can be used to create sets of objects, formally of kind IdentitySet. Such an object set acts as a proxy to a collection of objects and provides memory and structure information in a standard way. The sets are using address based object identity which means to distinguish objects by address and to handle all kinds of objects uniformly. Heapy object sets provide a number of operations such as:

- Set union, intersection, difference, complement.
- Informational attributes such as count, size, kind.
- Structural analysis attributes such as the dominated set, the shortest referring paths and the reference pattern.
- Representation as a table partitioned into rows by classification according to a default or chosen equivalence relation.

### 4.1.4 Kind object

A special type of set represents a 'kind'. Such a Kind object represents a set symbolically. It does not contain any actual element objects, but it has a method to tell if an object is a member of the set. The usual set theoretical operations are available for such Kind objects and in combination with object sets. Using such operations, objects of a particular kind, or a combination of kinds, may be extracted from an object set.

It is also possible to associate a particular Kind to an object or set of objects, according to some testable rule. Doing such a thing is called classification. Classification is useful in particular since it can be used to split, or partition, a set in subsets of different kind. For example, in Heapy this may be used to see what kinds of objects in the heap contribute the most to memory usage.

Different classification schemes give information that is useful in different situations. In fact, inventing new classification schemes seems to be an important way to extend Heapy to provide new kinds of useful information. It is then useful to not restrict the classification information to the attributes of the object itself, but also include information not usually associated with an object. Such information can be, how an object is referred to from its referrers (the Via classifier) and what kinds of referrers it is referred from (the Rcs classifier). Such a classification generates the kind of memory profile called a *retainer profile* in Chapter 2 and in [20].

### 4.1.5 Equivalence relation

Heapy provides a number of predefined classifiers. They are wrapped in top level objects representing equivalence relations. An equivalence relation defines a classifier and vice versa. The two terms may be used quite interchangeably. An equivalence relation may be represented mathematically as a set of equivalent pairs, obeying certain laws. It may be shown that the intersection of two equivalence relations is also an equivalence relation. (And that the union of two equivalence relations is NOT always an equivalence relation.) Such an intersection operation is supported by Heapy equivalence relation objects and it can be used to create new equivalence relations that can classify according to two or more combined criteria.

More about sets, kinds and equivalence relations is explained, with examples, in Chapter 5. For the formal description of these objects, see Appendix A.

There are also other objects that are useful for representing information about the relation of objects in the heap: shortest paths and reference patterns. These are the values of certain attributes or methods of object sets.

### 4.1.6 Path

A Path object represents a way to walk through the objects in the heap, from an initial object to a final object, not visiting the same object twice. The initial object is typically a designated root object, from which all other objects can be reached (if they can be reached at all). A path may be presented to the user in the form of a list of attribute names, indexing operations and some operations representing special C-only internal attributes. In case no special attributes are involved, such a path is a Python expression that can be evaluated to provide the final object from the initial one.

A path gives information about how an object is retained in memory, which may help explain why it has not been deleted despite it is not being used. There may in general be many (astronomical numbers of) paths to an object. These may be of different length, from the shortest ones to the very long ones that can involve many or all of the objects in the heap. Each path gives a different story, but having too many and long paths to look at would just blur the picture.

Among all possible paths, there is often a smaller number of paths, the shortest paths,

which more directly provide the desired information. An algorithm to find the shortest paths has been implemented in Heapy. The result is presented as a ShortestPaths object. This represents the shortest paths to an object although without necessarily generating them all. It turns out that the number of shortest paths, although often lower than the number of all paths, may be very high in some circumstances. It would then be practically impossible to actually generate those paths. The algorithm can however calculate the number of shortest paths, and present any particular path on demand. In many cases, it will suffice to look at one or a few paths. -- On the other hand, if the shortest paths do not give the required information, there are means to find longer paths.

An example of how shortest paths were used to find out how to fix a memory leak issue is in Chapter 6.

### 4.1.7 Reference pattern

Using reference patterns is another method, besides shortest paths, to get a picture of how objects are connected, to help find the cause of memory leaks. The term reference pattern is from [17]. The interested reader should consult this paper, where the background, motivation and algorithm is well described. The following is my attempt on a summary description.

The motivation is to have a way to get a picture of an object graph, when there are many objects involved, and it would be impractical to display the object graph itself; it would be too big and would not reveal useful information. The idea is that the references often follow particular patterns, which may be used to compress the graph when objects are classified and partitioned according to their kind (by some given classifier.) In this way, it may be possible to represent an entire set of references, say 1000 references from 1000 objects of kind A to 1000 objects of kind B, by a much smaller reference pattern, involving a single reference from a node representing the set of 1000 A objects to a node representing the set of 1000 B objects. Doing this systematically could reduce the original object graph to a smaller graph where each node represents a set of objects of the same kind.

The reference pattern algorithm starts with a given set of objects and finds its set of referrers, which it then partitions using a given classifier, and continues recursively until it reaches a given depth limit, or arrives to a set of objects it has already seen. The result is a graph of connected nodes, representing sets of objects. This is what is called the reference pattern and it may then be presented to the user as a graph or a spanning tree.

In the Heapy implementation, it has been found practical to have the option to reduce the number of referrers in each step by using the immediate dominators [7] which is a subset of the referrers and excludes referrers that are referred to by other direct referrers. There are also other Heapy-specific options but the basic reference pattern implementation is intended to follow the algorithm given in the paper, with reservation for any mistakes.

### 4.1.8 Remote monitor

The remote monitor makes it possible to monitor and debug Python programs with minimal interference. It uses a separate interpreter thread within the debugged program process. The monitor itself runs in a separate process and is able to have connections opened to several monitored processes. The monitored program can use unmodified original code, it just needs to do an initial call to enable the monitoring of itself which can be arranged separately from the program itself.

### 4.1.9 Memory profiling

It is possible to generate memory profile data by using Heapy methods to take snapshots, either at regular time intervals or at specific occasions. If there is a problem with memory usage, such as, in a long running application, that memory usage keeps increasing with time, such a memory profile can reveal what kinds of objects contribute to the increase in memory usage and at what time the increases happen.

To generate snapshots, there is a special attribute of object sets, .stat. This is a statistical summary of the set, containing the count and size for each kind of object in the set, according to the classifier in use. Such a data set may be appended to a file using the .dump() method.

The snapshots may be taken either from within the running application program, by adding snapshot-taking code, or independently from the application program, via a remote monitoring process. To generate and dump a snapshot, a call such as h.heap().dump(<filename>) may be sufficient. There is however a choice of data to dump, for example it may be useful to dump the difference from a starting point rather than the absolute value. The h.setref() method may be used to set a reference set of objects, or alternatively the statistical data may be subtracted from the data at a reference point. The latter method may give negative counts and sizes, whereas the first method will not.

### 4.1.10 Profile browser

To make the memory profile visible for analysis, the Heapy Profile Browser may be used. The profile browser provides an interactive graphical window. It has the following features:

- Showing a graph with total size or count of dominant kinds of objects, either as lines showing individual kinds or as bars stacked to show the cumulative sum.
- Showing a table detailing the information at a particular point in time, with a choice of comparisons between different points.
- Two movable markers for controlling what sample data the table is based on.
- Data is provided by a file and the display can be automatically updated periodically as data is added to the file.

Having used the profile browser to find out that there is a problem, such as an indefensible increase in memory use by some kind of objects at particular times, it remains only to find out the cause of the problem and how to fix it. Chapter 6 contains an example from real usage which involved finding and fixing a memory leak in the profile browser itself, which was found to be due to a feature (arguably a bug) of a Tkinter widget wrapper class.

## 4.2 Summary of API & UI

The API (Application Programming Interface), though designed to be suitable to use from analysis programs, also serves as the main UI (User Interface) to be used interactively from a Python console. The API is formally described in the documentation on the Guppy-PE homepage [1] and an extract is included in Appendix A. There are a variety of features, some have proven to be more useful, some are included for generality and may prove to be more useful in the future. To avoid the impression that the system would be complicated to use, the most commonly used features are summarized here.

### 4.2.1 Creating the Heapy Session Context.

```
>> from guppy import hpy
>> hp=hpy()
```

### 4.2.2 Commonly used operations on a session context

heap()

Find the objects in memory, return an IdentitySet. (Technically the reachable objects allocated after the optionally previously set reference point)

setref()

Set a reference point for heap().

iso()

Create an IdentitySet from some given objects.

pb()

Start the interactive graphical profile browser.

monitor()

Start the remote monitor.

A number of equivalence relations: Type, Size, Via etc.

These can be used for a number of purposes, in particular to create a Kind object (an equivalence class) or to create a combined equivalence relation to partition a set.

### 4.2.3 Common operations on IdentitySet objects.

- Standard representation in table format, with size, counts, and percentages of groups of objects, showing only the 10 largest groups at a time to provide practical accessibility even in a dumb terminal or an Emacs buffer.
- Indexing (or slicing) an IdentitySet selects the set of objects in a row (or range of rows) in the table representation.
- Standard set operations, with automatic conversion from Python types; for example (hp.heap() & str) is the set of all string objects in the heap.
- Some useful attributes:

  byid, byvia, bysize, bytype, byclass, byclodo etc.

  : These attributes each contains the same set of objects, but with different table representations.

  shpaths

  : Shortest referring paths.

  rp

  : Reference pattern.

  dominos

  : Dominated set of objects.

  dump('file')

  : Dump the table representation to the end of a file in a form that can be read by the profile browser.

## 4.3 Implementation overview

It is the intention that such details as where particular functions are implemented, should be of no concern for the user, who interacts only with the session context, object sets and some special purpose objects such as reference patterns and paths. The only import statement needed is 'from guppy import hpy', to import the session context constructor.

The system is technically implemented in several Python modules and two extension modules written in C. The following summarizes the overall structure.

guppy

: The top level umbrella package. For public Heapy API purpose, it contains the hpy constructor. For implementation purpose, it contains the following subpackages and modules:

heapy

A package containing Heapy specific modules.

Use.py

Public Heapy API.

UniSet.py

Universal set class with subclasses.

Classifiers.py

A variety of object classifiers.

Part.py

Represents a set as a partitioned table.

Paths.py

Shortest paths algorithm and presentation.

RefPat.py

Reference pattern algorithm and presentation.

View.py

Provides a particular heap view.

Prof.py

Browse memory profiles using a GUI.

Monitor.py

Monitor separate Python processes.

Remote.py

Enable a process to be remotely monitored.

heapyc.so

Provides C level object data and special algorithms.

etc

A package containing general support modules, in particular the following.

Glue.py

Provides a general session context mechanism.

sets

Special sets, provided by an extension module.

setsc.so

Bitsets and 'nodesets' implemented in C.

## 4.4 Extension modules

### 4.4.1 heapyc.so

The heapyc module provides the low level functionality implemented in C. It provides a main object type, HeapView, which provides the low level session context, with data (for example hiding options and configuration for extension types) and methods. In particular, it provides:

- Heap traversal methods: to find the objects reachable from a root, optionally avoiding some objects or kinds of objects.
- Methods to get the size and other data from objects of different types, supporting standard builtin types directly and nonstandard types via user supplied function tables. Apart from the size, another main information point is the referrer edge classification giving the attributes, keys or indices connecting two related objects.
- A method to find the 'owner' of each dict object in the heap.
- A method to find the referrers of objects, by using heap traversal so the referrers of all objects in the heap can be found in less than quadratic time (something like O(n(log(n))) with binary searching nodesets).
- A number of classifiers, using different kinds of object information, such as type, class, size or structural relationships such as owner or referrer classification, and referrer edge classifications.

### 4.4.2 setsc.so

This provides two kinds of sets implemented in C. Apart from the 'bitset', which is not directly used by Heapy, it provides a 'nodeset', in mutable or immutable variants. The objects are distinguished by address. The mutable nodeset is internally implemented by means of a mutable bitset whereas the immutable nodeset is represented as an array of objects sorted by address. The available operations include the usual set-theoretical variants as used also by Python builtin sets.

## 4.5 Python modules in guppy.heapy

### 4.5.1 Use.py

This module implements the interface to Heapy that is intended to be used and to be 'officially' supported. It is the Public API. Most methods and other objects are imported from other modules. (Some methods are defined directly in Use, typically historically for experimental purpose; they will or should in principle be moved to some other suitable module.) The Use module itself is intended to be used via a Guppy Glue session context; the standard one is provided by the hpy() constructor.

### 4.5.2 UniSet.py

This module contains the Universal Set base class (UniSet) which defines the operations common for all sets produced by Heapy. There are subclasses for symbolic sets (Kind), and identity based object sets (IdentitySet). Every UniSet instance has an attribute fam, an instance of a Family subclass. The family objects have access to the session context and provide most of the actual implementation. There is an IdentitySetFamily family used by IdentitySet, and a variety of family classes for different kinds of symbolic sets, for example AndFamily for sets representing the result of symbolic intersection. More families are provided by Classifiers.py.

### 4.5.3 Classifiers.py

This module implements different kinds of object classifiers. The classifiers wrap the low level classifier objects from the heapyc extension module. The wrapping layer provides conversion of low level classifications to high level Kind objects. A classifier also needs to supply information about itself, for example how it is to be named in a table header and how it is related to other classifiers. The classifiers themselves are wrapped in EquivalenceRelation objects providing the public API.

### 4.5.4 Part.py

This module implements the representation of object sets as tables, partitioned into rows of sets of objects of the same kind. Such a partition is used both for the default string representation of a set and to define the result of indexing and slicing. The partition, an instance of a subclass of Partition, is evaluated on demand, the first time a string representation or indexing is required, and memoized in a special attribute of the set. The partition objects are implemented the same for all classifiers except for the identity classifier. In this case, a preliminary partition is created using the size classifier to more efficiently handle large object sets.

### 4.5.5 Paths.py

This module implements a shortest paths algorithm, with options to generate possibly longer paths by avoiding certain objects or edges. It uses an inner loop implementation provided by heapyc. The algorithm is based on the Dijkstra algorithm ([11] as described in [10]) but simplified and speeded up because each edge is considered to represent the same distance. The shortest paths found are initially represented as a noncyclic graph. The string representation shows selected linearized paths using retainer edge classification as provided by heapyc.

### 4.5.6 RefPat.py

This module implements a reference pattern algorithm, freely after de Pauw and Sevitsky [17]. The reference pattern simplifies the reference graph by clustering objects of the same kind, according to a suitable equivalence relation. The result is a graph where the nodes represent sets of objects, and there is an edge from node A to node B if B is A's referrers of one kind. The graph is generated on demand and wrapped in an object of class ReferencePattern. It has a string representation as a spanning tree. It provides indexing and attribute access with a tree syntax to select individual nodes (sets of objects) for further investigation.

### 4.5.7 View.py

This module provides a first level wrapper around the low level heapyc module. It provides a particular view of the heap by means of hiding options and by definitions of extension module types, which it looks for in imported modules. It stores the memoization data used to speed up finding referrers in general and dict owners in particular. It provides the implementation of the heap() method making sure enough parts become imported so that the view of the heap will usually not contain data from new Heapy imports when comparing the heap to a reference point. It also implements algorithms such as the dominated set.

### 4.5.8 Prof.py

This module implements the Heapy Profile Browser. It has a GUI (Graphical User Interface) using the Python Tkinter module, wrapping underlying Tcl/Tk widgets. It reads data from a file in a format as generated by eg the IdentitySet.dump method and can update the display automatically as new data is appended to the file.

### 4.5.9 Monitor.py

This module implements the remote monitor. It uses a command line user interface which is designed to be portable, i.e. it does not depend on special terminal capabilities. Command line editing via the readline library works if available. The monitor depends on threading

and sockets. It works technically as a SocketServer.ThreadingTCPServer handling requests from processes that ask to be monitored.

### 4.5.10 Remote.py

This module implements remote control for a Python process, enabling an application to be monitored by the remote monitor. When enabled, a thread will keep trying to connect to the remote monitor via a socket. When connected, it can read commands from the other end and execute them in an interactive Python console. The thread uses a private Python interpreter state structure to isolate its data from those of the application program.

## 4.6 Python modules in guppy.etc

### 4.6.1 Glue.py

This module implements a general session context mechanism. It contains one public constructor, Root(), to create a new top level session context. This object acts as a proxy to all packages and modules in the installation. A package or module is imported on demand when the attribute with the corresponding name and place in the hierarchy is first accessed. A proxy is then created which provides session specific data as well as access to the underlying module. Context specific behaviour can be provided for a module via directives in a special class (_GLUECLAMP_).

## 4.7 Rationale

### 4.7.1 Why sets?

In Heapy most object collections are sets, and object kinds are also referred to as sets. Why is this, and could not some other abstraction be more useful?

The intent is to have a mathematical foundation for common operations and to provide these operations in an easily usable way.

There was a need for a general container for object collections. The set concept turned out to be a good fit since it provides standard operations for collections with at most one occurence of each object and when the order is of no significance. (The objects are ordered when presented, but that is outside the scope of the set abstraction.)

There was also a need to handle the results of object classifications. What can be done with such classifications, except printing them? How are different classifications related, especially when generated by different classifiers? What is a classification? It turns out a classification corresponds mathematically to a set, called an equivalence class. It is the set of all possible objects classified the same by a given classifier. Such a set must use a symbolic

representation, it can not be implemented as an actual collection of objects since it would be very big, if not infinite. It can still support the standard set operations. It can be used to extract or remove a certain kind of objects from a collection of objects. And relations between sets, such as the subset relation, can be used to clarify how different classifications are related.

There are already established Python operators for set operations. These operators are overloaded for Heapy sets. This makes expressions concice and easier to remember than if new names had to be defined for these operations.

Builtin Python sets can not be used because we need to:

- Handle all kinds of objects uniformly without relying on how they implement hashing and equality.
- Provide symbolic sets that don't contain actual objects but only represent kinds of objects.
- Provide Heapy specific attributes.

Why not use predicate functions rather than symbolic sets?

To select objects from a collection, a set intersection operator (A & B) can be used, where any operand may be a symbolic set. An alternative could be to use a function like filter. It takes a collection (a list) and a predicate (a function) to select some objects from the collection. But other methods are needed to select objects in other ways and to combine two functions into one. Using a set abstraction is more general and combines several such cases. It separates implementation from presentation.

Why not operations on objects directly?

There are no functions in Heapy that operate directly on unknown (application) objects provided by the user, except to make a set of it (the iso method), and all other operations are attributes or functions on sets. Would it not be practical to have more direct functions, for example to find the shortest paths to a specific object?

It would, technically, expose it in the frame it was evaluated in so that new paths were generated that could be hard to separate out from the paths that were really of interest. Keeping it in an IdentitySet keeps it isolated: the path through the set will not be reported.

Such considerations won't always apply, for example not for a sizeof function. But it would be redundant to provide both mechanisms. Since operations that return application objects return them wrapped in sets, it would be inconvenient to not be able to operate on such sets directly. So for this reason there should be a size operation on sets. Having also a sizeof operation that took a plain application object would be redundant and error prone since when passed a set, it would return the size of the set object itself, not the size of its contents - unless a special type check was done which could make it even more confusing.

### 4.7.2 Why equivalence relations?

Why not just classifiers? Classifiers correspond to equivalence relations. Why the need for both concepts?

Equivalence relation objects were introduced to provide a common public API, hiding the several methods of lower level classifier objects.

The word equivalence relation was chosen because it is a fundamental mathematical concept, defined in the textbook [13] and in the handbook [18] to be a reflexive, symmetric and transitive relation. The word classifier does not occur (in the index of) those books. I had to resort to Wikipedia [6] to find a definition:

> In mathematics, a classifier is a mapping from a (discrete or continuous) feature space X to a discrete set of labels Y.

A classifier $C : X \mapsto Y$ naturally generates an equivalence relation $R_C$ on $X$:

$$a \; R_C \; b \iff C(a) = C(b), \text{ for all } a, b \in X$$

That is, $a$ and $b$ are equivalent according to $R_C$, iff they are classified the same by $C$. The equivalence relation $R_C$ is *uniquely* determined by $C$.

On the other hand, given an equivalence relation $R$, there is no unique classifier defining $R$. The choice of mapping is arbitrary (or depends on implementation considerations) as long as the classifier defines the given equivalence relation $R$.

To hide this arbitrariness the top level API objects are called equivalence relations and they provide operations, such as set partitioning, that are independent of the arbitrary choice of classifier. The underlying set (X above) is not explicitly represented but could in principle be defined to include all possible views of memory as seen from all possible Python objects (a very big though not exactly infinite feature space).

There may still be a need for the result of classification to be available on the API level. There is one possible choice of classifier that seems less arbitrary than the others. An equivalence relation $R$ on a set $X$ determines mathematically a set of *equivalence classes*. For any element $a$ of $X$, the equivalence class is defined as:

$$[a]_R = \{x \in X | x \; R \; a\}$$

The set of equivalence classes is also called the quotient set of $X$ by $R$ and denoted $X/R$.

Given an equivalence relation $R$ on $X$ it should be possible to construct a classifier $C$, a mapping $C : X \mapsto Y$ where $Y$ is the set of equivalence classes $X/R$.

$$C(x) = [x]_R$$

The classifier is just the operation of creating the equivalence class corresponding to $x$. It is perhaps arguably not strictly a classifier according to the original definition; it depends on what is meant with $Y$ being a 'discrete set of labels' but it is a matter of definition.

The result of classification is in Heapy on the API level a Kind object, which is an object representing symbolically an equivalence class of the equivalence relation used for classification. It makes it possible to combine in a well defined manner the results of classifications made by different classifiers. In practice, a Kind is an object that can be used with standard set operations, in combination with any other Kind objects or IdentitySet objects.

Though the set theoretical operations of such Kind objects are uniquely defined by the represented equivalence class, there remains some seemingly unavoidable arbitrariness. Some text representation must be chosen for the unavoidable presentation, and it should preferrably uniquely identify each equivalence class. There seemed to be no choice but to involve the result of the underlying low level classification here. This was partly to do with that the representative objects themselves can not in general be uniquely and compactly represented.

### 4.7.3 What is this term Partition used when printing tables?

The word is used in the mathematical sense, not the hard disk sense.

> A partition of a set X is a set (P) of nonempty subsets of X such that every element x in X is in exactly one of these subsets. ...The elements of P are sometimes called the blocks of the partition. [6]

A partition of X is uniquely defined by an equivalence relation on X.

When an Heapy IdentitySet X is printed as a table each row corresponds to one block of the partition of X determined by the equivalence relation used by X (X.er) .

The formalism is mainly to make sure that the figures add up to 100 % -- no object will be forgotten or counted twice, regardless of which classification scheme is used!

### 4.7.4 Why session context - why not global variables?

Heapy is intended as a general library, and it uses internal data for configuration and state. It would be a bad idea to use global variables because it would restrict how Heapy could be used. There can be more than one Heapy session in use at a time. For example, a test function may create a local session context which exists only for a short time. It will use data isolated from other session contexts. After a session context is freed, its internal data may be freed as well, because they are not stored in global variables.

But why is session context data needed?

In general, there are data for configuration, reference points, and caching/memoizing common operations. The intent is to provide a useful & usable API by hiding complexity using

algorithms with internal data, but avoiding global variables. Using a session context is also a way to be prepared for future versions where new data may be needed.

- Configuration data includes flags to tell what part of the heap should be visible; how to handle special types; and how to present objects, for example the default maximum depth of reference pattern representations.
- Reference point data is generated by the setref() method which stores data representing the current heap population.
- Dict owner map. The word dict, short for dictionary, is the name of a Python type, which maps from keys to values, and is used throughout Python for dynamic storage of object attributes and many other purposes. Attributes are usually not stored in user objects directly, but in separate dict objects. The dict objects typically use a majority of memory. But since the type of the dict objects is the same, a type based memory profile would put them all in the same group. To tell more, we need some means to find out what classes different dicts belong to, their 'owners'.

  When classifying by dict owner, we need to find the owner of each dict object. The owner, if any, is an object that refers to the dict via a special internal attribute, often visible from Python as the __dict__ attribute. Given a dict object only, it is not known what its owner is, or if it has any owner at all. To find out, all potential owners in the heap would have to be searched. However, doing such a search each time a dict is to be classified, would be a waste of time, and could give a quadratic time complexity. The owners of all dict objects in the heap could as well be found in one search pass and then memoized. The session context contains such a memoized map, which maps from each dict object to either its owner or to None if it is a dict without owner. The dict owner map is created or updated automatically when needed. (It is freed when the session context is freed but may be cleared earlier, to avoid keeping references to dicts and their owners too long; it is cleared automatically at garbage collection time via a weakref callback.)
- Inverse reference graph, also called a retainer graph. The rationale is similar to the dict owner map: to speed up classification by avoiding quadratic complexity. The implementation is somewhat similar to the dict owner map but is kept separate because the retainer graph may be much bigger than the dict owner map and care must be taken to include only retainers from the visible application heap. The retainer graph is consulted especially by the Rcs or Via classifiers (to generate retainer profiles).

Why not a shorthand

```
import hp
hp.heap()
```

Such a module, hp, would have global variables allocated that may stay there for very long - until the program exits or the module is removed in some way. I don't know if it is desirable to have such behaviour. It seems more flexible and general to let the user decide where to put global variables.

But it is also somewhat longer to write. It is possible to make such a shorthand module quite easily. Here's a start:

```
# Module hp
from guppy import hpy
hp=hpy()
heap=hp.heap
iso=hp.iso
...
```

And then you can do even, "from hp import *" and get all methods heap, iso etc directly. I may consider distributing such a module, but such a name as hp seems to be too short to pollute the global name space with. Or maybe not, nobody else uses such short module names anyway? Or I could call it guppy.hp . But the question is still if it is worth the trouble when the general method is available.

On the other hand, when using the remote monitor a session context (hp) is predefined in the remote interactive console. This is because it has already taken control over the environment and there is a place to store things other than in global data.

Why not just:

```
import hpy
hp=hpy()
```

Wouldn't this be shorter than have to write 'from guppy import hpy'?

1. We should define as few top level names as we can, since they all share (aka pollute) the same Python package namespace. I thought it was best to define just one name, and it turned out to be guppy and there are possibly a lot of other things beside Heapy I might want to include.

2. Even if we defined hpy at top level this wouldnt work. What you import in this way must be a module, and modules are not callable and can not be made callable by Python. (For historical or whatever reason.)

### 4.7.5 Why API == UI?

Except for the graphical profile browser, the user interface of Heapy is provided directly by methods and other operations on Python objects, both when used interactively and when used from a program. This provides accessibility so these parts of Heapy can be used anywhere you can use Python, even in embedded systems or servers without graphical windowing.

Why not a special, though command line based, user interface?

Entering a special user interface would take extra steps and a context switch for the user. Though it could provide useful features, it turned out that such an interface was out of the scope for Heapy. It was better to provide features that could be used directly from the

Python interactive console. It is also possible to use the IPython package [2], which provides an enhanced interactive Python shell with a variety of features; a macro system, session logging and restoring etc.

### 4.7.6 Why a variety of C functions?

Though all functions are not strictly necessary they are included for speed considerations. Even though computers get faster, memory and program sizes increase as well, making the analysis of all objects in the heap slower. And we may also want a speed margin for new more complex analysis methods.

### 4.7.7 Why nodesets?

It is not possible to use (dict-based) Python sets as containers for all kinds of objects since all objects don't provide the hashing and equality operations required. The nodesets used by Heapy treat objects as equal or different based only on object identity (address), regardless of the operations provided by the objects.

Mutable nodesets are used in particular to keep track of visited objects while doing heap traversal. An alternative would have been to use int objects containing the addresses and store them in a dict-based set. But it would require allocating an int object for each object visited. In particular I was worried that those int objects would never be deallocated again, since in the current Python implementation they would be kept in a pool to speed up future allocations. It would be intrusive if taking a heap census would suddenly make the program use much new memory that couldn't be recovered (even though Heapy can't currently provide information about the int object pool).

Immutable nodesets are used to store the results of various operations, in particular as internal storage for the API level IdentitySet objects. They are also used for internally representing the result of retainer classification; the efficiency of small immutable nodesets is especially useful in such situations.

### 4.7.8 Why not importing Use directly?

One may wonder why the hpy function is defined in guppy top level, when it is the Use module that defines the public Heapy API. It is a shorthand. The hpy function is just a small wrapper. Technically, defining hpy directly in the top level guppy package rather than in Use.py is to avoid importing Use.py when hpy is not going to be used.

### 4.7.9 Why family objects, why not many subclasses of UniSet?

There are just a few subclasses of UniSet, such as IdentitySet and Kind. However, there are many more implementations. Though the implementations could be distinguished by

more subclasses of UniSet, it would become a big class hierarchy that would be difficult to maintain. For one thing, the original doc strings of the operations in the UniSet base class, though still valid, would disappear if the methods were redefined in a subclass. For another thing, there would be many private attributes that should (perhaps ideally) be kept separate from the public methods by some naming convention which might be hard to ensure all the time. The implementation is instead using an extra indirection via the fam attribute containing a Family subclass instance. The attributes and methods of family objects are private to the implementation without requiring a special naming convention. The data consists especially of objects from the session context; such data would have been needed anyway so the fam attribute takes at least no extra space. Also because shared data is stored in the family object, the few UniSet subclasses may use __slots__ efficiently to become compactly represented.

### 4.7.10 Why is monitor a server?

A monitor that is a socket server can accept connections from several monitored processes; in fact, from every (Python) process in the system, so only one monitor is needed. Otherwise, if we would have to use one monitor for each process it would be impossible to handle them manually if there were hundreds or thousands such processes. Some script would be needed, but it would require some scripting environment outside the scope of the monitors themself. If there were few enough processes to handle manually, we would still rely on some system dependent job control or windowing system. The server monitor solves these problems by being a single process that may be used interactively or be programmed to monitor different processes. (It should also be possible to use several monitors if needed.)

### 4.7.11 Why Glue.py?

Why this particular implementation of session context?

There are many ways to structure a session context for a library or application, but the choices are often arbitrary which may give the programmer an headache. Using global variables is easy to start with, but often it turns out to be a bad choice since it restricts how a library can be used. What one typically resorts to is to define a class so that the session context can be dynamically allocated. (A module specific session context class, sometimes with the same name as the module it is defined in.) But some troubling arbitrary choices then arise: When is the class to be instantiated? Where are the instances to be stored? How are instances from different modules related to each other?

One would wish that when writing a new module for some functionality, it should be sufficient to make that module, period. But if it needs to be part of a dynamically allocated session context, the allocation needs to be done some way. Let us assume there should be at most one such allocation per session context. Where is it to be done? Perhaps in some upper level package. But the upper level wouldn't know beforehand that we actually want to use this new module, since it would depend on the application. It seems the actual application

should be in charge of what such instances are created - but the application can not call the constructor at each place it needs such an instance, because it should only be constructed once per session context. Though some solution can be devised in each particular case, it often imposes an inflexible hierarchical structure on the system (or so is the theory), and it should be better to provide a general application independent mechanism.

Glue.py addresses such problems by providing a way to construct a top level session context that is expanded on demand when parts of it are accessed, reflecting the structure of the underlying packages and modules. There is then no need for the programmer to decide how to instantiate module specific session context classes and where to store the results -- it is taken care of in a standard way.

# Chapter 5

# Sets, kinds and equivalence relations

An introduction with examples. [1]

Heapy handles a single object and a group of objects the same, as a set of objects. A set is, generally, a collection of elements where the order is of no significance. A new set may be created from an old set by, for example, adding an element; but if the element to add was 'the same' as an element already in the set, the resulting new set will be the same as the old one. What is regarded as the same element is a matter of definition. The sets provided by the Python standard library have the sameness defined by the equality operator, '==', which may be redefined by the objects themselves. On the other hand, the sets provided by Heapy have the sameness defined by the 'is' operator which is based on address comparison and can not be redefined. In this way, Heapy sets can contain any kind of objects without any special requirements or possible exceptions.

To use Heapy, you first need to create a session context object:

```
>>> from guppy import hpy
>>> h=hpy()
```

## 5.1 Sets

Then given some objects,

```
>>> a=[]
>>> b=[]
>>> c=[1]
```

a set containing those objects may be created as follows:

```
>>> x=h.iso(a,b,c)
```

Here, the iso method is used. The name stands for 'is object' or 'identity set of objects'.

[1] The exact result of the examples depend on the Python version used. In this case it was Python 2.3.5 on Linux / i686, and the Guppy version was 0.1.2.

The resulting set contains the objects passed as arguments to the method. Such a set has several attributes that give information about the elements it contains. For example:

```
>>> x.count
3
```

The number of objects in x.

```
>>> x.size
116
```

The total memory size allocated to the objects in x.

## 5.2 Kinds

For the purpose of this section, the kind attribute is the most relevant:

```
>>> x.kind
h.Type.List
```

This is the 'kind' of objects in x, according to the classifier (equivalence relation) in use by x. A kind is a symbolic set representing all objects (from some Python object universe) that would make a particular predicate true. For example, the objects in h.Type.List have a particular Python type, the builtin list type aka types.ListType. For any IdentitySet x, x.kind is a superset of x, so the following will always hold:

```
>>> x.kind >= x
True
```

Though it is always a superset of x, the particular value of x.kind depends on the equivalence relation in use by x. There is a default but others may be chosen as will be shown later.

The representation of a kind object, 'h.Type.List' in the example, is a Python expression that can be used to recreate it. (The system has been able to figure out that h is our Heapy session context since it is defined in __main__.) The kind object is thus the List attribute of the Type attribute of h.

## 5.3 Equivalence relations and classes

The h.Type attribute is an object that represents an Equivalence Relation. An equivalence relation is a rule to tell if two objects are considered equivalent in some sense, or in other words, that they are of the same kind. The rule should obey certain mathematical laws. An equivalence relation defines a classification scheme, which may be used to associate a specific Heapy 'kind' object with every Python object. Each such 'kind' object represents an equivalence class of the classifying equivalence relation. Equivalence classes are the disjoint sets of objects considered equivalent as defined by an equivalence relation.

The h.Type.List attribute, the kind of x in the example, represents one of the equivalence

classes of h.Type . It is the kind of objects with builtin Python type being types.ListType where 'types' is the standard Python library module that contains the names of many predefined types. There is one such attribute in h.Type for each type defined in 'types'. The convention is that the name of the attribute is the name of the type in types except that the trailing part of the name, 'Type', is stripped.

If the type had been user defined or otherwise not part of the types module, the form of its representation as a kind had been different.

```
>>> class T(object):
...  pass
...
>>> h.iso(T()).kind
h.Type('<type __main__.T at 0x8227dbc>')
```

Calling h.Type as a function in the way suggested recreates the kind object for the type specified in the string argument, as long as there is actually a type with the specified name at the given address. This representation of a type is intended to be a globally unique representation which may be evaluated in any context where h is available. If the type itself is available it may be used directly as the argument when calling h.Type to construct an equivalent kind:

```
>>> h.Type(T) == h.iso(T()).kind
True
```

Kinds may be combined with the usual set theoretical operations: intersection (x & y), union (x | y) and set difference (x - y). There is also a complement operator (~x). The operands may be kind objects as well as IdentitySet objects. For example, to show the number of int or list objects in the heap the following expression may be used:

```
>>> (h.heap() & (h.Type.Int | h.Type.List)).count
296
```

The value of the kind attribute of a set, x.kind, depends not only on the elements of x but also on the classifier used. Each set is equipped with an attribute x.er which is an equivalence relation defining the classifier to use to determine x.kind and also for other purposes. A classifier is almost the same thing as an equivalence relation, so I will use the words interchangeably in the sequel.

```
>>> x.er
h.Clodo
```

This is the default equivalence relation for a set created by iso. Its name stands for Class Or Dict Owner. The classifier is designed to be informative enough in usual cases. In particular, it will provide extra information about objects of dict type. Since dicts are often used as the main storage for class instances, it is useful to know, about such dicts, what kind of objects they belong to. For example, using our previously defined class T:

```
>>> t=T()
>>> print h.iso(t.__dict__).kind
dict of __main__.T
```

This means that t.__dict__ is indeed a dict object that belongs to an instance of the T class. Printing a kind as in the example uses the str() representation which is intended to be short and readable. When evaluating an expression directly at the interactive Python prompt, the more formal representation repr() is used:

```
>>> h.iso(t.__dict__).kind
h.Type('<type __main__.T at 0x81d0694>').dictof
>>>
```

This is the representation that can be used to recreate this kind. Given a kind object k, k.dictof is the kind representing dict objects that are owned by objects of kind k.

Since h.Clodo is the default equivalence relation, one may wonder what other classifiers are available and how they can be used. There are a number of different equivalence relations available in the default Heapy session context. We have seen h.Type and h.Clodo . The h.Type classifier gives less information than h.Clodo, since it doesn't distinguish by dict ownership and also not distinguishes old-style classes. Equivalence relations are themselves related by a partial order. For example:

```
>>> h.Type > h.Class > h.Clodo
True
```

The h.Class classifier is between h.Type and h.Clodo in detail. It distinguishes between old-style classes but not between dicts owned in different ways. (The relationship may be formalized as the subset relation between equivalence relations represented as sets of equivalent pairs.)

Another useful equivalence relation is h.Size . This classifies by memory size. To use h.Size or any particular equivalence relation for classificaion of a set, the set should have the er attribute set to the equivalence relation to use. It is not possible to set this attribute directly. A new set must be created with a new value of er. This may be done in two ways, a general way and an easy way. The general way is to call the by method of the set passing the new equivalence relation as an argument. This is mainly for special uses when new equivalence relations have been created. The easy way is to use one of the predefined byxxx attributes, where xxx is the name of the new equivalence relation to use. (The same name as the equivalence relation has in the session context except that it has to be lowercase!) For example, to show the kind of objects of x, but using the h.Size equivalence relation for classification instead of the default, you may do as follows:

```
>>> x.bysize.kind
(h.Size(28) | h.Size(60))
```

This shows that there are objects of two different sizes in x, 28 and 60 bytes. In general, if a set has objects that are classified differently, the kind of all objects taken together will be the union of the kinds given by the different classifications.

## 5.4 Partitions

One may wonder what objects there are of each particular kind. We may answer such questions starting with the representation of a set as a partition table, which is the default representation when evaluated at the interactive prompt:

```
>>> x.bysize
Partition of a set of 3 objects. Total size = 116 bytes.
 Index  Count   %     Size   % Cumulative  % Individual Size
     0      1  33       60  52         60  52         60
     1      2  67       56  48        116 100         28
```

This shows something called a 'partition'. This term is used in its mathematical sense as a way of splitting a set into disjoint subsets according to some rule. The rule used here is to split according to the classifier in use by the set. Since the classifier is Size and there are two different sizes of objects in the set, there will be two subsets in the partition and two rows in the table.

The first column, Index, is the index of each row in the table. It goes from 0 and upwards. It is especially useful with larger tables to see what index to use to extract a particular subset from the partition.

The Count column contains the number of objects in the subset of that row. The following '%' column is the percentage of the count relative to the total number of objects in the set.

The Size column contains the total memory size of the objects in that row. The following '%' column is the percentage of the memory size relative to the total memory size of the set.

The Cumulative column is the cumulative sum of the Size column. The following '%' column is the percentage of the cumulative size relative to the total memory size of the set.

The rightmost column contains generally a representation of the kind of objects in each row. The header and the column values depend on the classifier used for partitioning. In this case it was the Size classifier. The last column is therefore the size of the individual objects in each row.

The table is sorted on the Size column so that the rows with largest total memory size appear first. If some rows have the same size, they are sorted alphabetically on the kind as represented in the last column,

Such a table is the usual way to show an overview of a set of objects. The different kinds of equivalence relations determine how the set is partitioned into rows in different ways. To provide the most detailed information, and the longest possible table, there is an equivalence relation that treats all objects as different. This is the h.Id equivalence relation which will give a table where each row contains exactly one object:

```
>>> x.byid
Set of 3 <list> objects. Total size = 116 bytes.
 Index     Size   %   Cumulative  %   Address*Length
     0       60  51.7        60  51.7 0xb7a109ec*1
```

```
     1       28  24.1         88  75.9 0xb7a00d6c*0
     2       28  24.1        116 100.0 0xb7a0750c*0
```

In this case, the Count column would be 1 in every row so it is excluded for brevity.

The rightmost column here is a representation of the object itself which is chosen to be short and safe to print in all cases. It is possible to get access to the actual objects themself by selecting a particular row and using the .theone attribute:

```
>>> x.byid[0].theone
[1]
```

This is the actual object at row 0 of x.byid; the largest object in x. The .theone attribute is used only when we need to look at an actual object extracted from a set; it is often more useful to let the objects stay in sets and use other attributes on sets to get information about an object or group of objects.

## 5.5 The heap() method

The examples so far have used x, a set of just 3 elements, for clarity of exposition. In practice, the sets may contain many thousands of elements and then the utility of different kinds of partitioning into rows becomes more apparent. A first step in looking at the memory situation in a program is to make a set of all live objects reachable in the heap.

```
>>> y=h.heap()
>>> y
Partition of a set of 24777 objects. Total size = 1619512 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0  14205  57   845464  52    845464  52 str
     1   5053  20   201200  12   1046664  65 tuple
     2   1511   6    96704   6   1143368  71 types.CodeType
     3     68   0    92960   6   1236328  76 dict of module
     4    185   1    84680   5   1321008  82 dict of class
     5   1440   6    80640   5   1401648  87 function
     6     82   0    63184   4   1464832  90 dict of type
     7     85   0    36668   2   1501500  93 type
     8     40   0    26560   2   1528060  94 dict (no owner)
     9    663   3    23868   1   1551928  96 __builtin__.wrapper_descriptor
<37 more rows. Type e.g. '_.more' to view.>
```

This gives quite a long table since there are apparently 37+10 or 47 different kinds as given by the standard classifier. In this case, the first 10 rows contain the objects contributing to 96 % of the total memory usage, so this is where an optimization effort would primarily be concentrated. The remaining rows are not shown per default but using the more attribute will show 10 more rows and it can be repeated on the resulting object (a MorePrinter). It is also possible to show selected rows, for example the 10 last rows may be shown by y[-10:].

## 5.6 The Via classifier

Concentrating on the more interesting objects, those that take the most memory, we see that 52 % of memory is used by objects of type str, the builtin string type. We might want to know what these strings are mainly used for. We can get information about this by selecting the first row and then using the Via classifier.

```
>>> z=y[0].byvia
>>> z
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'
     2    149   1    73668   9    329340  39 "['__doc__']"
     3    251   2    59384   7    388724  46 '.func_doc', '[0]'
     4   1433  10    52016   6    440740  52 '.co_lnotab'
     5   1443  10    49984   6    490724  58 '.co_name', '.func_name'
     6    315   2    16744   2    507468  60 '[1]'
     7    155   1     6488   1    513956  61 '[2]'
     8    113   1     4372   1    518328  61 '[3]'
     9     57   0     3624   0    521952  62 "['__file__']"
<4495 more rows. Type e.g. '_.more' to view.>
```

The Via classifier is used to distinguish objects by how they are referred to from referrers in the heap.For example, the objects in row 0 are all referred by some attribute named '.co_code', and only by that name. Formally, the kind of those objects according to the Via classifier is represented as:

```
>>> z[0].kind
h.Via('.co_code')
```

Some objects are referred to in more than one way, for example the objects in row 3 are referred to via some attribute '.func_doc' and also via some indexing operation '[0]' which may be from lists, tuples or dicts or something similar with a built in indexing operation.

```
>>> z[3].kind
h.Via('.func_doc', '[0]')
```

Information of the actual referrer objects is not provided by the Via classifier but may be provided by other means. One way is to use the .referrers attribute:

```
>>> z[0].referrers
Partition of a set of 1510 objects. Total size = 96640 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0   1510 100    96640 100     96640 100 types.CodeType
```

All referrers of row 0 were of type CodeType, which one might have guessed, knowing something about code objects, since they were all referred to via the '.co_code' attribute. These objects are all code strings, i.e. the actual Python byte code that is executed.

## 5.7 An optimization possibility

Looking back in the table printed for z itself, we see in row 0 that 18 % of the string memory usage is for program code, which is not especially surprising and not much to do about here. Looking at row 1 however, shows that 12 % or 99432 bytes of strings are referred to via something called '.co_filename'. This is the name of the file that the code was compiled from. We also see that there are 1511 filename strings for 1510 code strings. So there is one new filename for each different code string (plus one extra for some reason.). This shows that there should be a possibility for optimization here, since there is not that many different filenames so the same filename string could be used again.

Such an optimization has in fact been introduced in Python 2.4. These examples were created using Python 2.3, for historical reasons, but when I found this optimization possibility and realized it had been fixed in 2.4 I thought it was motivated to not change the examples from Python 2.3, but to keep this example to show how a partition table as printed using different classifiers may be used to get an overview of information that can reveal possible optimizations.

# Chapter 6

# Using Heapy to find and seal a memory leak

## 6.1 Background

The Heapy Profile Browser, as is common in similar programs, provides a menu command to open a new window. Such a window will use memory for allocated objects. The window may be closed again. One would then hope that the allocated memory would be released. Opening and closing a window several times would otherwise allocate increasingly more memory. This would be what is called a memory leak. Ultimately the program would run out of memory so it would fail and have to be restarted. A situation like this is sometimes called a 'usability catastrophe'.

Testing with Heapy did indeed show that some of the memory allocated when opening a new window, was not released after closing it. The excess memory was found to be about 5300 bytes. Though a leak of this magnitude is arguably not a major issue in this particular case, the analysis revealed it was caused by a general problem which could cause worse leaks in other situations.

The leak was found to be due to some tricky issues regarding the interface to tkinter, the extension module in the standard Python library that implements the graphical interface used by the profile browser.

In Tkinter, the Python wrapper around tkinter, there is a class, Menu, that is used to create pull-down menus. Menu items can be added to Menu objects and can also be removed. When adding a menu item, a command (a callable object) is passed as an argument, and it will be called when the menu item is selected. The Menu object needs to somehow keep a reference to the command. It must be available to be called, for as long as the menu item exists. But what happens when the menu item is removed? The command may not be called anymore from that menu item, so the reference could be removed. But it turns out that there will still be a reference to the command. The reference will not be removed until the Menu object itself is deleted.

In the profile browser, the actual situation was as follows.

The Menu object was the Window menu. Each time a new window is opened, an item is added to the Window menu. This menu item, when selected, will select that window. The menu item, when created, is passed the command to be invoked when selected. When closing the window, the menu item is removed. But the selection command is still being referenced. (From somewhere internal to the low level tkinter extension module.) The selection command is a method of an object representing the window it operates on. It thus has a reference to this window object, which in turn contains other subobjects. All these objects, directly or indirectly referenced by the selection method, will be retained in memory even though the window itself was closed.

Having figured out the cause of the problem, it is possible to devise a solution. When deleting a menu item, the reference to its associated command should also be deleted. I made a subclass of Menu redefining the menu item delete method to do that. The source code of the profile browser (Prof.py) now uses this subclass. When using this variant, the memory leak is sealed, according to tests using Heapy.

I thought the case was potentially serious enough so I reported it to the Python bug tracker at Sourceforge (bug #1342811) and attached the Menu subclass as a possible solution.

## 6.2 Debugging approach

1. For somebody wanting to repeat this experiment, the leak should be made to actually appear again. Since the current version of the profile browser has sealed the leak, it must be changed. A way to do this, is to comment out two lines of the Menu class in guppy.heapy.Prof.py, like this:

   ```
   #for c in cmds:
           #    self.deletecommand(c)
   ```

2. In one terminal window, start Python, enable remote monitoring, and start the profile browser.

   ```
   python -c"import guppy.heapy.RM;from guppy import hpy;hpy().pb()"
   ```

3. In another terminal window, start Python and the remote monitor.

   ```
   python -c"from guppy import hpy;hpy().monitor()"
   <Monitor>
   *** Connection 1 opened ***
   ```

   The monitor tells us it has found a connection. It is the connection to the profile browser. We need to actually connect to it and start an interactive console:

   ```
   <Monitor> sc 1
   Remote connection 1. To return to Monitor, type <Ctrl-C> or .<RETURN>
   <Annex> int
   Interactive console. To return to Annex, type '-'.
   >>>
   ```

4. In the interactive console, there is a ready-made Heapy session context, named hp. We can, for example, check the heap of the running process by doing hp.heap(). For the purpose of this experiment, we will first set up a reference point so that further heap information will only show new data relative to that point.

```
>>> hp.setref()
>>> hp.heap()
hp.Nothing
```

   This means there is nothing new in the heap since we just did hp.setref().

5. We will now open a new window in the profile browser. (Menu item File->New Profile Browser)

6. Back in the monitor again, we check the current heap report. It should look like something like this:

```
>>> hp.heap()
Partition of a set of 685 objects. Total size = 91300 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0    269  39    13332  15     13332  15 str
     1     81  12    12936  14     26268  29 dict (no owner)
     2     14   2     7280   8     33548  37 dict of Tkinter.Frame
     3     10   1     5200   6     38748  42 dict of Tkinter.Label
     4     10   1     5200   6     43948  48 dict of Tkinter.Menu
     5     10   1     5200   6     49148  54 dict of guppy.heapy.Prof.ColSpec
     6      9   1     4680   5     53828  59 dict of Tkinter.Canvas
     7      8   1     4160   5     57988  64 dict of Tkinter.Menubutton
     8      8   1     4160   5     62148  68 dict of guppy.heapy.Prof.ClickButton
     9      6   1     3120   3     65268  71 dict of Tkinter.Checkbutton
<55 more rows. Type e.g. '_.more' to view.>
```

   The table shows the objects that are currently allocated, except those allocated at the time we did setref(). These objects are (or should mostly be) those required by the new window. We will now close this window again, using menu item File->Close Window. Make sure it is the new window that is closed, not the old one.

7. After closing the window, we may hope that the objects used by it should go away. In the monitoring window again, we check the current state of the heap:

```
>>> hp.heap()
Partition of a set of 3 objects. Total size = 132 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0      3 100      132 100       132 100 str
```

8. This shows there is an excess of 3 strings! Let us save this set of objects for later reference.

```
>>> x=_
```

9. What are the strings? byid will show the actual strings.

```
>>> x.byid
Set of 3 <str> objects. Total size = 132 bytes.
 Index     Size   %   Cumulative  %   Representation (limited)
```

```
     0       44  33.3        44  33.3 '-1223623828wakeup'
     1       44  33.3        88  66.7 '-1223666788wakeup'
     2       44  33.3       132 100.0 '-1224167876wakeup'
```

This shows that there are 3 similar strings with strange looking contents. Certainly not any strings that I knew I had used.

10. One may wonder here, if these strings are of any importance, they are not so big. But I wanted to find out more of them. Especially, how they are referred from other objects. A listing of the shortest paths from the root often gives useful summary information.

```
>>> x.shpaths
 0: ...['windowmenus'][1].__dict__['menu'].__dict__['_tclCommands'][6]
 1: ...['windowmenus'][1].__dict__['menu'].__dict__['_tclCommands'][7]
 2: ...['windowmenus'][1].__dict__['menu'].__dict__['_tclCommands'][8]
>>>
```

11. There are 3 shortest paths. (Abridged for clarity.) They are all the same except for the last indexing operation. We see that the strings are in something called '_tclCommands' . It doesn't tell us what kind of object this is. Another perspective is given by the reference pattern:

```
>>> x.rp
Reference Pattern by <[dict of] class>.
 0: _ --- [-] 3 str: '-1224189156wakeup'..., '-1224191356wakeup'
 1: a      [-] 1 list: -0x491c0634*6
 2: aa ---- [-] 1 dict of guppy.heapy.Prof.Menu: -0x48f018d4
 3: a3       [-] 1 guppy.heapy.Prof.Menu: -0x48f018d4
 4: a4 ------ [-] 1 dict (no owner): -0x48f0039c*1
 5: a5         [-] 1 dict of Tkinter.Menubutton: -0x48f01874
 6: a6 -------- [-] 1 Tkinter.Menubutton: -0x48f01874
 7: a7           [+] 1 dict (no owner): -0x48f0339c*6
 8: a6b -------- [^ 2] 1 dict of guppy.heapy.Prof.Menu: -0x48f018d4
 9: a6c          [+] 1 dict of guppy.heapy.Prof.WindowMenu: -0x48f019f4
10: a3b ----- [^ 9] 1 dict of guppy.heapy.Prof.WindowMenu: -0x48f019f4
```

Each line represents a set of objects. The set of 3 strings is itself on the first line. The next line represents one list object that references the strings. From the shortest path information, we know that the strings are at index 6, 7 and 8 in that list. The list itself is referenced by a dict object on the next row. It is the __dict__ of a Tkinter.Menu instance. In combination with the shortest path info, we can thus tell that the list is actually the _tclCommands attribute of the Tkinter.Menu object.

(It should be possible to combine shortest paths and reference pattern info in some sort of combined table, but I had not yet figured it out at the time of this writing. It may often be clearer to look at them separately even though one has to look back and forward to get the full info.)

One may then wonder what purpose there is with those strings and why they remain after the window is closed. One may also wonder what will happen if a new window

is opened and then closed. Checking this reveals that new strings are added to the _tclCommands list every time. The list keeps growing.

To know more, we need to look in the actual code of Tkinter.Menu. After some browsing around, I figured out what was happening. The strings are the names of commands that have been passed as arguments when adding menu items. They are used to keep track of the command objects so the references to them can be deleted when the Menu object itself is deleted.

But if there are references kept to the command objects, one may wonder why not those command objects show up when doing hp.heap(). Such an object should be a method unique to the window that was closed. It turns out the command objects are referenced only from inside an external C library. They are not found by the heap traversal done by heap(). There are however other means to find such objects (at least those with GC info and their referents).

12. Information about such unreachable objects is currently provided by a separate method, heapu(). Although there are means to get at the actual objects, the result of heapu is usually a table containing summary data only. Using this method we find out about the objects held on to from internally in tkinter.

```
>>> hp.heapu()
Data from unreachable objects relative to: Sun Feb  5 16:30:01 2006.
Summary of difference operation (A-B).
        Count     Size
  A       983   487060
  B       943   481860
  A-B      40     5200  =      1.08 % of B
Differences by kind, largest absolute size diffs first.
 Index  Count     Size  Cumulative  % of B Class
     0     14     4208      4208     0.873 dict
     1      9      416      4624      0.96 str
     2      8      288      4912      1.02 types.MethodType
     3      3       96      5008      1.04 Tkinter.CallWrapper
     4      1       32      5040      1.05 Tkinter.Frame
     5      1       32      5072      1.05 Tkinter.Menubutton
     6      1       32      5104      1.06 Tkinter.Toplevel
     7      1       32      5136      1.07 guppy.heapy.Prof.Menu
     8      1       32      5168      1.07 guppy.heapy.Prof.Window
     9      1       32      5200      1.08 guppy.heapy.Prof.WindowMenu
>>>
```

This shows the relative difference of objects since the last time we set the reference point, which was when we used the hp.setref() method. It shows that there are now 40 more objects and the total size increased with 5200 bytes. It then shows the difference per class.

Looking back in the code, it is possible to figure out what can be tried to solve the problem. The deletecommand method should be called to remove the reference to the commands not longer needed. It is possibly to try this out directly.

Looking at the reference pattern again:

```
>>> x.rp
...
 3: a3       [-] 1 guppy.heapy.Prof.Menu: -0x48f018d4
...
```

The reference pattern is an object that can be used to get access to the object sets represented on each line, either by indexing or attribute access. For example, either x.rp[3] or x.rp.a3 gives the set containing the Menu object. To get at the actual underlying object is then possible using the .theone attribute, in case the set is a singleton. Otherwise, the iterable .nodes attribute is always available.

13. Using these access methods, it is possible to write a little loop that will call the deletecommand method of the Menu object, for each of the strings in x.

```
>>> for s in x.nodes:
...  x.rp.a3.theone.deletecommand(s)
...
>>>
```

14. The big question now is of course if this code actually did something good. The result can be verified directly. The strings don't show up in the reachable heap any more:

```
>>> hp.heap()
hp.Nothing
```

15. And similarly hp.heapu() shows that the unreachable objects have also been removed.

```
>>> hp.heapu()
...
  A-B       0         0  =          0 % of B
```

There is zero difference from when the reference point was set.

16. So this verified that a call to deletecommand would solve the problem. Remains then to try this out in practice. This was done by subclassing Menu replacing the delete method. For those of us doing this experiment at a later time, it remains to uncomment the lines previously commented out so they read:

```
for c in cmds:
            self.deletecommand(c)
```

17. Then, the profile browser can be restarted and it may then be checked that opening and closing a window will not leave any remaining memory allocated, except that sometimes there will be one int object.

```
>>> x=hp.heap()
>>> x
Partition of a set of 1 object. Total size = 12 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0      1 100       12 100        12 100 int
```

```
>>> x.shpaths
 0: Src.t3085265120_f3.f_globals['_varnum']
>>> x.rp
Reference Pattern by <[dict of] class>.
 0: _ --- [-] 1 int: 113
 1: a      [S] 1 dict of module: Tkinter
>>>
```

From this, we see that there is a global variable called _varnum in the Tkinter module. I think this object shows up the first time when the counter passes 100, because smaller integers are preallocated. There is no problem with this, it is no leak since successive opens and closes will not add any more such int objects. The previous one will always be deallocated. This may be verified by opening and closing a window several times; there is still only one excess int object.

# Chapter 7

# Conclusions and future work

Heapy provides a variety of features to support debugging and optimization of memory allocation issues. Some of those issues are called memory leaks. It has been shown that Heapy may be used to find and seal a memory leak in the following way:

- The hpy().monitor() command is used to remotely monitor the suspect program.
- The monitor provides an interactive Python console in the monitored process, in which a Heapy session context (hp) is predefined.
- The hp.setref() method sets a reference point for heap usage measurement.
- After setting a reference point, some suspected operation in the application, such as opening and then closing a window, can be performed.
- The hp.heap() and hp.heapu() methods report reachable and unreachable memory usage respectively, relative to the reference point.
- Attributes of object sets (shpaths and rp) provide the shortest referring paths and the reference pattern; different perspectives of how objects are referred. This told us the attributes and objects retaining the leaking objects. It was then possible to find the relevant source code and figure out the cause of the leak.
- Attributes of a reference pattern have a tree selection syntax used to select object sets. These have attributes (theone and nodes) for access to the underlying objects. Using these objects, a problem solution could, in the example, be tested directly without restarting the application program.
- It was possible to verify that the leaking objects were deallocated by using hp.heap() and hp.heapu() again in the same session.
- Using the information gained interactively, it was possible to figure out how to update the program so that the leak was sealed permanently.

In common for most Heapy operations, is that they either create or use an *identity based object set*. In the example, hp.heap() creates a set containing the objects in the heap, less those that existed at the reference point. In other cases, the objects may be known beforehand, and then the iso method would be used. Object sets have a variety of attributes (mostly with values created on demand when accessed). In this case, two structural analysis attributes were used, shpaths and rp. In other cases, other attributes might have been useful, for example `dominos` the dominated set or `byvia` a kind of retainer profile.

The use of the remote monitor was a way of getting in touch with an unmodified application program, and making a clear distinction between application objects and analysis objects. It is however also possible to use the Heapy package directly from within the same process as the application. It still distinguishes between Heapy internal data and application data, though the application data may be mixed in with some of the data created by the user's analysing program, unless special precautions are taken.

The scenario in the example is of course just one of many possible cases. There are also optimization cases that are not usually called memory leaks. Such cases may be found by looking at object populations from different perspectives. In Chapter 5 it was shown that using the Via classifier to generate a retainer profile for all the strings in the heap, revealed an optimization possibility regarding the Python 2.3 runtime. The profile showed that there were many strings retained by a certain attribute (co_filename), indicating that they were allocated each time a new code object was created, without checking if the same string was already allocated. They did not need to be separately allocated since strings are immutable. This optimization has been introduced in version 2.4 of the Python runtime. But application programs need to take care of such optimizations themselves. There may be surprisingly many separate strings or other objects with the same value that can be shared.

In general, the various classifiers, used one at a time or in combinations, provide different viewpoints of object populations; quick overviews which may reveal optimization possibilities not thought of beforehand.

Heapy provides also some features that are not used in any examples in this report, though I think they are useful enough to mention here:

Dominated set

> The set of objects that would be deallocated when some particular dominating object (or dominating set of objects) is deallocated. This can be useful for getting a picture of an entire composite object to decide which objects might be worthwhile to optimize.

Profile browser

> Interactive graphical presentation of memory usage by different kinds of objects, as it evolves with time. This is useful to get an overall picture of the memory behaviour of a program.

It is the intention that Heapy should become generally available as a Python package — it is available as source code, though it will take some work to make sure it is compatible with different platforms. There is most certainly a need for memory analysis systems —

even though memory sizes get larger, programs also get larger and more complex. Though one may get by anyway — and most programmers do — by guesswork or using improvised tools, it may get very frustrating in the long run. Such was mine experience when I had written a somewhat complex program, a compiler, which worked quite well except it ran out of memory in certain situations. It was hard — if at all possible — to figure out such things as what kind of objects used the most memory. In fact, I gave up on the compiler for the time being and started writing Heapy instead.

When at all possible, it is generally better to *know* than to *guess*. But what is possible to know depends on the tools used. Such tools include mathematical tools but also physical measurement instruments. A memory usage analyser is a measuring instrument in software form.

> In science, *observation* plays a central role in the advancing of knowledge. Careful study of accurate observations often precedes the formulation of valuable new ideas. But observation usually requires *instruments*, and the availability or lack of suitable instruments in some field of enquiry often proves critical. [20, p. 588]

## 7.1 Future work

Though the examples provided in this report have indicated the potential usefulness of the implemented system — and of course it should be useful to have access to information that was previously unknown — there seems to be nothing that theoretically proves that particular analysis features are generally useful. More examples from real use cases would strengthen the case, or motivate the addition of new features, and testing with new examples is certainly an important future work. But any amount of successful testing would still not guarantee that the system generalises well to unseen examples. The background study is based on existing work that arguably share the same problem.

It is possible that the situation could be improved by better theoretical analysis and model building. One could make quite advanced models of the problem situation. The examples — use cases — could be treated as objects in the model, and within the model it could be predicted, to some extent, what new cases could occur, and how they would be treated by the system. Given that a fully automatic programming system is impossible, it would also be necessary to include the programmer in the model. To make a model of the human programmer it may help to look into cognitive science. Though a complete understanding seems out of the reach — maybe until the time when there will be no more need for programmers — some simplified models may still be useful. In the thesis [23] the human and computer is modelled as one computational system. The task for a software engineering tool is then generally to improve the computational abilities of that system. Two examples of the means to do that are:

- Externalization. Moving data and processing from the human to the computer. Or in general to some artifact. A shopping list is a simple example of externalization of memory.

- Specialization. Replacing a general resource-consuming process with a specialized streamlined process. Presumably in the computer. A simple calculator would be an example of this.

The thesis, besides the "distributed cognition framework" presented, also provides an overview of much research and raises some fundamental questions about the nature of human-computer interaction.

Besides theoretical analysis and model building, possible future work also includes the following.

- More classifiers

  For example, I think it could be useful to have some way of combining the dominated set with classifying, to cluster objects by some generalized notion of which objects they are owned by.

- Better handling of complexity to find out how objects are retained. Sometimes, reference patterns tend to be complicated and it is then hard to get the overall picture.

- Some general way to let the programmer specify the expected structure of data and have the system compare for differences. This would be useful for automated test and validation.

- Handling of special types in extension modules. Though there are hooks in Heapy, the actual functions need to be written to handle each case.

- More examples and documentation will of course always be useful.

There are some especially murky areas regarding technical aspects of the implementation.

The way to take a snapshot of the heap population and separating out application objects from the analysis objects is based on traversing the heap from a root, avoiding analysis objects. Such a traversal will not find objects that are not reachable from a root, perhaps they are used only by C routines in extension modules. A special routine had to be added for this purpose. However, it will not normally collect those objects into sets since they would become reachable. This means it is harder, if not impossible, to analyse those objects to tell why they are kept in memory. A way out could perhaps be to disallow any analysis objects at all in the heap. But then the system could not be used as a library directly from within an application. This seems to be so useful so it is unthinkable to drop this possibility. Perhaps some compromise could be found in the future.

The way reference points are created is normally as sets. An object set keeps a reference to each element. So even if an object would otherwise be deallocated it will not be deallocated as long as it is in the reference set. Though it will appear to be deallocated, since it is not longer reachable except via the set, it is not really deallocated which may have some effect on the program, especially if it has a destructor (__del__ method) since it will not be invoked. There may be a way out involving hooking into the deallocation methods at the C level but

it seemed so risky so it is not officially supported in the current version. The problem has to do with the constraint to not require a specially modified Python interpreter.

There are also other problems but this will be enough for now!

# Appendix A

# API specification (extract)

This is an abridged specification, containing only the UniSet and IdentitySet API. For the full Heapy API specification, see the documentation links at the Guppy-PE homepage: http://guppy-pe.sourceforge.net

## .tgt.heapykinds.UniSet

### Name

#### UniSet

### Synopsis

#### For any object *x* of kind *UniSet*:

**Methods**

x. *disjoint*(*y* : *UniSet+*) ↦ *boolean*
x. *get_ckc*()
x. *get_render*() ↦ *callable*

**Operators**

**bool**(x) ↦ *boolean*
**str**(x) ↦ *string*
**repr**(x) ↦ *string*
**~**(x) ↦ *UniSet*
x **&** *y* : *UniSet+* ↦ *UniSet*
x **|** *y* : *UniSet+* ↦ *UniSet*
x **^** *y* : *UniSet+* ↦ *UniSet*
x **-** *y* : *UniSet+* ↦ *UniSet*
*y* : *Any+***in** x ↦ *boolean*
x **==** *y* : *UniSet+* ↦ *boolean*
x **!=** *y* : *UniSet+* ↦ *boolean*
x **<=** *y* : *UniSet+* ↦ *boolean*

x < *y* : *UniSet+* ↦ *boolean*
x >= *y* : *UniSet+* ↦ *boolean*
x > *y* : *UniSet+* ↦ *boolean*

**Attributes**

x. *biper* : *EquivalenceRelation*
x. *brief* : *string*
x. *dictof* : *UniSet*

# Methods

x.**disjoint**(*y* : *UniSet+*) ↦ *boolean*

**Returns** True if x and y are disjoint sets, False otherwise. This is equivalent to calculating

not (x & y)

but may be implemented more efficiently in some cases.

x.**get_ckc**()

Get low-level classification information, where available. Returns a tuple (classifier, kind, comparator).

x.**get_render**() ↦ *callable*

**Returns** a function that may be used to render the representation of the elements of self.

This is mainly intended for internal representation support. The function returned depends on the kind of elements x contains. The rendering function is choosen so that it will be appropriate, and can be used safely, for all objects of that kind. For the most general kind of objects, the rendering function will only return an address representation. For more specialized kinds, the function may provide more information, and can be equivalent to the builtin repr() when the kind is narrow enough that it would work for all elements of that kind without exception.

# Operators

**bool**(x) ↦ *boolean*

**Returns** True if self contains some element, False otherwise.

**str**(x) ↦ *string*

Convert the set to a printable string. The string is usually the same as the .brief attribute, but a major exception is the IdentitySet class.

**repr**(x) ↦ *string*

Convert the set to a printable string. This may be a more formal representation than str().

~(x) ↦ *UniSet*

Complement: the set of objects that are not in x.

x & *y* : *UniSet+* ↦ *UniSet*

Intersection: the set of objects that are in both x and y.

x | *y* : *UniSet+* ↦ *UniSet*

Union: the set of objects that are in either x or y.

x ^ *y* : *UniSet+* ↦ *UniSet*

Symmetric set difference: the set of objects that are in exactly one of x and y.

x **-** *y* **:** *UniSet+* ↦ *UniSet*

Set difference: the set of objects that are in x but not in y.

*y* **:** *Any+***in** x ↦ *boolean*

Inclusion test. True if y is a member of x, False otherwise.

x **==** *y* **:** *UniSet+* ↦ *boolean*

Equal: x and y contain the same elements.

x **!=** *y* **:** *UniSet+* ↦ *boolean*

Not equal: x and y do not contain the same elements.

x **<=** *y* **:** *UniSet+* ↦ *boolean*

Subset, non-strict: all elements in x are also in y.

x **<** *y* **:** *UniSet+* ↦ *boolean*

Subset, strict: all elements in x are also in y, and y contains some element not in x.

x **>=** *y* **:** *UniSet+* ↦ *boolean*

Superset, non-strict: all elements in y are also in x.

x **>** *y* **:** *UniSet+* ↦ *boolean*

Superset, strict: all elements in y are also in x, and x contains some element not in y.

## Attributes

x.**biper** **:** *EquivalenceRelation*

A bipartitioning equivalence relation based on x. This may be used to partition or classify sets into two equivalence classes:

x.biper(0) == x

The set of elements that are in x.

x.biper(1) == ~x

The set of elements that are not in x.

x.**brief** **:** *string*

A string representation of self, which is brief relative to the representation returned by str() and repr(). (In many cases it is the same - both are then brief - but for IdentitySet objects the brief representation is typically much shorter than the non-brief one.)

x.**dictof** **:** *UniSet*

If x represents a kind of objects with a builtin __dict__ attribute, x.dictof is the kind representing the set of all those dict objects. In effect, .dictof maps lambda x:getattr(x, '__dict__') on the objects in x. But it is symbolically evaluated to generate a new symbolic set (a Kind).

# .tgt.heapykinds.IdentitySet

## Name

### IdentitySet

## Synopsis

***Subkind of* : *UniSet***

**For any object *x* of kind *IdentitySet*:**

**Methods**

x. *by*(*er* : *EquivalenceRelation*) ↦ *IdentitySet*
x. *diff*(*y* : *IdentitySet*) ↦ *Stat*
x. *dump*(*fn* : *writeable_filename_or_file+* [`mode` = *writing_mode_string+*])
x. *get_rp*( **draw:**[`depth` = *positive+* , `er` = *EquivalenceRelation+* , `imdom` = *boolean+* , `bf` = *boolean+* , `src` = *IdentitySet+* , `stopkind` = *UniSet+* , `nocyc` = *boolean+*]) ↦ *ReferencePattern*
x. *get_shpaths*( **draw:**[`src` = *IdentitySet+* , `avoid_nodes` = *IdentitySet+* , `avoid_edges` = *NodeGraph+*]) ↦ *Paths*

**Operators**

**hash**(x) ↦ *int*
**len**(x) ↦ *notnegative*

**Attributes**

x. *byclass* : *IdentitySet*
x. *byclodo* : *IdentitySet*
x. *byid* : *IdentitySet*
x. *bymodule* : *IdentitySet*
x. *byrcs* : *IdentitySet*
x. *bysize* : *IdentitySet*
x. *bytype* : *IdentitySet*
x. *byunity* : *IdentitySet*
x. *byvia* : *IdentitySet*
x. *count* : *notnegative*
x. *dominos* : *IdentitySet*
x. *domisize* : *notnegative*
x. *er* : *EquivalenceRelation*
x. *imdom* : *IdentitySet*
x. *indisize* : *notnegative*
x. *kind* : *Kind*
x. *maprox* : *MappingProxy*
x. *more* : *MorePrinter*
x. *nodes* : *ImmNodeSet*
x. *size* : *notnegative*
x. *owners* : *IdentitySet*
x. *partition* : *Partition*
x. *parts* : *iterable*
x. *pathsin* : *Paths*
x. *pathsout* : *Paths*
x. *referents* : *IdentitySet*
x. *referrers* : *IdentitySet*
x. *rp* : *ReferencePattern*
x. *shpaths* : *Paths*

x. *stat* : *Stat*

## Subkind of : *UniSet*

## Methods

x.**by**(*er* : *EquivalenceRelation*) ↦ *IdentitySet*

**Returns** a copy of x, but using a different equivalence relation for partitioning. That is, the *er* attribute of the copy will be set to the er argument.

**See also**

*partition*

x.**diff**(*y* : *IdentitySet*) ↦ *Stat*

**Returns** a statistics object that is the difference of the statistics of x and y, i.e. it is a shorthand for "`x.stat - y.by(x.er).stat`".

**See also**

*stat*

x.**dump**(*fn* : *writeable_filename_or_file+* [`mode =` *writing_mode_string+*])

Dump statistical data to a file.

A shorthand for x.stat.dump.

**Arguments**

*fn* : *writeable_filename_or_file+*

A file name or open file to dump to.

`mode =` *writing_mode_string+*

The mode to open the file with, if not already open.

**Default:** 'a'

Per default, the file is opened in append mode.

**See also**

*stat*

x.**get_rp**( **draw:**[`depth =` *positive+* , `er =` *EquivalenceRelation+* , `imdom =` *boolean+* , `bf =` *boolean+* , `src =` *IdentitySet+* , `stopkind =` *UniSet+* , `nocyc =` *boolean+*]) ↦ *ReferencePattern*

**Returns** An object containing the pattern of references to the objects in x, including references to those referrers in turn. It has a string representation in the form of a tree of a certain maximum depth, with cycle patterns marked.

**Arguments**

`depth =` *positive+*

The depth to which the pattern will be generated.

**Default:** The depth attribute of the RefPat module glue.

`er =` *EquivalenceRelation+*

The equivalence relation to partition the referrers.

The default is to use the *Use.Clodo* equivalence relation

`imdom =` *boolean+*

If true, the immediate dominators will be used instead of the referrers. This will take longer time to calculate, but may be useful to reduce the complexity of the reference pattern.

**Default:** False

`bf` = *boolean+*

If true, the pattern will be printed in breadth-first order instead of depth-first. (Experimental.)

**Default:** False

`src` = *IdentitySet+*

An alternative reference source.

**Default:** The default heapy root.

`stopkind` = *UniSet+*

The referrers of objects of kind stopkind will not be followed.

The default is a kind representing a union of all modules, classes, types and their dicts; and also all code and frame objects. Actually, it is made by code like this:

```
stopkind = (
            hp.Type.Module |
            hp.Type.Class |
            hp.Type.Type |
            hp.Type.Module.dictof |
            hp.Type.Class.dictof |
            hp.Type.Type.dictof |
            hp.Type.Code |
            hp.Type.Frame
            )
```

`nocyc` = *boolean+*

When True, certain cycles will not be followed.

**Default:** False

**See also**

*rp*, *shpaths*

x.**get_shpaths**( **draw:**[`src` = *IdentitySet+* , `avoid_nodes` = *IdentitySet+* , `avoid_edges` = *NodeGraph+*]) ↦ *Paths*

**Returns** an object containing the shortest paths to objects in x.

**Arguments**

`src` = *IdentitySet+*

An alternative source set of objects.

**Default:** The default heapy root.

`avoid_nodes` = *IdentitySet+*

Nodes to avoid

**Default:** No nodes are avoided, except those that must be avoided to hide the data in the heapy system itself.

`avoid_edges` = *NodeGraph+*

Edges to avoid

**Default:** No edges are avoided.

**See also**

*shpaths*

# Operators

**hash**(x) ↦ *int*

Hashing

**Returns** a hash value based on the addresses of the elements.

**len**(x) ↦ *notnegative*

**Returns** the number of different sets in the partitioning of x, as determined by its equivalence relation ( *er*) attribute.

**See also**

*partition*

# Attributes

x.**byclass** : *IdentitySet*

A copy of x, but with *Use.Class* as the equivalence relation.

x.**byclodo** : *IdentitySet*

A copy of x, but with *Use.Clodo* as the equivalence relation.

x.**byid** : *IdentitySet*

A copy of x, but with *Use.Id* as the equivalence relation.

x.**bymodule** : *IdentitySet*

A copy of x, but with *Use.Module* as the equivalence relation.

x.**byrcs** : *IdentitySet*

A copy of x, but with *Use.Rcs* as the equivalence relation.

x.**bysize** : *IdentitySet*

A copy of x, but with *Use.Size* as the equivalence relation.

x.**bytype** : *IdentitySet*

A copy of x, but with *Use.Type* as the equivalence relation.

x.**byunity** : *IdentitySet*

A copy of x, but with *Use.Unity* as the equivalence relation.

x.**byvia** : *IdentitySet*

A copy of x, but with *Use.Via* as the equivalence relation.

x.**count** : *notnegative*

The number of individual objects in x.

x.**dominos** : *IdentitySet*

The set 'dominated' by the set of objects in x. This is the objects that will become deallocated, directly or indirectly, when the objects in x are deallocated.

**See also**

*domisize*

x.**domisize** : *notnegative*

The dominated size of the set x. The dominated size of x is the total size of memory that will become deallocated, directly or indirectly, when the objects in x are deallocated.

**See also**

*dominos*, *size*

x.**er** : *EquivalenceRelation*

The Equivalence Relation of x, used for partitioning when representing / printing this set.

x.**imdom :** *IdentitySet*

The immediate dominators of x. The immediate dominators is a subset of the referrers. It includes only those referrers that are reachable directly, avoiding any other referrer.

x.**indisize :** *notnegative*

The total 'individual' size of the set of objects. The individual size of an object is the size of memory that is allocated directly in the object, not including any externally visible subobjects.

**Synonym**

*size*

**See also**

*domisize*

x.**kind :** *Kind*

The kind of objects in x. It is the union of the element-wise classifications as determined by the equivalence relation of x, *er*.

x.**maprox :** *MappingProxy*

An object that can be used to map operations to the objects in x, forming a new set of the result. This works as follows:

- Getting an attribute of the MappingProxy object will get the attribute from each of the objects in the set and form a set of the results. If there was an exception when getting some attribute, it will be ignored.
- Indexing the MappingProxy object will index into each of the objects in the set and return a set of the results. Exceptions will be ignored.

**Example**

```
>>> hp.iso({'a':'b'}, {'a':1}).maprox['a'].byid
Set of 2 <mixed> objects. Total size = 40 bytes.
 Index     Size   %   Cumulative  %   Brief
     0       28  70.0        28  70.0 str: 'b'
     1       12  30.0        40 100.0 int: 1
>>>
```

x.**more :** *MorePrinter*

An object that can be used to show more lines of the string representation of x. The object returned, a MorePrinter instance, has a string representation that continues after the end of the representation of x.

x.**nodes :** *ImmNodeSet*

The actual objects contained in x. These are called nodes because they are treated with equality based on address, and not on the generalized equality that is used by ordinary builtin sets or dicts.

x.**size :** *notnegative*

The total 'individual' size of the set of objects. The individual size of an object is the size of memory that is allocated directly in the object, not including any externally visible subobjects.

**Synonym**

*indisize*

**See also**

*domisize*

x.**owners :** *IdentitySet*

The set of objects that 'own' objects in x. The owner is defined for an object of type dict, as the object (if any) that refers to the object via its special __dict__ attribute.

x.**partition :** *Partition*

A partition of the set of objects in x. The set is partitioned into subsets by equal kind, as given by an equivalence relation. Unless otherwise specified, the equivalence relation used is 'byclodo', which means it classifies 'by type or class or dict owner'. Different equivalence relations can be specified by using 'by_...' attributes of any IdentitySet object.

**See also**

*er*, *by*, *parts*

x.**parts :** *iterable*

An iterable object, that can be used to iterate over the 'parts' of x, i.e. the subsets in the partitioning of x. The iteration order is determined by the sorting order of the partition. The partition is normally sorted with the subsets that have the larger total object size first, so the first element of x.parts will then be the subset in the partition of x that uses the most memory.

**See also**

*partition*

x.**pathsin :** *Paths*

The paths from the direct referrers of the objects in x.

x.**pathsout :** *Paths*

The paths to the referents of the objects in x.

x.**referents :** *IdentitySet*

The set of objects that are directly referred to by any of the objects in x.

x.**referrers :** *IdentitySet*

The set of objects that directly refer to any of the objects in x.

x.**rp :** *ReferencePattern*

The current pattern of references to the objects in x.

**See also**

*get_rp*

x.**shpaths :** *Paths*

An object containing the shortest paths to objects in x.

**See also**

*get_shpaths*

x.**stat :** *Stat*

An object summarizing the statistics of the partitioning of x. This is useful when only the statistics is required, not the objects themselves. The statistics can be dumped to a file, unlike the set of objects itself.

# Bibliography


[1] Guppy-PE: A Python Programming Environment. http://guppy-pe.sourceforge.net.

[2] IPython: an enhanced interactive Python shell. http://ipython.scipy.org/.

[3] The Official Python Programming Language Website. http://www.python.org.

[4] PySizer - a memory profiler for Python. http://pysizer.8325.org.

[5] Python Memory Validator - Overview. http://www.softwareverify.com/pythonMemoryValidator/index.html.

[6] Wikipedia, the free encyclopedia. http://en.wikipedia.org.

[7] S. Alstrup, J. Clausen, and K. Jorgensen. An $O(|V| * |E|)$ algorithm for finding immediate multiple-vertex dominators. Department of Computer Science, University of Copenhagen, 1996.

[8] Borland. Borland optimizeit suite 5.5 feature matrix. http://www.borland.pl/optimizeit/opt55_suite_wlasciwosci.pdf, 2003.

[9] J. Campan and E. Muller. Performance tuning essentials for J2SE and J2EE. http://www.borland.com/resources/en/pdf/white_papers/opt_java_memory_leaks.pdf, 2002.

[10] N. Christofides. *GRAPH THEORY An Algorithmic Approach*. Academic Press, 1975.

[11] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, (1):269, 1959.

[12] ej technologies. Jprofiler reference manual. http://resources.ej-technologies.com/jprofiler/help/doc/help.pdf.

[13] R. P. Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction*. Addison-Wesley, 1994.

[14] C. Häggström. Detection of memory allocation bugs on system level. Master's thesis, Dept of Computing Science, Umeå University, 2005.

[15] M. Hauswirth and T. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *SIGPLAN Not. 39, 11*, pages 156–164, 2004.

[16] R. Jones and R. Lins. *Garbage Collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, 1996.

[17] W. D. Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. In *Proceedings of the ECOOP '99 European Conference on Object-oriented Programming*, pages 116–134, 1999.

[18] L. Råde and B. Westergren. *Mathematics Handbook for Science and Engineering*. Studentlitteratur, 1995.

[19] N. Rojemo and C. Runciman. Lag, drag, void and use - heap profiling and space-efficient compilation revisited. In *International Conference on Functional Programming*, pages 34–41, 1996.

[20] C. Runciman and N. Rojemo. New dimensions in heap profiling. *Journal of Functional Programming*, 6(4):587–620, July 1996.

[21] M. Serrano and H.-J. Boehm. Understanding memory allocation of Scheme programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 245–256. ACM Press, 2000.

[22] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 104–113. ACM Press, 2001.

[23] A. Walenstein. *Cognitive Support In Software Engineering Tools: A Distributed Cognition Framework*. PhD thesis, School of Computing Science, Simon Fraser University, 2002.

All links vere functional per 8 Mar 2006.