Profiler internals (rough draft)

Nick Smallbone

November 11, 2005

Contents

1 Introduction **2**

2 The scanner **2**

2.1 Collecting objects 2

2.2 Constructing wrapper types 5

2.2.1 Built-in wrapper types 7

2.2.2 CPython-specific wrapper types 7

2.2.3 Other CPython-specific code 8

3 Possible extensions **8**

3.1 Virtual objects 8

3.2 Overlapping objects 9

1 Introduction

This document shows how the profiler works internally, a little bit. The details of it aren't going to be correct forever: the design is still changing quite quickly.

2 The scanner

The scanner's job is to start from a set of roots, find as many objects as it can reach from those, and collect as much information as it can about each object into a wrapper.

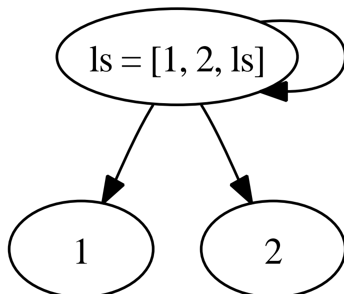
It's the only part of the profiler that should poke objects directly—everything else should just look at the wrappers. So it's important that each wrapper contains enough information.

2.1 Collecting objects

The best way to show how the scanner works is probably through an example. We'll look at what the scanner does when asked to scan `ls`, where `ls` is defined as:

```
ls = [1,2]
ls.append(ls)
```

The references from `ls` look like:



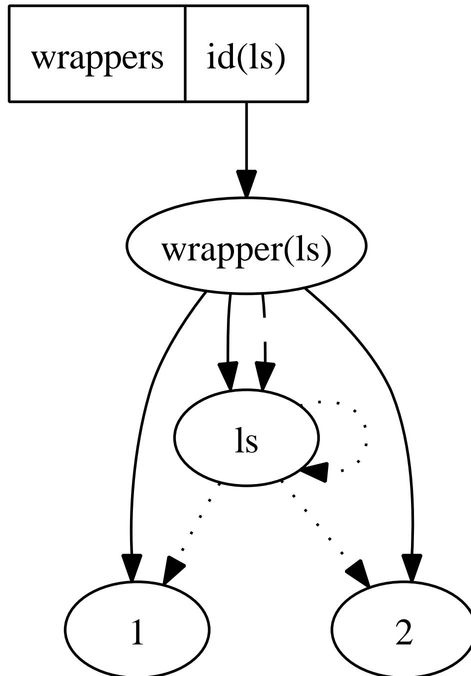
The scanner will be started by a call to `scanner.Objects(ls)`, which will call `scanner.scan(ls)`.

It keeps a queue of objects waiting to be scanned, and a dictionary which stores wrappers for already-scanned objects (it maps from `id(object)` to a wrapper).

To begin with, the dictionary will be empty and the queue will only contain `ls`.

The scanner will take `ls` off the queue, and find a class which can be used to count it. The next section talks about how it finds that class.

Then it makes an instance of that class, passing `ls` to the constructor. That will result in a wrapper, which it enters in the wrappers dictionary:



The wrapper contains a field `children`, which is a list of child objects referenced by the parent object. The solid arrows represent these here. The dotted arrows are references between the original objects. The dashed arrows are between a wrapper and its object.

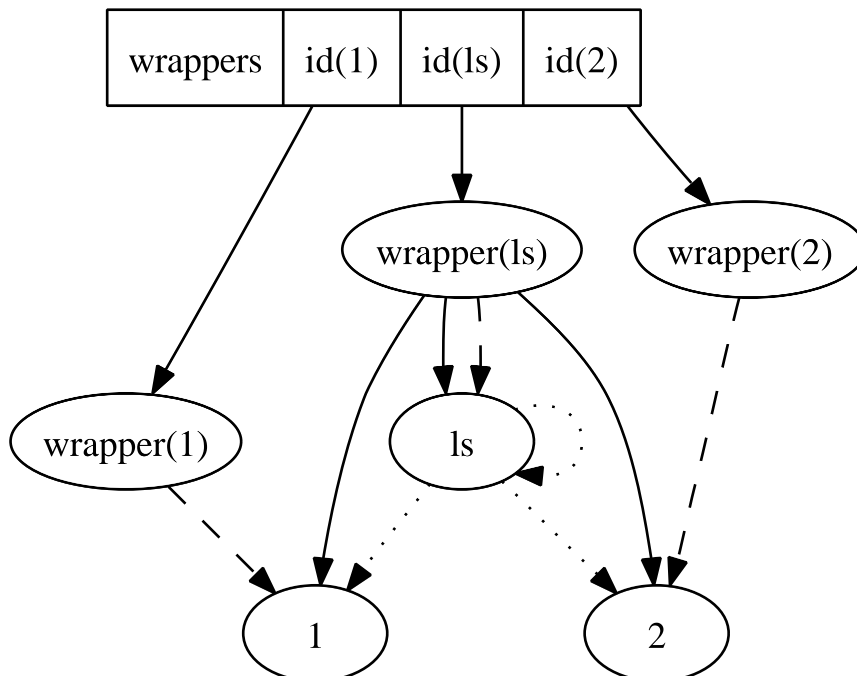
■ **Note:** A lot of important details of the wrapper have been left out—only what matters to the scanner is left. See the HappyDoc-documentation for details of what's there.

Now the scanner will add all of the object's children—`1`, `2` and `ls`—to the queue. It will then take `1` from the queue, and again will find a wrapper class, use that to make a new wrapper and add that wrapper to the wrappers dictionary.

It will do the same with `2`. Since `1` and `2` have no child references, nothing will be added to the queue.

Then it will take `ls` off the queue[1] and see that it's already been counted (since `id(ls)` is in the dictionary).

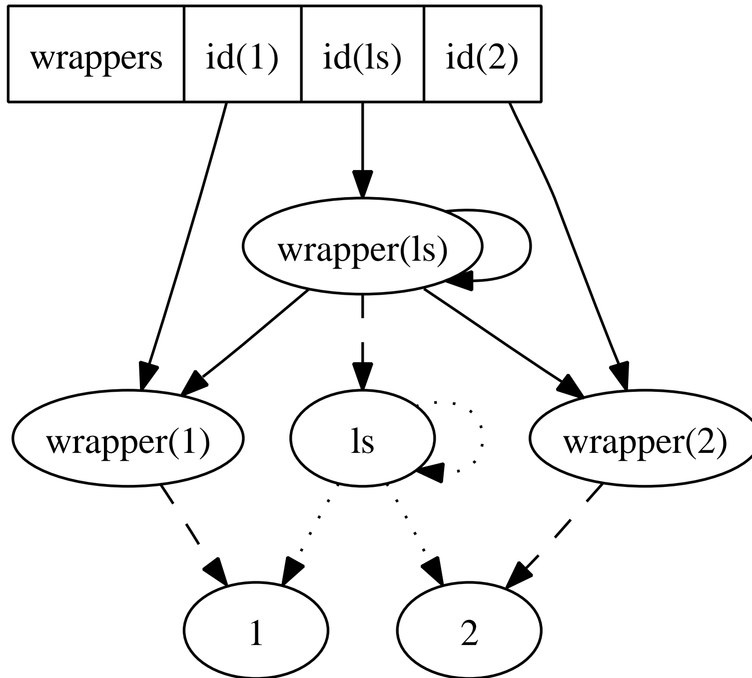
So the queue will become empty, and now the scanner will have built a dictionary of wrappers:



Here, `wrapper(ls)` contains references to `1`, `2` and `ls` directly.

The final step is to change these references to point at the wrappers rather than the objects. The scanner does this by calling the `liftrefs` method of each wrapper, which changes the references itself. This will result in:

[1] It was added along with 1 and 2.



In this way the scanner has found all objects that can be reached from `ls` and constructed a set of wrappers that "mirrors" the set of objects.

2.2 Constructing wrapper types

This section describes how the scanner finds a wrapper class for each object.

All necessary information is contained in the `sizes` module. There are four dictionaries which map from `id(something)` to wrapper class.

Here are the first three. If the `id` of the object, or type of the object, as appropriate, is found in one of these, the wrapper class associated with it is used without looking further. They are tried in this order:

- `sizes.object`—maps from object id to wrapper class.

  This is useful when some single objects need to be treated specially.

- `sizes.wholetype`—maps from type id to wrapper class.

  Subclasses of the type do not match this - only the exact type given. This dictionary is normally not filled manually, but by the scanner as it finds new types, using `sizes.type` (see below).

- `sizes.instance`—maps from type id to wrapper class. Each entry matches any (non-proper) subclass of the type id.

  This can be used to tag objects specially by deriving them from some class. For example, any object deriving from `wrapper.ignored` is given a size of 0 and no children by an entry in here.

If any of those three match, the wrapper class is found very quickly. But complex objects are possible, through inheritance. Every possible compound type can't be kept, so instead the scanner generates new wrapper classes from the entries in `sizes.type`.

Each entry is a wrapper class for a single type, excluding its base classes. The scanner, when given an object of an unknown type, will find its MRO[2], look up each class found there in `sizes.type` to get a list of wrapper classes, and make a new wrapper class which inherits from all those it found.

It will then put the new wrapper class in `sizes.wholetype`, so that it won't have to be generated again for this type.

For example (this is not a valid class, but never mind),

```
class ListDict(list, dict):
    pass
```

would, if it was valid, have a MRO of (list, dict, object). The scanner will find

```
sizes.type[id(list)] == ListObject
sizes.type[id(dict)] == DictObject
sizes.type[id(object)] == GCObject
```

and will construct a compound wrapper class:

```
class ListDictObject(ListObject, DictObject, GCObject):
    pass
sizes.wholetype[ListDict] = ListDictObject
```

and instantiate `ListDictObject` to get a wrapper for a `ListDict` instance.

There's one other complication: entries in the dictionaries don't *have* to be ids. They can also be direct object or type references (where these are hashable) or, in the case of types, the name of the type (useful where it's hard to find an instance of the type or the type itself to put into the dictionary).

[2] Method resolution order—a list of all superclasses.

For example, DictObject could be entered into `sizes.type[id(dict)]`, `sizes.type[dict]` or `sizes.type["__builtin__.dict"]`. When the scanner encounters it there it will turn it into the id form anyway.

2.2.1 Built-in wrapper types

These classes are defined in `wrapper.py`, and aren't specific (hopefully) to any particular implementation of Python:

- `ObjectWrapper`—this is the base class for all wrappers. It defines various functions which are used by the scanner, for example `liftrefs`. It defines enough that most subclasses only need to override the `__init__` function (to add type-specific code).
- `ZeroObject`—this is just `ObjectWrapper` in disguise. When given an object it will give it a size of 0 and no children.

2.2.2 CPython-specific wrapper types

These classes are defined in `cpython/rules.py` and `cpython/crules.pyx`, and are CPython-specific.

- `BaseObject`—this used to do something interesting, but now is yet another alias for `ObjectWrapper`...
- `SimpleObject`—using `str` on an instance of this class will print out the wrapped object (whereas `ObjectWrapper` will only mention the type).
- `GCObject`—instances of this class will ask the garbage collector what references the parent object has to find children. `object`'s entry in `sizes.type` uses this, so any type that doesn't go out of its way to do otherwise will use this.
- `SimpleGCObject`—a descendant of `SimpleObject` and `GCObject`.
- `Object`—counts any instance of `object`. It uses the object's type's `__basicsize__` as the size of the object.
- `ListObject`, `TupleObject`, `StringObject`, `UnicodeObject`, `DictObject`, `SetObject`, `FrameObject`, `SliceObject`, `InstanceObject`, `ModuleObject`, `DictIterObject`, `LongObject`, `CodeObject`, `FileWrapper`—wrapper classes for various types.

There is also an extension module, `utils.c`, which contains a few functions used by the wrapper classes.

2.2.3 Other CPython-specific code

`cpython/rules.py` contains a couple of other CPython-specific functions that are used by the rest of the profiler:

- `getname(t)`—given a type object, produce a vaguely sensible name for it. The scanner uses it to allow entering type names instead of types into the various `sizes` dictionaries.
- `roots()`—returns a list of objects which should be used as roots if none are specified in the call to `scan`. A sensible thing to use for this is `sys._getframe()`.

  The function must return roots only and not all objects! In other words, it should return a few objects from which everything else can be reached in a top-down way. This is so that the scanner can ignore wrappers - it will go no further if it encounters a wrapper object, but if `roots()` returns parts of a wrapper, the scanner will have no way of knowing and will count them anyway. And it's not very useful to be told that all the memory is being used by wrappers :-)

  So `gc.get_objects()` is *not* a suitable implementation of `roots()`.

3 Possible extensions

These aren't thought out very well yet—they're just ideas.

3.1 Virtual objects

The scanner only counts real Python objects—in CPython, ones which are represented by a `PyObject *`. ...It might be useful to be able to represent things like shared libraries as well.[3]

These could be represented by wrappers which didn't refer to a real object, but instead to some dummy object just used to give the wrapper *something* to refer to.

[3] Although perhaps not, since data from a program image normally uses copy-on-write.

In fact this can be done without changing the scanner, like so:

- Define a `VirtualObject` class which carries around with it size and reference information.
- Define a `VirtualWrapper` class which, when it is instantiated with a `VirtualObject`, just asks the `VirtualObject` for its size and references.
- Set `sizes.instance[VirtualObject] = VirtualWrapper`.

Then code which wants to reference a non-Python object can just give a `VirtualObject` instance as one of its children. The scanner will see the entry in `sizes.instance` and instantiate a `VirtualWrapper` on the object, and everything will just work :-).

Well, not quite. If the object is a shared library, it will contain definitions of other Python objects, so parts will be counted twice. This problem appears again below.

3.2 Overlapping objects

Some of the Python objects counted are tiny and it seems pedantic to count them separately.

For example, you might expect a wrapper for a module to contain references to everything defined at the top level of the module. Instead, it just contains a reference to the module's dictionary—and the dictionary contains the references to objects.

It would be nice to collapse the module and its dictionary into a single wrapper. But that can't be done yet, because there might be direct references to the dictionary.

I'm not sure how to fix that. Perhaps having views, or something like that, would help... in any case, a good solution to this would also fix the problem with virtual objects above, since a raw C object containing a Python object somewhere could be represented as a virtual object overlapping with a Python object.