objgraph Documentation

Release 3.4.0

Marius Gedminas

Feb 13, 2018

Contents

`objgraph` is a module that lets you visually explore Python object graphs.

You'll need graphviz if you want to draw the pretty graphs.

I recommend xdot for interactive use. `pip install xdot` should suffice; objgraph will automatically look for it in your `PATH`.

CHAPTER 1

Installation and Documentation

`pip install objgraph` or download it from PyPI.

Documentation lives at https://mg.pov.lt/objgraph.

CHAPTER 2

Quick start

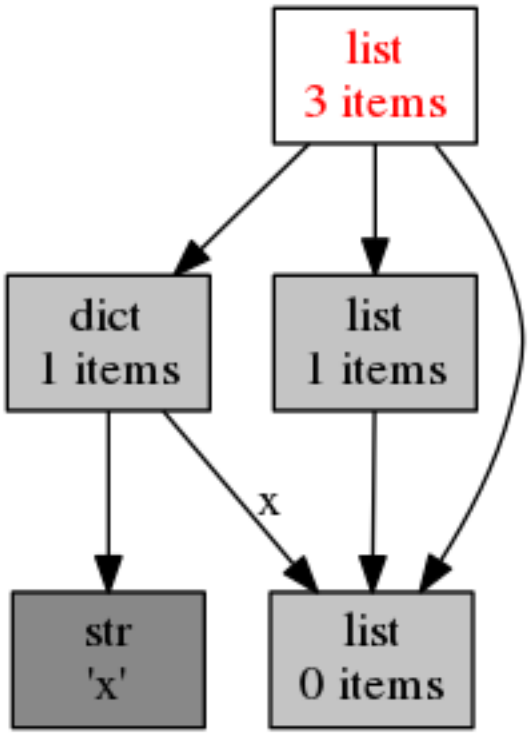
Try this in a Python shell:

```
>>> x = []
>>> y = [x, [x], dict(x=x)]
>>> import objgraph
>>> objgraph.show_refs([y], filename='sample-graph.png')
Graph written to ....dot (... nodes)
Image generated as sample-graph.png
```

(If you've installed `xdot`, omit the filename argument to get the interactive viewer.)

You should see a graph like this:

If you prefer to handle your own file output, you can provide a file object to the `output` parameter of `show_refs` and `show_backrefs` instead of a filename. The contents of this file will contain the graph source in DOT format.


list
3 items
dict
1 items
list
1 items
x
str
'x'
list
0 items

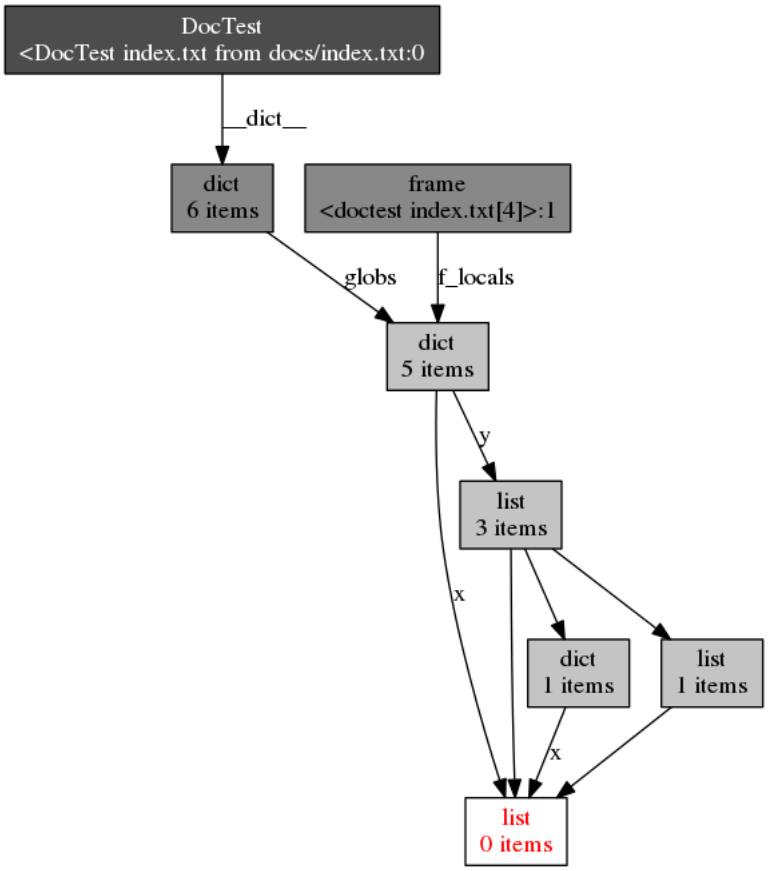
CHAPTER 3

Backreferences

Now try

```
>>> objgraph.show_backrefs([x], filename='sample-backref-graph.png')
...
Graph written to ....dot (8 nodes)
Image generated as sample-backref-graph.png
```

and you'll see


DocTest
<DocTest index.txt from docs/index.txt:0
__dict__
dict
6 items
frame
<doctest index.txt[4]>:1
globs
f_locals
dict
5 items
y
list
3 items
x
dict
1 items
list
1 items
x
list
0 items

CHAPTER 4

Memory leak example

The original purpose of *objgraph* was to help me find memory leaks. The idea was to pick an object in memory that shouldn't be there and then see what references are keeping it alive.

To get a quick overview of the objects in memory, use the imaginatively-named *show_most_common_types()*:

```
>>> objgraph.show_most_common_types()
tuple                      5224
function                   1329
wrapper_descriptor         967
dict                       790
builtin_function_or_method 658
method_descriptor          340
weakref                    322
list                       168
member_descriptor          167
type                       163
```

But that's looking for a small needle in a large haystack. Can we limit our haystack to objects that were created recently? Perhaps.

Let's define a function that "leaks" memory

```
>>> class MyBigFatObject(object):
...     pass
...
>>> def computate_something(_cache={}):
...     _cache[42] = dict(foo=MyBigFatObject(),
...                       bar=MyBigFatObject())
...     # a very explicit and easy-to-find "leak" but oh well
...     x = MyBigFatObject() # this one doesn't leak
```

We take a snapshot of all the objects counts that are alive before we call our function

```
>>> objgraph.show_growth(limit=3)
tuple                  5228     +5228
```

```
function                1330     +1330
wrapper_descriptor       967      +967
```

and see what changes after we call it

```
>>> computate_something()
>>> objgraph.show_growth()
MyBigFatObject             2        +2
dict                     797        +1
```

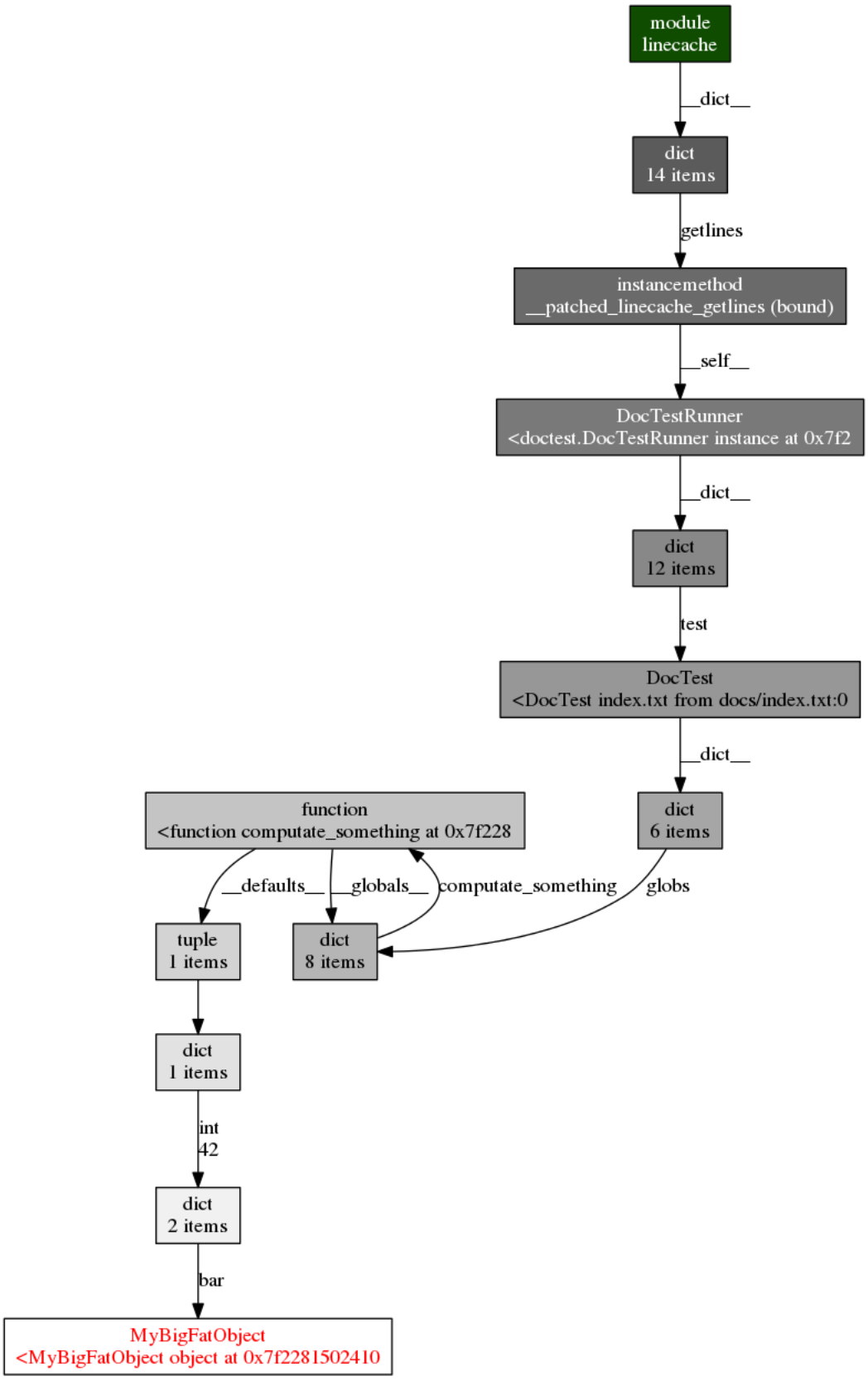
It's easy to see `MyBigFatObject` instances that appeared and were not freed. I can pick one of them at random and trace the reference chain back to one of the garbage collector's roots.

For simplicity's sake let's assume all of the roots are modules. `objgraph` provides a function, *`is_proper_module()`*, to check this. If you've any examples where that isn't true, I'd love to hear about them (although see *Reference counting bugs*).

```
>>> import random
>>> objgraph.show_chain(
...     objgraph.find_backref_chain(
...         random.choice(objgraph.by_type('MyBigFatObject')),
...         objgraph.is_proper_module),
...     filename='chain.png')
Graph written to ...dot (13 nodes)
Image generated as chain.png
```

It is perhaps surprising to find `linecache` at the end of that chain (apparently `doctest` monkey-patches it), but the important things – `computate_something()` and its cache dictionary – are in there.

There are other tools, perhaps better suited for memory leak hunting: heapy, Dozer.


module
linecache
__dict__
dict
14 items
getlines
instancemethod
__patched_linecache_getlines (bound)
__self__
DocTestRunner
<doctest.DocTestRunner instance at 0x7f2
__dict__
dict
12 items
test
DocTest
<DocTest index.txt from docs/index.txt:0
__dict__
function
<function computate_something at 0x7f228
dict
6 items
__defaults__
__globals__
computate_something
globs
tuple
1 items
dict
8 items
dict
1 items
int
42
dict
2 items
bar
MyBigFatObject
<MyBigFatObject object at 0x7f2281502410

CHAPTER 5

Reference counting bugs

Bugs in C-level reference counting may leave objects in memory that do not have any other objects pointing at them. You can find these by calling *get_leaking_objects()*, but you'll have to filter out legitimate GC roots from them, and there are a *lot* of those:

```
>>> roots = objgraph.get_leaking_objects()
>>> len(roots)
4621
```

```
>>> objgraph.show_most_common_types(objects=roots)
...
tuple          4333
dict           171
list           74
instancemethod 4
listiterator   2
MemoryError    1
Sub            1
RuntimeError   1
Param          1
Add            1
```

```
>>> objgraph.show_refs(roots[:3], refcounts=True, filename='roots.png')
...
Graph written to ...dot (19 nodes)
Image generated as roots.png
```

CHAPTER 6

API Documentation

6.1 objgraph

Tools for drawing Python object reference graphs with graphviz.

You can find documentation online at https://mg.pov.lt/objgraph/

6.1.1 Statistics

objgraph.**count**(*typename*[, *objects*])

Count objects tracked by the garbage collector with a given class name.

The class name can optionally be fully qualified.

Example:

```
>>> count('dict')
42
>>> count('mymodule.MyClass')
2
```

Note: The Python garbage collector does not track simple objects like int or str. See https://docs.python.org/3/library/gc.html#gc.is_tracked for more information.

Instead of looking through all objects tracked by the GC, you may specify your own collection, e.g.

```
>>> count('MyClass', get_leaking_objects())
3
```

See also: *get_leaking_objects()*.

Changed in version 1.7: New parameter: `objects`.

Changed in version 1.8: Accepts fully-qualified type names (i.e. 'package.module.ClassName') as well as short type names (i.e. 'ClassName').

`objgraph.`**`typestats`**(*[objects, shortnames=True]*)
Count the number of instances for each type tracked by the GC.

Note that the GC does not track simple objects like int or str.

Note that classes with the same name but defined in different modules will be lumped together if `shortnames` is True.

If `filter` is specified, it should be a function taking one argument and returning a boolean. Objects for which `filter(obj)` returns `False` will be ignored.

Example:

```
>>> typestats()
{'list': 12041, 'tuple': 10245, ...}
>>> typestats(get_leaking_objects())
{'MemoryError': 1, 'tuple': 2795, 'RuntimeError': 1, 'list': 47, ...}
```

New in version 1.1.

Changed in version 1.7: New parameter: `objects`.

Changed in version 1.8: New parameter: `shortnames`.

Changed in version 3.1.3: New parameter: `filter`.

`objgraph.`**`most_common_types`**(*[limit=10, objects, shortnames=True]*)
Count the names of types with the most instances.

Returns a list of (type_name, count), sorted most-frequent-first.

Limits the return value to at most `limit` items. You may set `limit` to None to avoid that.

If `filter` is specified, it should be a function taking one argument and returning a boolean. Objects for which `filter(obj)` returns `False` will be ignored.

The caveats documented in *typestats()* apply.

Example:

```
>>> most_common_types(limit=2)
[('list', 12041), ('tuple', 10245)]
```

New in version 1.4.

Changed in version 1.7: New parameter: `objects`.

Changed in version 1.8: New parameter: `shortnames`.

Changed in version 3.1.3: New parameter: `filter`.

`objgraph.`**`show_most_common_types`**(*[limit=10, objects, shortnames=True, file=sys.stdout]*)
Print the table of types of most common instances.

If `filter` is specified, it should be a function taking one argument and returning a boolean. Objects for which `filter(obj)` returns `False` will be ignored.

The caveats documented in *typestats()* apply.

Example:

```
>>> show_most_common_types(limit=5)
tuple                      8959
function                   2442
wrapper_descriptor         1048
dict                       953
builtin_function_or_method 800
```

New in version 1.1.

Changed in version 1.7: New parameter: `objects`.

Changed in version 1.8: New parameter: `shortnames`.

Changed in version 3.0: New parameter: `file`.

Changed in version 3.1.3: New parameter: `filter`.

objgraph.**growth**([*limit=10*, *peak_stats={}*, *shortnames=True*, *filter=None*])
Count the increase in peak object since last call.

Returns a list of (type_name, total_count, increase_delta), descending order by increase_delta.

Limits the output to `limit` largest deltas. You may set `limit` to None to see all of them.

Uses and updates `peak_stats`, a dictionary from type names to previously seen peak object counts. Usually you don't need to pay attention to this argument.

If `filter` is specified, it should be a function taking one argument and returning a boolean. Objects for which `filter(obj)` returns `False` will be ignored.

The caveats documented in *typestats()* apply.

Example:

```
>>> growth(2)
[(tuple, 12282, 10), (dict, 1922, 7)]
```

New in version 3.3.0.

objgraph.**show_growth**([*limit=10*, *peak_stats={}*, *shortnames=True*, *file=sys.stdout*, *filter=None*])
Show the increase in peak object counts since last call.

if `peak_stats` is None, peak object counts will recorded in func *growth*, and your can record the counts by yourself with set `peak_stats` to a dictionary.

The caveats documented in *growth()* apply.

Example:

```
>>> show_growth()
wrapper_descriptor       970       +14
tuple                  12282       +10
dict                    1922        +7
...
```

New in version 1.5.

Changed in version 1.8: New parameter: `shortnames`.

Changed in version 2.1: New parameter: `file`.

Changed in version 3.1.3: New parameter: `filter`.

objgraph.**get_new_ids**([*skip_update=False*, *limit=10*, *sortby='deltas'*, *shortnames=True*, *file=sys.stdout*])

Find and display new objects allocated since last call.

Shows the increase in object counts since last call to this function and returns the memory address ids for new objects.

Returns a dictionary mapping object type names to sets of object IDs that have been created since the last time this function was called.

`skip_update` (bool): If True, returns the same dictionary that was returned during the previous call without updating the internal state or examining the objects currently in memory.

`limit` (int): The maximum number of rows that you want to print data for. Use 0 to suppress the printing. Use None to print everything.

`sortby` (str): This is the column that you want to sort by in descending order. Possible values are: 'old', 'current', 'new', 'deltas'

`shortnames` (bool): If True, classes with the same name but defined in different modules will be lumped together. If False, all type names will be qualified with the module name. If None (default), `get_new_ids` will remember the value from previous calls, so it's enough to prime this once. By default the primed value is True.

`_state` (dict): Stores old, current, and new_ids in memory. It is used by the function to store the internal state between calls. Never pass in this argument unless you know what you're doing.

The caveats documented in *`growth()`* apply.

When one gets new_ids from *`get_new_ids()`*, one can use *`at_addrs()`* to get a list of those objects. Then one can iterate over the new objects, print out what they are, and call *`show_backrefs()`* or *`show_chain()`* to see where they are referenced.

Example:

```
>>> _ = get_new_ids() # store current objects in _state
>>> _ = get_new_ids() # current_ids become old_ids in _state
>>> a = [0, 1, 2] # list we don't know about
>>> b = [3, 4, 5] # list we don't know about
>>> new_ids = get_new_ids(limit=3) # we see new lists
======================================================================
Type                      Old_ids  Current_ids      New_ids Count_Deltas
======================================================================
list                          324          326           +3           +2
dict                         1125         1125           +0           +0
wrapper_descriptor           1001         1001           +0           +0
======================================================================
>>> new_lists = at_addrs(new_ids['list'])
>>> a in new_lists
True
>>> b in new_lists
True
```

New in version 3.4.

6.1.2 Locating and Filtering Objects

objgraph.**get_leaking_objects**([*objects*])

Return objects that do not have any referents.

These could indicate reference-counting bugs in C code. Or they could be legitimate.

Note that the GC does not track simple objects like int or str.

New in version 1.7.

`objgraph.`**`by_type`**(*typename*[, *objects*])
Return objects tracked by the garbage collector with a given class name.

Example:

```
>>> by_type('MyClass')
[<mymodule.MyClass object at 0x...>]
```

Note that the GC does not track simple objects like int or str.

Changed in version 1.7: New parameter: `objects`.

Changed in version 1.8: Accepts fully-qualified type names (i.e. 'package.module.ClassName') as well as short type names (i.e. 'ClassName').

`objgraph.`**`at`**(*addr*)
Return an object at a given memory address.

The reverse of id(obj):

```
>>> at(id(obj)) is obj
True
```

Note that this function does not work on objects that are not tracked by the GC (e.g. ints or strings).

`objgraph.`**`at_addrs`**(*address_set*)
Return a list of objects for a given set of memory addresses.

The reverse of [id(obj1), id(obj2), ...]. Note that objects are returned in an arbitrary order.

When one gets `new_ids` from *`get_new_ids()`*, one can use this function to get a list of those objects. Then one can iterate over the new objects, print out what they are, and call *`show_backrefs()`* or *`show_chain()`* to see where they are referenced.

```
>>> a = [0, 1, 2]
>>> new_ids = get_new_ids()
>>> new_lists = at_addrs(new_ids['list'])
>>> a in new_lists
True
```

Note that this function does not work on objects that are not tracked by the GC (e.g. ints or strings).

New in version 3.4.

`objgraph.`**`is_proper_module`**(*obj*)
Returns `True` if `obj` can be treated like a garbage collector root.

That is, if `obj` is a module that is in `sys.modules`.

```
>>> import types
>>> is_proper_module([])
False
>>> is_proper_module(types)
True
>>> is_proper_module(types.ModuleType('foo'))
False
```

New in version 1.8.

6.1.3 Traversing and Displaying Object Graphs

objgraph.**find_ref_chain**(*obj*, *predicate*[, *max_depth=20*, *extra_ignore=()*])
Find a shortest chain of references leading from obj.

The end of the chain will be some object that matches your predicate.

`predicate` is a function taking one argument and returning a boolean.

`max_depth` limits the search depth.

`extra_ignore` can be a list of object IDs to exclude those objects from your search.

Example:

```
>>> find_ref_chain(obj, lambda x: isinstance(x, MyClass))
[obj, ..., <MyClass object at ...>]
```

Returns `[obj]` if such a chain could not be found.

New in version 1.7.

objgraph.**find_backref_chain**(*obj*, *predicate*[, *max_depth=20*, *extra_ignore=()*])
Find a shortest chain of references leading to obj.

The start of the chain will be some object that matches your predicate.

`predicate` is a function taking one argument and returning a boolean.

`max_depth` limits the search depth.

`extra_ignore` can be a list of object IDs to exclude those objects from your search.

Example:

```
>>> find_backref_chain(obj, is_proper_module)
[<module ...>, ..., obj]
```

Returns `[obj]` if such a chain could not be found.

Changed in version 1.5: Returns `obj` instead of `None` when a chain could not be found.

objgraph.**show_chain**(*chain*[, ..., *highlight=None*, *filename=None*, *extra_info=None*, *refcounts=False*, *shortnames=True*])
Show a chain (or several chains) of object references.

Useful in combination with *find_ref_chain()* or *find_backref_chain()*, e.g.

```
>>> show_chain(find_backref_chain(obj, is_proper_module))
```

You can specify if you want that chain traced backwards or forwards by passing a `backrefs` keyword argument, e.g.

```
>>> show_chain(find_ref_chain(obj, is_proper_module),
...            backrefs=False)
```

Ideally this shouldn't matter, but for some objects `gc.get_referrers()` and `gc.get_referents()` are not perfectly symmetrical.

You can specify `highlight`, `extra_info`, `refcounts`, `shortnames`, `filename` or `output` arguments like for *show_backrefs()* or *show_refs()*.

New in version 1.5.

Changed in version 1.7: New parameter: `backrefs`.

Changed in version 2.0: New parameter: `output`.

objgraph.**show_backrefs**(*objs*[, *max_depth=3*, *extra_ignore=()*, *filter=None*, *too_many=10*, *highlight=None*, *filename=None*, *extra_info=None*, *refcounts=False*, *shortnames=True*])

Generate an object reference graph ending at `objs`.

The graph will show you what objects refer to `objs`, directly and indirectly.

`objs` can be a single object, or it can be a list of objects. If unsure, wrap the single object in a new list.

`filename` if specified, can be the name of a .dot or a image file, whose extension indicates the desired output format; note that output to a specific format is entirely handled by GraphViz: if the desired format is not supported, you just get the .dot file. If `filename` and `output` are not specified, `show_backrefs` will try to display the graph inline (if you're using IPython), otherwise it'll try to produce a .dot file and spawn a viewer (xdot). If xdot is not available, `show_backrefs` will convert the .dot file to a .png and print its name.

`output` if specified, the GraphViz output will be written to this file object. `output` and `filename` should not both be specified.

Use `max_depth` and `too_many` to limit the depth and breadth of the graph.

Use `filter` (a predicate) and `extra_ignore` (a list of object IDs) to remove undesired objects from the graph.

Use `highlight` (a predicate) to highlight certain graph nodes in blue.

Use `extra_info` (a function taking one argument and returning a string) to report extra information for objects.

Specify `refcounts=True` if you want to see reference counts. These will mostly match the number of arrows pointing to an object, but can be different for various reasons.

Specify `shortnames=False` if you want to see fully-qualified type names ('package.module.ClassName'). By default you get to see only the class name part.

Examples:

```
>>> show_backrefs(obj)
>>> show_backrefs([obj1, obj2])
>>> show_backrefs(obj, max_depth=5)
>>> show_backrefs(obj, filter=lambda x: not inspect.isclass(x))
>>> show_backrefs(obj, highlight=inspect.isclass)
>>> show_backrefs(obj, extra_ignore=[id(locals())])
```

Changed in version 1.3: New parameters: `filename`, `extra_info`.

Changed in version 1.5: New parameter: `refcounts`.

Changed in version 1.8: New parameter: `shortnames`.

Changed in version 2.0: New parameter: `output`.

objgraph.**show_refs**(*objs*[, *max_depth=3*, *extra_ignore=()*, *filter=None*, *too_many=10*, *highlight=None*, *filename=None*, *extra_info=None*, *refcounts=False*, *shortnames=True*])

Generate an object reference graph starting at `objs`.

The graph will show you what objects are reachable from `objs`, directly and indirectly.

`objs` can be a single object, or it can be a list of objects. If unsure, wrap the single object in a new list.

`filename` if specified, can be the name of a .dot or a image file, whose extension indicates the desired output format; note that output to a specific format is entirely handled by GraphViz: if the desired format is not supported, you just get the .dot file. If `filename` and `output` is not specified, `show_refs` will try to display the graph inline (if you're using IPython), otherwise it'll try to produce a .dot file and spawn a viewer (xdot). If xdot is not available, `show_refs` will convert the .dot file to a .png and print its name.

`output` if specified, the GraphViz output will be written to this file object. `output` and `filename` should not both be specified.

Use `max_depth` and `too_many` to limit the depth and breadth of the graph.

Use `filter` (a predicate) and `extra_ignore` (a list of object IDs) to remove undesired objects from the graph.

Use `highlight` (a predicate) to highlight certain graph nodes in blue.

Use `extra_info` (a function returning a string) to report extra information for objects.

Specify `refcounts=True` if you want to see reference counts.

Examples:

```
>>> show_refs(obj)
>>> show_refs([obj1, obj2])
>>> show_refs(obj, max_depth=5)
>>> show_refs(obj, filter=lambda x: not inspect.isclass(x))
>>> show_refs(obj, highlight=inspect.isclass)
>>> show_refs(obj, extra_ignore=[id(locals())])
```

New in version 1.1.

Changed in version 1.3: New parameters: `filename`, `extra_info`.

Changed in version 1.5: Follows references from module objects instead of stopping. New parameter: `refcounts`.

Changed in version 1.8: New parameter: `shortnames`.

Changed in version 2.0: New parameter: `output`.

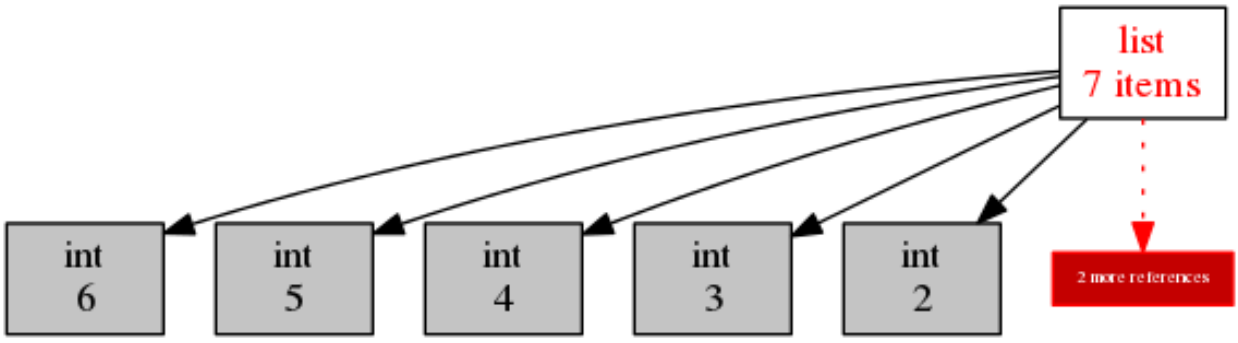
CHAPTER 7

More examples, that also double as tests

7.1 Too many references

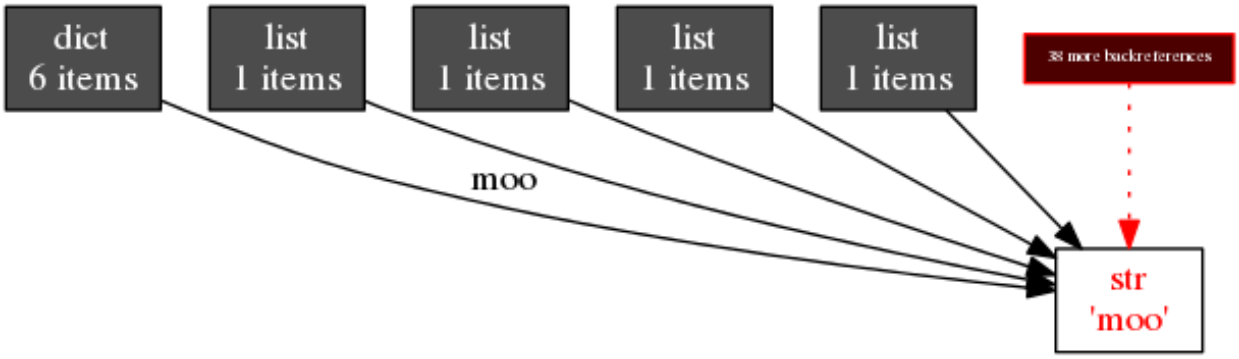
Objects that have too many references are truncated

```
>>> import objgraph
>>> objgraph.show_refs([list(range(7))], too_many=5, filename='too-many.png')
Graph written to ....dot (6 nodes)
Image generated as too-many.png
```



The same sort of thing applies to backreferences

```
>>> moo = 'moo'
>>> refs_to_moo = [[moo] for n in range(42)]
>>> objgraph.show_backrefs([moo], too_many=5, max_depth=1, filename='42.png')
Graph written to ....dot (6 nodes)
Image generated as 42.png
```



7.2 Reference counts

You can enable reference counts. The number of arrows pointing to an object should match the number in square brackets, usually, but there can be exceptions. E.g. objects internal to objgraph's implementation may inflate the reference count somewhat.

```
>>> import sys
>>> one_reference = object()
>>> objgraph.show_backrefs([one_reference], refcounts=True,
...     filename='refcounts.png')
Graph written to ....dot (5 nodes)
Image generated as refcounts.png
```

We see two references to the `one_reference` object: the one not shown comes from the list passed to show_backrefs.

I think the extra references to the frame object and locals dict come from the interpreter internals.

7.3 Extra information

You can add extra information to object graphs, if you desire. For example, let's add object IDs:

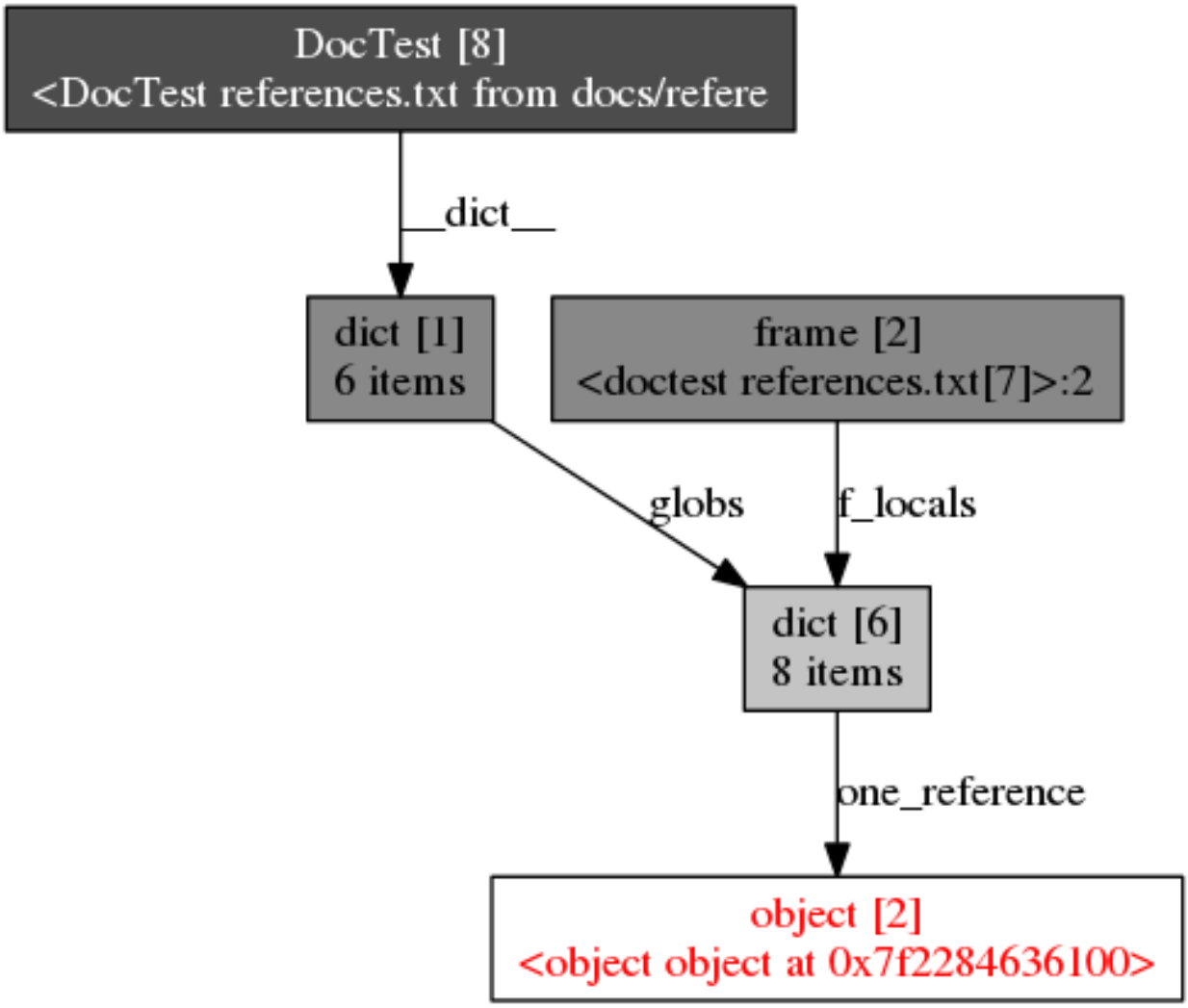
```
>>> x = []
>>> y = [x, [x], dict(x=x)]
>>> import objgraph
>>> objgraph.show_refs([y], extra_info=lambda x: hex(id(x)),
...                    filename='extra-info.png')
Graph written to ....dot (... nodes)
Image generated as extra-info.png
```

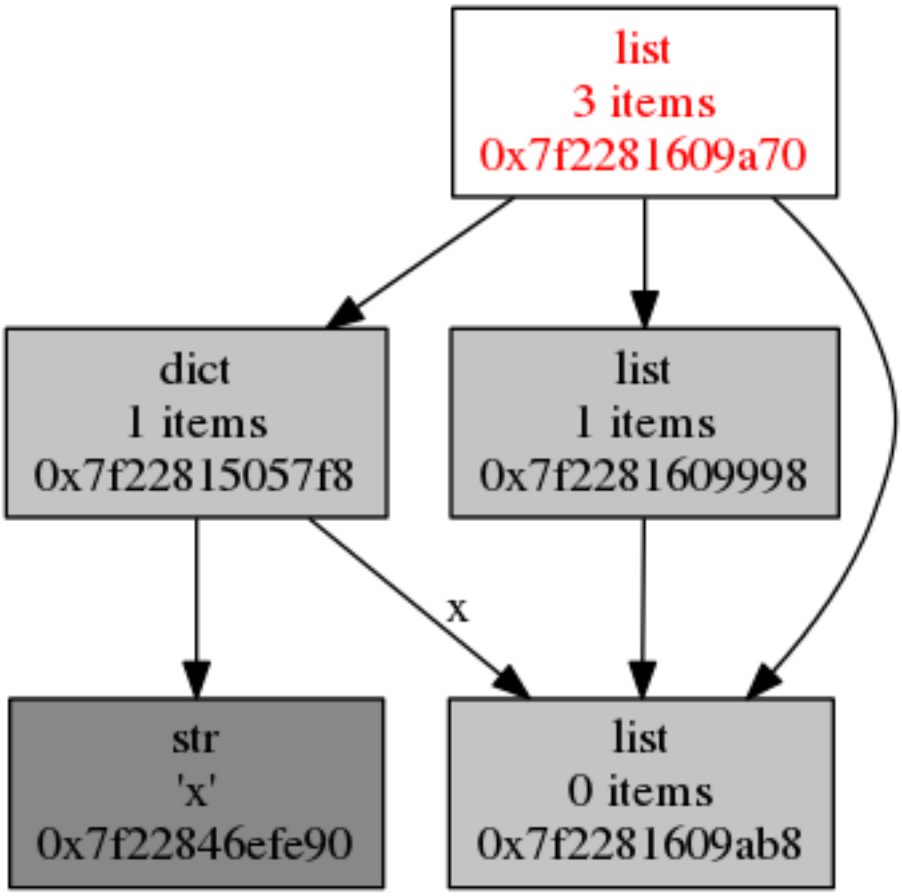
This way you can then look them up later with *at()*, if you desire to get a closer look at a particular object:

```
>>> objgraph.at(id(x)) is x
True
```

Warning: this doesn't work with strings or ints or other simple types that aren't tracked by the cyclic garbage collector:

```
>>> a = 'a string'
>>> objgraph.at(id(a))
```


DocTest [8]
<DocTest references.txt from docs/refere
__dict__
dict [1]
6 items
frame [2]
<doctest references.txt[7]>:2
globs
f_locals
dict [6]
8 items
one_reference
object [2]
<object object at 0x7f2284636100>


list
3 items
0x7f2281609a70
dict
1 items
0x7f22815057f8
list
1 items
0x7f2281609998
x
str
'x'
0x7f22846efe90
list
0 items
0x7f2281609ab8

7.4 Highlighting

You can highlight some graph nodes.

```
>>> class Node(object):
...     def __init__(self, *neighbours):
...         self.neighbours = list(neighbours)
>>> a = Node()
>>> b = Node(a)
>>> c = Node(b)
>>> d = Node(c)
>>> a.neighbours.append(d)
```

```
>>> import objgraph
>>> objgraph.show_backrefs(a, max_depth=15,
...     extra_ignore=[id(locals())],
...     highlight=lambda x: isinstance(x, Node),
...     filename='highlight.png')
Graph written to ....dot (12 nodes)
Image generated as highlight.png
```

7.5 Uncollectable garbage

Objects that have a `__del__` method cannot be collected by the garbage collector if they participate in a cycle, prior to Python 3.4.

```
>>> class Nondestructible(list):
...     def __del__(self):
...         pass
```

```
>>> x = Nondestructible()
>>> y = []
>>> z = []
>>> x.append(y)
>>> y.append(z)
>>> z.append(x)
```

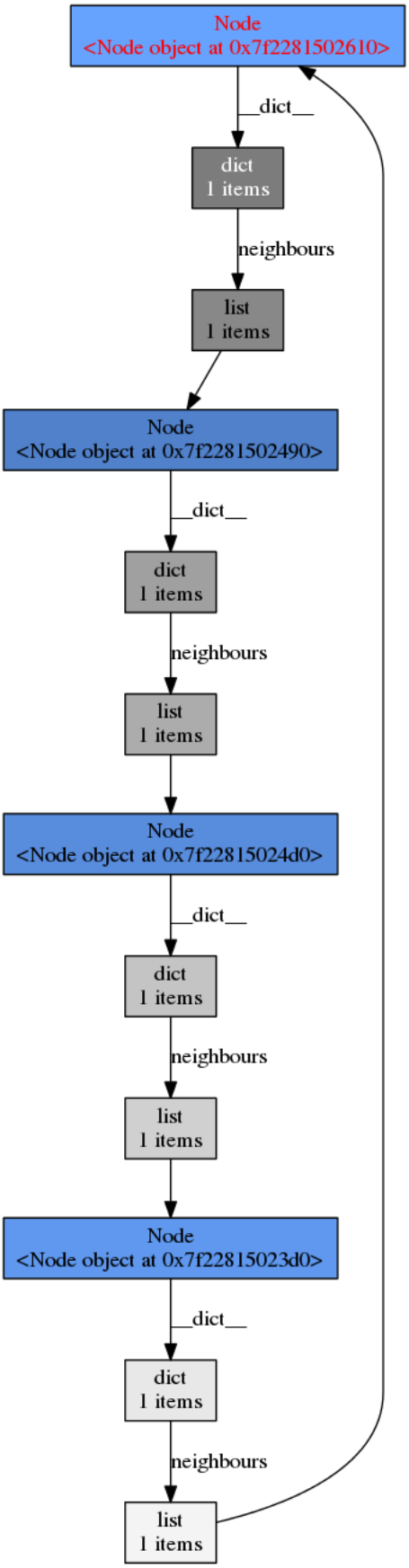
When you remove all other references to these, they end up in `gc.garbage`.

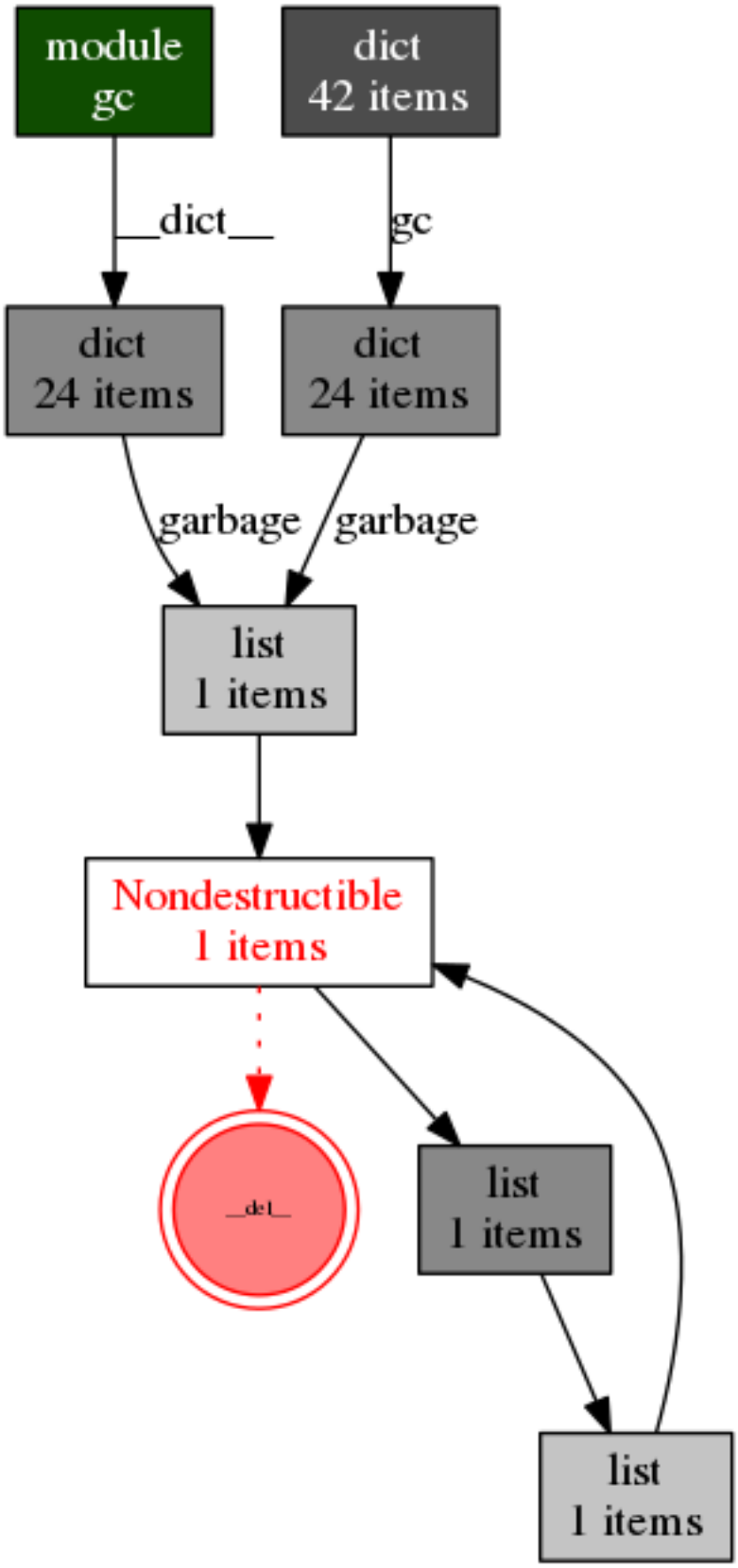
```
>>> import objgraph
>>> del x, y, z
>>> import gc
>>> _ = gc.collect()
>>> len(gc.garbage)
1
```

We highlight these objects by showing the existence of a `__del__`.

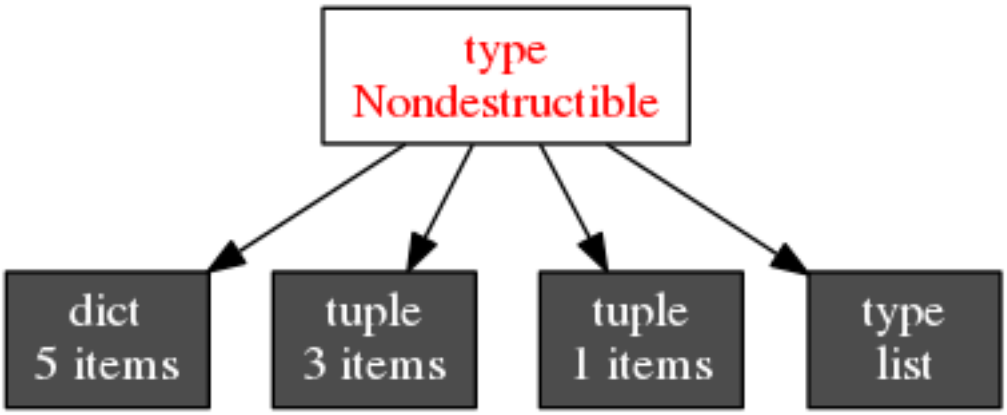
```
>>> objgraph.show_backrefs(objgraph.by_type('Nondestructible'),
...                        filename='finalizers.png')
Graph written to ....dot (8 nodes)
Image generated as finalizers.png
```

Note that classes that *define* a `__del__` method do not have this indicator


Node
<Node object at 0x7f2281502610>
__dict__
dict
1 items
neighbours
list
1 items
Node
<Node object at 0x7f2281502490>
__dict__
dict
1 items
neighbours
list
1 items
Node
<Node object at 0x7f22815024d0>
__dict__
dict
1 items
neighbours
list
1 items
Node
<Node object at 0x7f22815023d0>
__dict__
dict
1 items
neighbours
list
1 items


module
gc
dict
42 items
__dict__
gc
dict
24 items
dict
24 items
garbage
garbage
list
1 items
Nondestructible
1 items
__del__
list
1 items
list
1 items

```
>>> objgraph.show_refs(Nondestructible, max_depth=1,
...                    filename='class-with-finalizers.png')
Graph written to ....dot (5 nodes)
Image generated as class-with-finalizers.png
```



7.6 Stack frames and generators

Let's define a custom class

```
>>> class Canary(object):
...     pass
```

Suppose we've a generator that uses it

```
>>> def count_to_three():
...     tweety = Canary()
...     yield 1
...     yield 2
...     yield 3
```

and we make it active

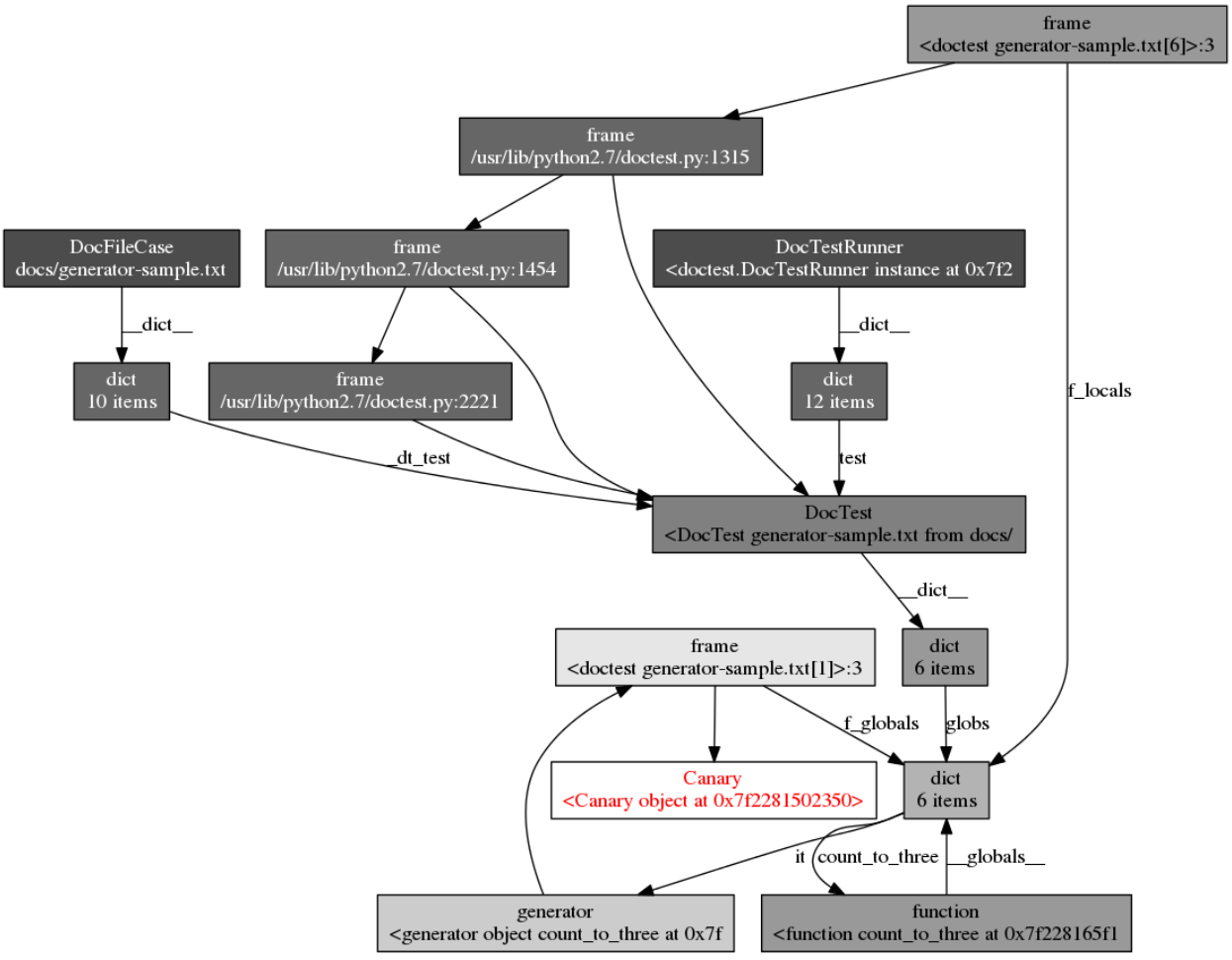
```
>>> it = count_to_three()
>>> next(it)
1
```

Now we can see that our Canary object is alive in memory

```
>>> import objgraph
>>> objgraph.count('Canary')
1
```

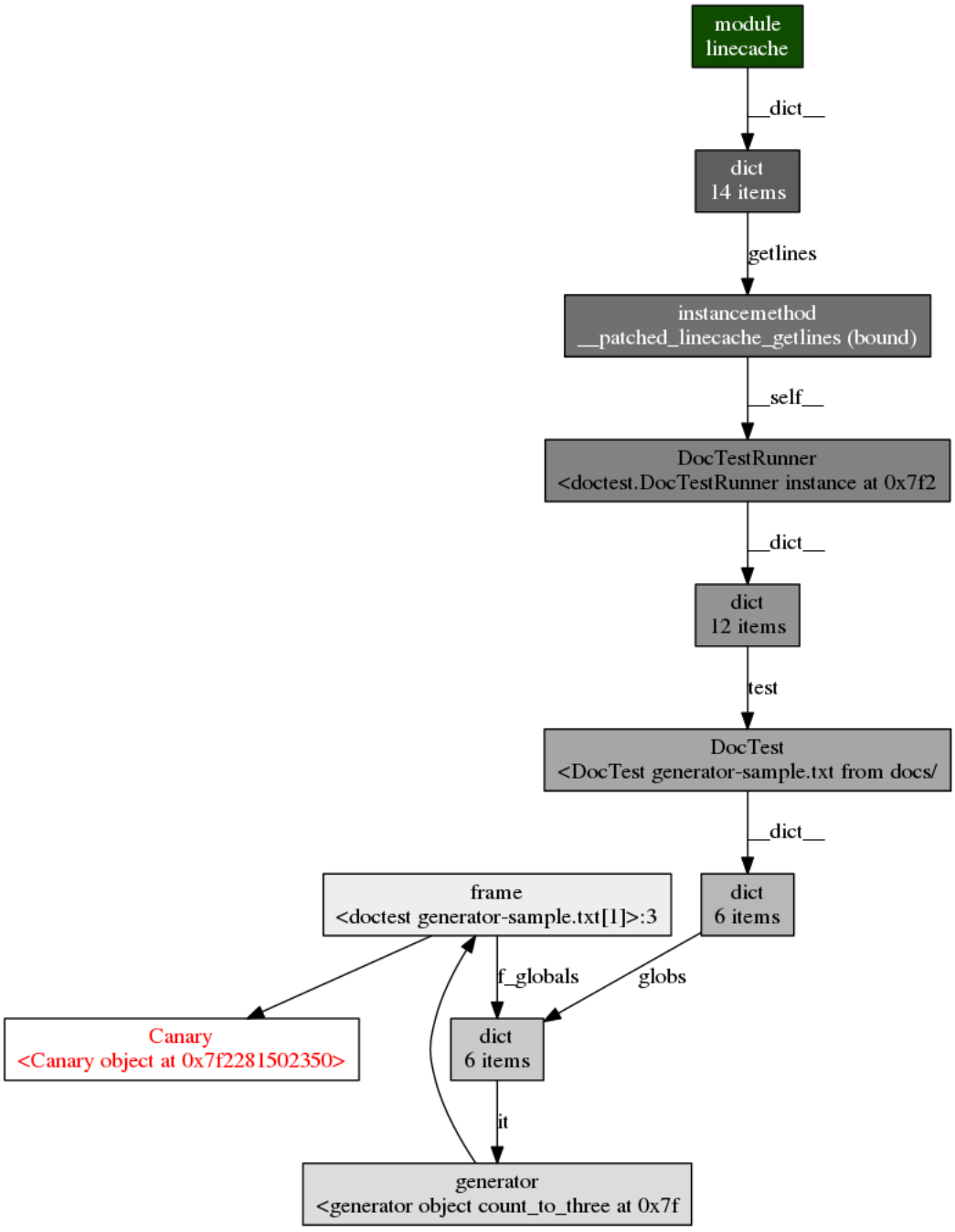
and we can see what holds it in memory

```
>>> objgraph.show_backrefs(objgraph.by_type('Canary'),
...                        max_depth=7,
...                        filename='canary.png')
Graph written to ....dot (15 nodes)
Image generated as canary.png
```


frame
<doctest generator-sample.txt[6]>:3
frame
/usr/lib/python2.7/doctest.py:1315
DocFileCase
docs/generator-sample.txt
frame
/usr/lib/python2.7/doctest.py:1454
DocTestRunner
<doctest.DocTestRunner instance at 0x7f2
__dict__
__dict__
dict
10 items
frame
/usr/lib/python2.7/doctest.py:2221
dict
12 items
f_locals
_dt_test
test
DocTest
<DocTest generator-sample.txt from docs/
__dict__
frame
<doctest generator-sample.txt[1]>:3
dict
6 items
f_globals
globs
Canary
<Canary object at 0x7f2281502350>
dict
6 items
it
count_to_three
__globals__
generator
<generator object count_to_three at 0x7f
function
<function count_to_three at 0x7f228165f1

Or we can examine just one of the reference chains leading straight to a module.

```
>>> objgraph.show_chain(
...     objgraph.find_backref_chain(objgraph.by_type('Canary')[0],
...                                 objgraph.is_proper_module),
...     filename='canary-chain.png')
Graph written to ....dot (11 nodes)
Image generated as canary-chain.png
```



To a first approximation, modules are garbage-collection roots, which makes the latter technique most useful.

7.7 Graph searches

The other day I was wondering why pickling a particular object errored out with an error deep in one of the subobjects.

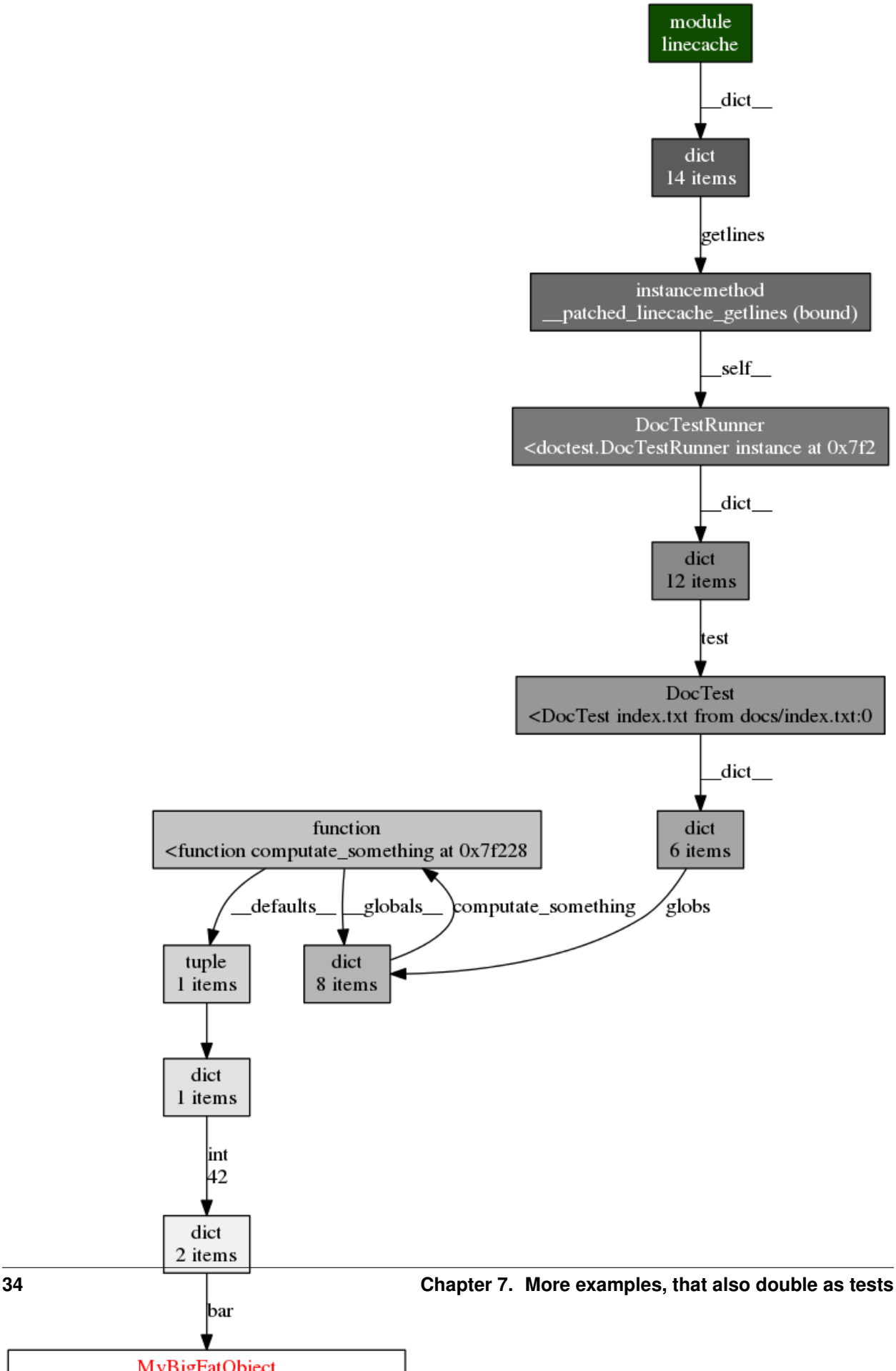
```
>>> class MyUnpicklableObject(object):
...     def __getstate__(self):
...         raise NotImplementedError
...
>>> my_object = dict(foo=dict(unrelated='things'),
...                  bar=[dict(nesting='fun'),
...                       dict(boobytrap=MyUnpicklableObject())])
>>> import objgraph
>>> objgraph.show_chain(
...     objgraph.find_ref_chain(
...         my_object,
...         lambda x: isinstance(x, MyUnpicklableObject)),
...     backrefs=False,
...     filename='forward-chain.png')
Graph written to ...dot (4 nodes)
Image generated as forward-chain.png
```

7.8 Quoting unsafe characters

Let's make sure our string quoting function can handle *anything*:

```
>>> import objgraph
>>> all_the_chars = dict((chr(i), i) for i in range(256))
>>> objgraph.show_refs(all_the_chars, too_many=600,
...                    filename='all-the-chars.dot')
Graph written to all-the-chars.dot (... nodes)
```

Trust me, you do not want to see the resulting image.


module
linecache
__dict__
dict
14 items
getlines
instancemethod
__patched_linecache_getlines (bound)
__self__
DocTestRunner
<doctest.DocTestRunner instance at 0x7f2
__dict__
dict
12 items
test
DocTest
<DocTest index.txt from docs/index.txt:0
__dict__
function
<function computate_something at 0x7f228
dict
6 items
__defaults__
__globals__
computate_something
globs
tuple
1 items
dict
8 items
dict
1 items
int
42
dict
2 items
bar

CHAPTER 8

History

I've developed a set of functions that eventually became objgraph when I was hunting for memory leaks in a Python program. The whole story – with illustrated examples – is in this series of blog posts:

- Hunting memory leaks in Python
- Python object graphs
- Object graphs with graphviz

And here's the change log

8.1 Changes

8.1.1 3.4.0 (2018-02-13)

- New functions: *get_new_ids()*, *at_addrs()*.

  Contributed by Justin Black in PR 36.

8.1.2 3.3.0 (2017-12-28)

- New function: *growth()*.

8.1.3 3.2.0 (2017-12-20)

- New `filter` argument for *typestats()*, *most_common_types()*, *show_most_common_types()*, *show_growth()*.
- Show lambda functions in a more human-friendly way.

8.1.4 3.1.2 (2017-11-27)

- Correct UTF-8 mojibake in the changelog and switch all links to HTTPS.

8.1.5 3.1.1 (2017-10-30)

- Add support for Python 3.6.
- Replace bare `except:` in `safe_repr()` with `except Exception:`.

8.1.6 3.1.0 (2016-12-07)

- Support displaying graphs inline in IPython/Jupyter notebooks (*issue 28 <https://github.com/mgedmin/objgraph/pull/28>*).

8.1.7 3.0.1 (2016-09-17)

- The `file` argument of *`show_most_common_types()`* and *`show_growth()`* now defaults to `None` instead of `sys.stdout`. `None` is interpreted to be the same as `sys.stdout`, which means the right stdout will be used if you change it at runtime (which happens, in doctests).

8.1.8 3.0.0 (2016-04-13)

- *`show_most_common_types()`* and *`show_growth()`* now accept a `file` argument if you want to redirect the output elsewhere.

  Fixes issue 24. Contributed by "d-sun-d".

- Don't trust `__class__` to be accurate and `__name__` to be a string. Fixes errors in some convoluted corner cases when mocks are involved.

  Contributed by Andrew Shannon Brown in PR 26.

- Drop support for Python 2.4, 2.5, and 2.6.
- Drop support for Python 3.1 and 3.2.
- Add support for Python 3.5.

8.1.9 2.0.1 (2015-07-28)

- Avoid creating reference cycles between the stack frame and the local `objects` variable in *`by_type()`*, *`count()`*, and *`typestats()`*.

  Fixes issue 22. Contributed by Erik Bray.

8.1.10 2.0.0 (2015-04-18)

- *`show_refs()`* and *`show_backrefs()`* now accept a file-like object (via the new `output` argument) as an alternative to a filename.

- Made internal helper methods private. This includes `find_chain()`, `show_graph()`, `obj_node_id()`, `obj_label()`, `quote()`, `long_typename()`, `safe_repr()`, `short_repr()`, `gradient()`, `edge_label()`, and `_program_in_path()`.
- Correctly determine the name of old-style classes in *`count()`*, *`by_type()`*, and graph drawing functions.

  Fixes issue 16. Contributed by Mike Lambert.

8.1.11 1.8.1 (2014-05-15)

- Do not expect file objects to have an `encoding` attribute. Makes objgraph compatible with Eventlet's monkey-patching.

  Fixes issue 6. Contributed by Jakub Stasiak.

8.1.12 1.8.0 (2014-02-13)

- Moved to GitHub.
- Python 3.4 support (LP#1270872).
- New function: *`is_proper_module()`*.
- New `shortnames` argument for *`typestats()`*, *`most_common_types()`*, *`show_most_common_types()`*, *`show_growth()`*, *`show_refs()`*, and *`show_backrefs()`*.

  `count()` and *`by_type()`* accept fully-qualified type names now.

  Fixes issue 4.

8.1.13 1.7.2 (2012-10-23)

- Bugfix: setup.py sdist was broken on Python 2.7 (UnicodeDecodeError in tarfile).
- The `filename` argument for *`show_refs()`* and *`show_backrefs()`* now allows arbitrary image formats, not just PNG. Patch by Riccardo Murri.
- Temporary dot files are now named *objgraph-*.dot* instead of *tmp*.dot*.
- Python 3.3 support: no code changes, but some tests started failing because the new and improved dictionary implementation no longer holds references to str objects used as dict keys.
- Added a tox.ini for convenient multi-Python testing.

8.1.14 1.7.1 (2011-12-11)

- Bugfix: non-ASCII characters in object representations would break graph generation on Python 3.x, in some locales (e.g. with LC_ALL=C). Reported and fixed by Stefano Rivera.
- Bugfix: setup.py was broken on Python 3.x
- Bugfix: dot.exe/xdot.exe were not found on Windows (LP#767239).
- Documentation updates: document the forgotten *`find_ref_chain()`*, update *`show_chain()`* prototype.

8.1.15 1.7.0 (2011-03-11)

- New function: *find_ref_chain()*.
- New `backrefs` argument for *show_chain()*.
- New function: *get_leaking_objects()*, based on a blog post by Kristján Valur.
- New `objects` argument for *count()*, *typestats()*, *most_common_types()*, *show_most_common_types()*, and *by_type()*.
- Edges pointing to function attributes such as __defaults__ or __globals__ are now labeled.
- Edge labels that are not simple strings now show the type.
- Bugfix: '0' and other unsafe characters used in a dictionary key could break graph generation.
- Bugfix: show_refs(..., filename='graph.dot') would then go to complain about unrecognized file types and then produce a png.

8.1.16 1.6.0 (2010-12-18)

- Python 3 support, thanks to Stefano Rivera (fixes LP#687601).
- Removed weird weakref special-casing.

8.1.17 1.5.1 (2010-12-09)

- Avoid test failures in uncollectable-garbage.txt (fixes LP#686731).
- Added HACKING.txt (later renamed to HACKING.rst).

8.1.18 1.5.0 (2010-12-05)

- Show frame objects as well (fixes LP#361704).
- New functions: *show_growth()*, *show_chain()*.
- *find_backref_chain()* returns `[obj]` instead of `None` when a chain could not be found. This makes `show_chain(find_backref_chain(...), ...)` not break.
- Show how many references were skipped from the output of *show_refs()*/*show_backrefs()* by specifying `too_many`.
- Make *show_refs()* descend into modules.
- Do not highlight classes that define a `__del__`, highlight only instances of those classes.
- Option to show reference counts in *show_refs()*/*show_backrefs()*.
- Add Sphinx documentation and a PyPI long description.

8.1.19 1.4.0 (2010-11-03)

- Compatibility with Python 2.4 and 2.5 (`tempfile.NamedTemporaryFile` has no `delete` argument).
- New function: *most_common_types()*.

8.1.20 1.3.1 (2010-07-17)

- Rebuild an sdist with no missing files (fixes LP#606604).
- Added MANIFEST.in and a Makefile to check that setup.py sdist generates source distributions with no files missing.

8.1.21 1.3 (2010-07-13)

- Highlight objects with a `__del__` method.
- Fixes LP#483411: suggest always passing `[obj]` to *`show_refs()`*, *`show_backrefs()`*, since obj might be a list/tuple.
- Fixes LP#514422: *`show_refs()`*, *`show_backrefs()`* don't create files in the current working directory any more. Instead they accept a filename argument, which can be a .dot file or a .png file. If None or not specified, those functions will try to spawn xdot as before.
- New extra_info argument to graph-generating functions (patch by Thouis Jones, LP#558914).
- setup.py should work with distutils now (LP#604430, thanks to Randy Heydon).

8.1.22 1.2 (2009-03-25)

- Project website, public source repository, uploaded to PyPI.
- No code changes.

8.1.23 1.1 (2008-09-10)

- New function: *`show_refs()`* for showing forward references.
- New functions: *`typestats()`* and *`show_most_common_types()`*.
- Object boxes are less crammed with useless information (such as IDs).
- Spawns xdot if it is available.

8.1.24 1.0 (2008-06-14)

- First public release.

CHAPTER 9

Support and Development

The source code can be found in this Git repository: https://github.com/mgedmin/objgraph.

To check it out, use `git clone https://github.com/mgedmin/objgraph`.

Report bugs at https://github.com/mgedmin/objgraph/issues.

For more information, see *Hacking on objgraph*.

9.1 Hacking on objgraph

Start by geting the latest source with

```
git clone https://github.com/mgedmin/objgraph
```

Run the test suite with

```
make test
```

The test suite is mostly smoke tests (i.e. crashes will be noticed, subtly wrong output will be missed). I hope to improve that in the future, but don't hold your breath. Most of the testing is done manually or semi-automatically, e.g. by running `make images` and eyeballing the results (imgdiff is handy there).

9.1.1 Sending me patches

GitHub pull requests are probably the best way to send me patches. Or just email them to <marius@gedmin.as>.

I'd appreciate issues in GitHub for each proposed change, be it a bug or a feature request.

9.1.2 Supported Python versions

Python 2.4 through 2.7, as well as 3.x.

You can run the test suite for all supported Python versions with

```
make test-all-pythons
```

or with detox (which will be faster, since it runs the tests in parallel).

If a test fails, often the easiest way to debug is is to compare the output visually

```
make images PYTHON=pythonX.Y
git config diff.imgdiff.command 'f() { imgdiff --eog -H $1 $2; }; f'
git diff docs/*.png
git checkout -- docs/*.png docs/*.dot
```

An easy way to get Pythons 2.4 through 2.7 (and 3.x) on Ubuntu is to use Felix Krull's "deadsnakes" PPA:

```
sudo add-apt-repository ppa:fkrull/deadsnakes
sudo apt-get update
sudo apt-get install python2.{4,5,6,7} python3.{1,2,3}
```

9.1.3 Test coverage

As I mentioned, the tests are mostly smoke tests, and even then they're incomplete. Install coverage to see how incomplete they are with

```
make coverage
```

I use a vim plugin to higlight lines not covered by tests while I edit

```
make coverage
vim objgraph.py
  :HighlightCoverage
```

If you prefer HTML reports, run

```
make coverage
coverage html
```

and then browse `htmlcov/index.html`.

9.1.4 Documentation

To fully rebuild the documentation, run

```
make clean images docs
```

Please `git checkout --` the png files that haven't changed significantly. (Many of the images include things like memory addresses which tend to change from run to run.)

imgdiff is useful for comparing the images with their older versions:

```
git config diff.imgdiff.command 'f() { imgdiff $1 $2; }; f'
git diff docs/*.png
```

It has a few options that may make the changes easier to see. I personally like:

```
git config diff.imgdiff.command 'f() { imgdiff --eog -H $1 $2; }; f'
git diff docs/*.png
```

When you add a new doctest file, remember to include it in `docs/index.txt`.

When you add a new function, make sure it has a PEP-257-compliant docstring and add the appropriate autodoc directive to `objgraph.txt`.

I insist on one departure from PEP-257: the closing `"""` should *not* be preceded by a blank line. Example:

```
def do_something():
    """Do something.

    Return something valuable.
    """
```

If Emacs is broken, fix emacs, do not make my docstrings ugly.

On the other hand, if the last thing in a docstring is an indented block quote or a doctest section, it should be surrounded by blank lines. Like this:

```
def do_something():
    """Do something.

    Return something valuable.

    Example:

        >>> do_something()
        42

    """
```

I find restview very handy for documentation writing: it lets me see how the text looks by pressing Ctrl-R in a browser window, without having to re-run any documentation building commands. The downside is that `restview` doesn't support Sphinx extensions to ReStructuredText, so you end up with error messages all over the place. Then again this is useful for bits that *can't* use Sphinx extensions, like the PyPI long description.

To preview the PyPI long description (which is generated by concatenating `README.rst` and `CHANGES.rst`) with `restview`, use this handy command:

```
make preview-pypi-description
```

because typing

```
restview -e "python setup.py --long-description"
```

is tedious, and bash has tab-completion for makefile rules.

9.1.5 Making releases

You need write access to the PyPI package and to the Git branch on GitHub. At the moment of this writing, this means you must be me.

Run `make release` and follow the instructions. It is safe to run this command at any time: it never commits/pushes/uploads to PyPI, it just tells you what to do.

9.1.6 Avoiding incomplete releases

It is important to keep MANIFEST.in up to date so that source tarballs generated with `python setup.py sdist` aren't missing any files, even if you don't have the right setuptools version control plugins installed. You can run

```
make distcheck
```

to be sure this is so, but it's not necessary – `make release` will do this every time.

(I've later written a standalone tool, check-manifest that can do this check for every Python package.)

Python Module Index

o

Index