Debugging and profiling techniques

By Giovanni Bajo <rasky@develer.com>

EuroPython 2011
Florence, June 22nd

cc BY NC SA

develer

Who I am

- 9 yrs Python experience
- Maintainer of PyInstaller
- Organizer of PyCon Italy & EuroPython
- CTO of Develer

Roadmap

- Memory leaks: definitions and basic analysis
- Memory leaks: garbage collection
- Python code profiling
- Debugging C/C++ extensions

Fair notice

Will focus on CPython almost exclusively

Roadmap

- **Memory leaks: definitions and basic analysis**
- Memory leaks: garbage collection
- Python code profiling
- Debugging C/C++ extensions

Memory vs Resource leaks

- Technically different
  - Memory: malloc w/o free
  - Resource: open w/o close
- Coindicents in Python
  - Memory: allocate through objects
  - Resource: cleanup through object destructors (eg: file.__del__ → file.close)

Object leak

- Object collection is supposed to be automatic
  - No references → collect
- Object leak: “logically” dead but not collected
  - “Lost”/”hidden” reference
  - C/C++ extension bugs
  - “Bug”/”QoI” in CPython core

Counting allocated objects

- gc.get_objects()
  - List of all Python objects
  - Easy per-type check
- Check “before & after”
  - Comparing length might be enough
  - If it grows → leak

Counting allocated objects

```
>>> class A: pass
...
>>> def howmany(cls):
...     return len([x for x in gc.get_objects() if isinstance(x,cls)])
...
>>> howmany(A)
0
>>> a = A()
>>> howmany(A)
1
>>> del a
>>> howmany(A)
0
```

“Lost” references

- sys.getrefcount(obj) [... -1!]
  - Check refcount before logical “destruction”
  - Call gc.collect(), before and after
  - If > 2, someone else is referencing
- Can be used in testsuites
  - You can test an object not to leak references
- How do I find a lost reference?

“Lost” references

- gc.get_referrers(*objs)
  - Objects with **direct** references to the specified objects
  - Often must recurse more levels
    - Instance's __dict__
    - containers
  - Try to recursively analyse (w/ debugger or print)

Example: reference analysys

```
>>> class A: pass
...
>>> a1 = A()
>>> a2 = A()
>>> sys.getrefcount(a1)-1
1

>>> pprint(gc.get_referrers(a1))
[{'A': <class __main__.A at 0x7f31f04a4770>,
  '__builtins__': <module '__builtin__' (built-in)>,
  '__doc__': None,
  '__name__': '__main__',
  'a1': <__main__.A instance at 0x7f31f04bd440>,
  'a2': <__main__.A instance at 0x7f31f04bd488>,
  'gc': <module 'gc' (built-in)>,
  'pprint': <function pprint at 0x7f31f04b69b0>,
  'sys': <module 'sys' (built-in)>}]
```

What's this dictionary?

Example: reference analysys

```
>>> gc.get_referrers(a1)[0] is sys._getframe().f_locals
True
>>> gc.get_referrers(a1)[0] is globals()
True
```

- The current “context” is availabile through sys
  - Usually, locals() or globals()

Example: reference analysys

```
>>> a1 = A()
>>> a2 = A()
>>> a2.xxx = a1
>>> sys.getrefcount(a1)-1
2

>>> pprint(gc.get_referrers(a1))
[{'xxx': <__main__.A instance at 0x7f7a78cb85f0>},
 {'A': <class __main__.A at 0x7f7a7d68be90>,
  '__builtins__': <module '__builtin__' (built-in)>,
  '__doc__': None,
  '__name__': '__main__',
  'a1': <__main__.A instance at 0x7f7a78cb85f0>,
  'a2': <__main__.A instance at 0x7f7a78cb8638>,
  'gc': <module 'gc' (built-in)>,
  'pprint': <function pprint at 0x7f7a78cbc230>}]

>>> gc.get_referrers(a1)[0] is a2.__dict__
True
>>> gc.get_referrers(a2.__dict__)[0] is a2
True
```

Roadmap

- Memory leak: definizioni ed analisi base
- **Memory leak: garbage collector**
- Profiling di codice Python
- Debugging di estensioni C/C++

CPython: reference counting

- CPython uses refcount
- **Immediate** destruction when count gets to 0
  - Explicit: “del a.ref”
  - Implicit: function exit
  - Guaranteed by CPython (“forever”)
    - Not officially Python
- You can rely on it if you target CPython

CPython: reference counting

```
def readfile(fn):
    f = open(fn)
    return f.read()
```

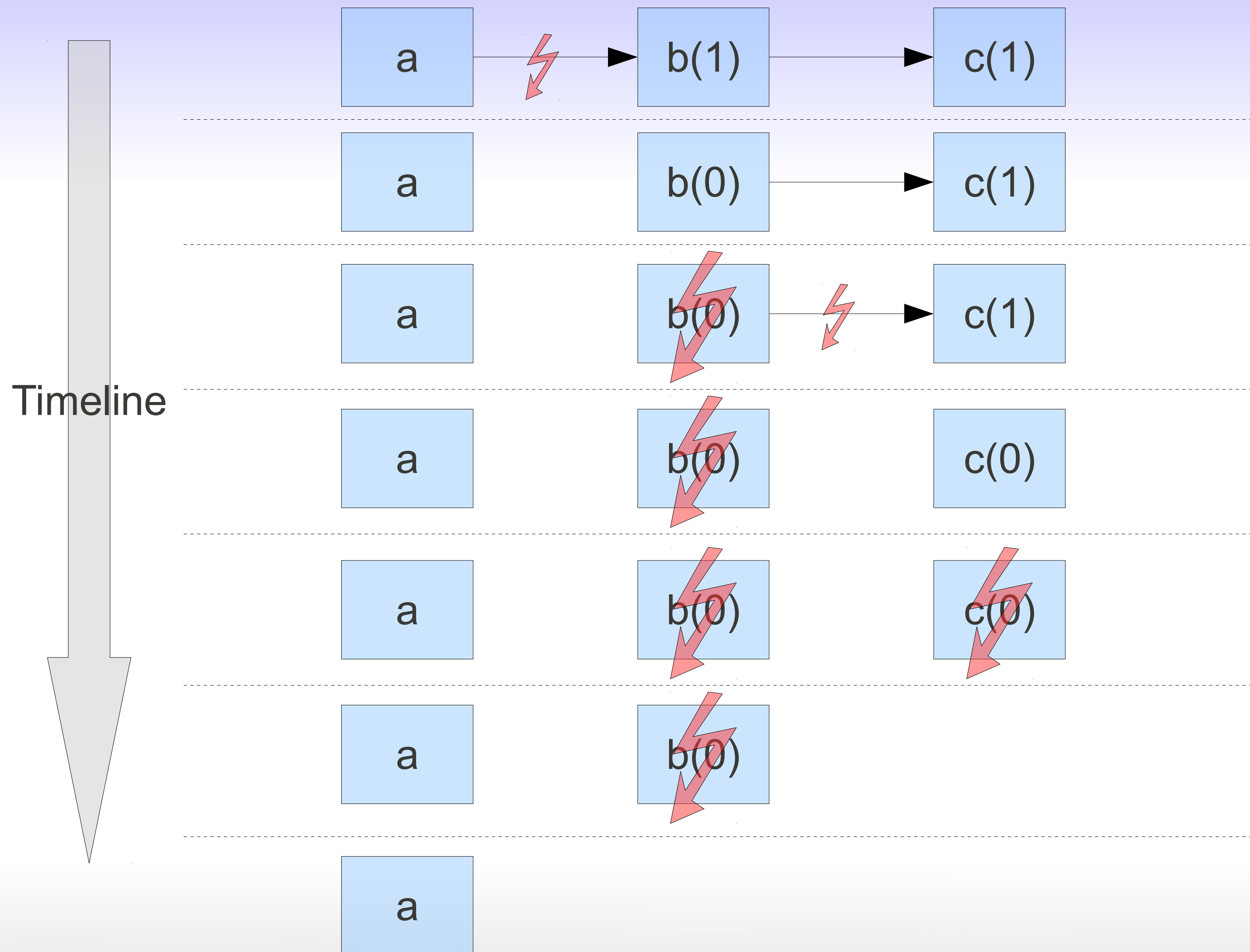
```
data = open(fn).read()
```

- No leaks in the above examples
  - File closed immediately
  - “Formally”, it's wrong

```
f = open(fn)
try:
    data = f.read()
finally:
    f.close()
```

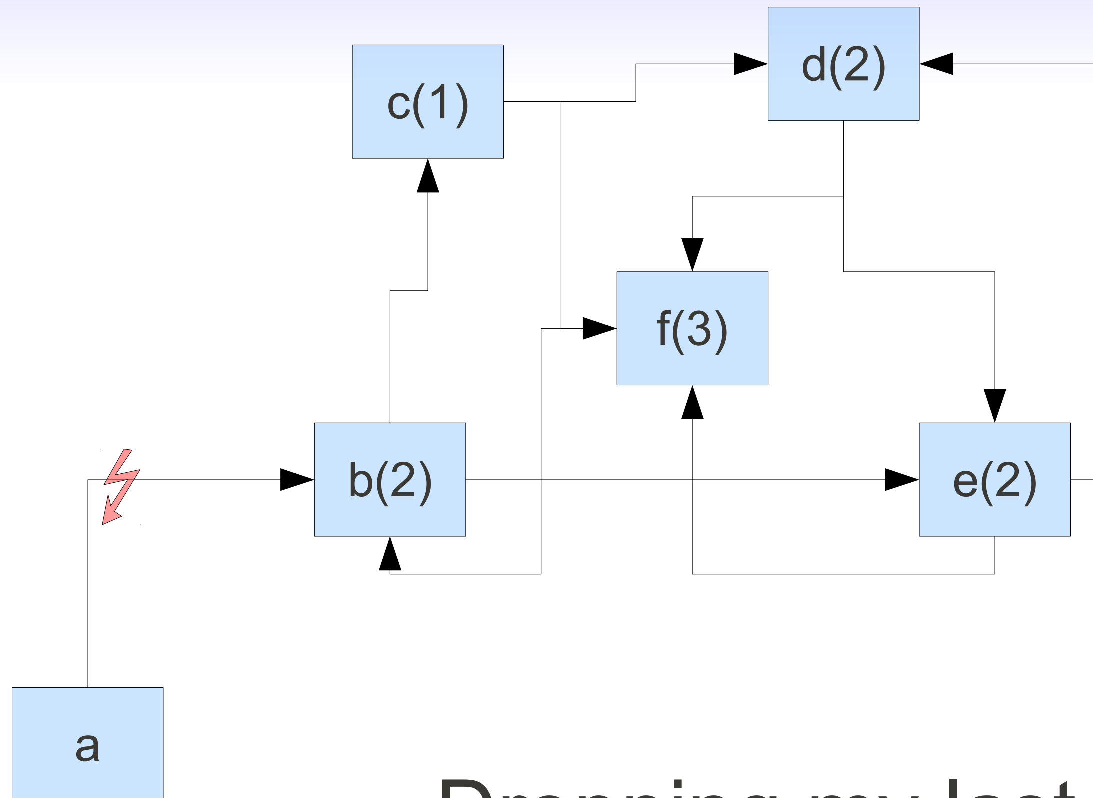
```
with open(fn) as f:
    data = f.read()
```

CPython: reference counting



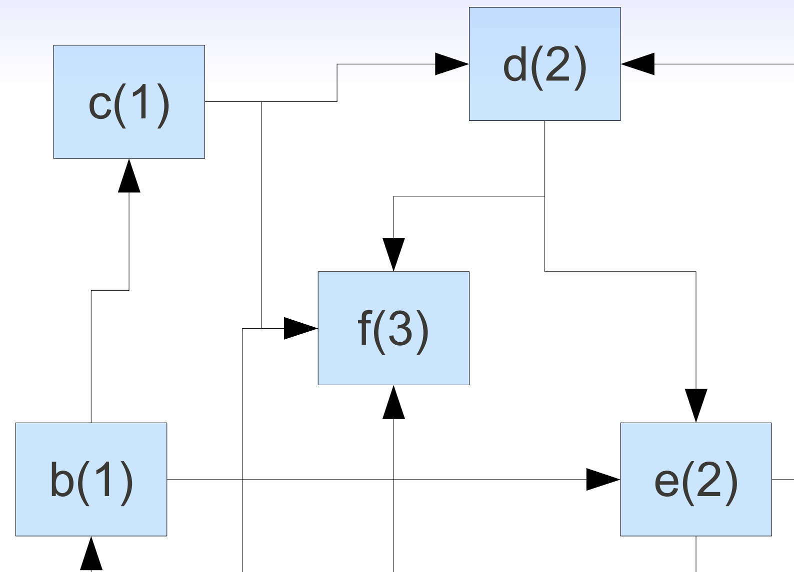
develer

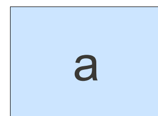
CPython: reference cycles



Dropping my last reference to a group of objects...

CPython: reference cycles



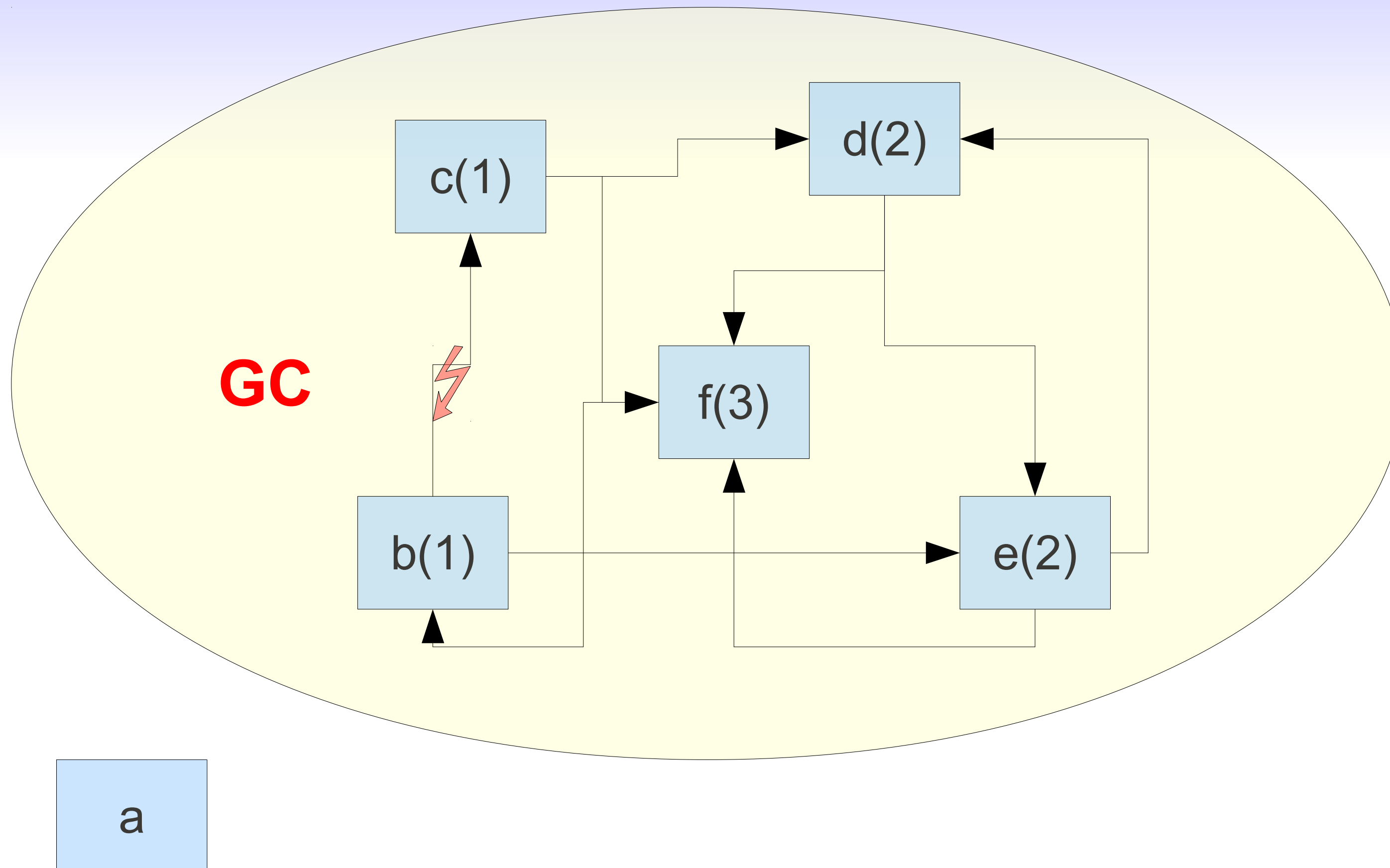


… but no counters go to zero!

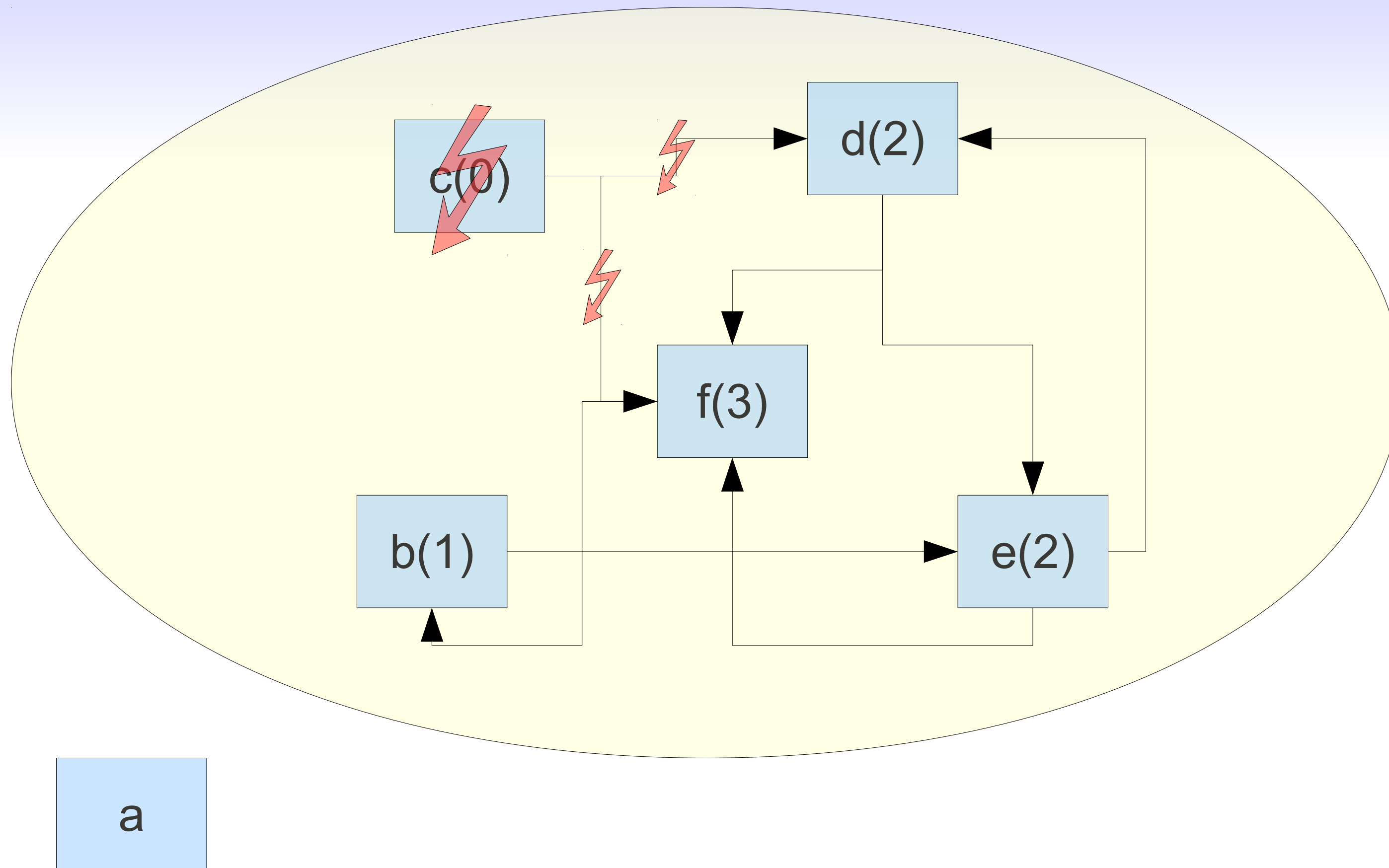
CPython: garbage collector

- Garbage collector
  - Used **only** to free cycles of objects
  - Find cycles with no outside reference
  - Break references randomically
- Automatically invoked
  - gc.collect(): force now
  - gc.disable() / gc.enable()

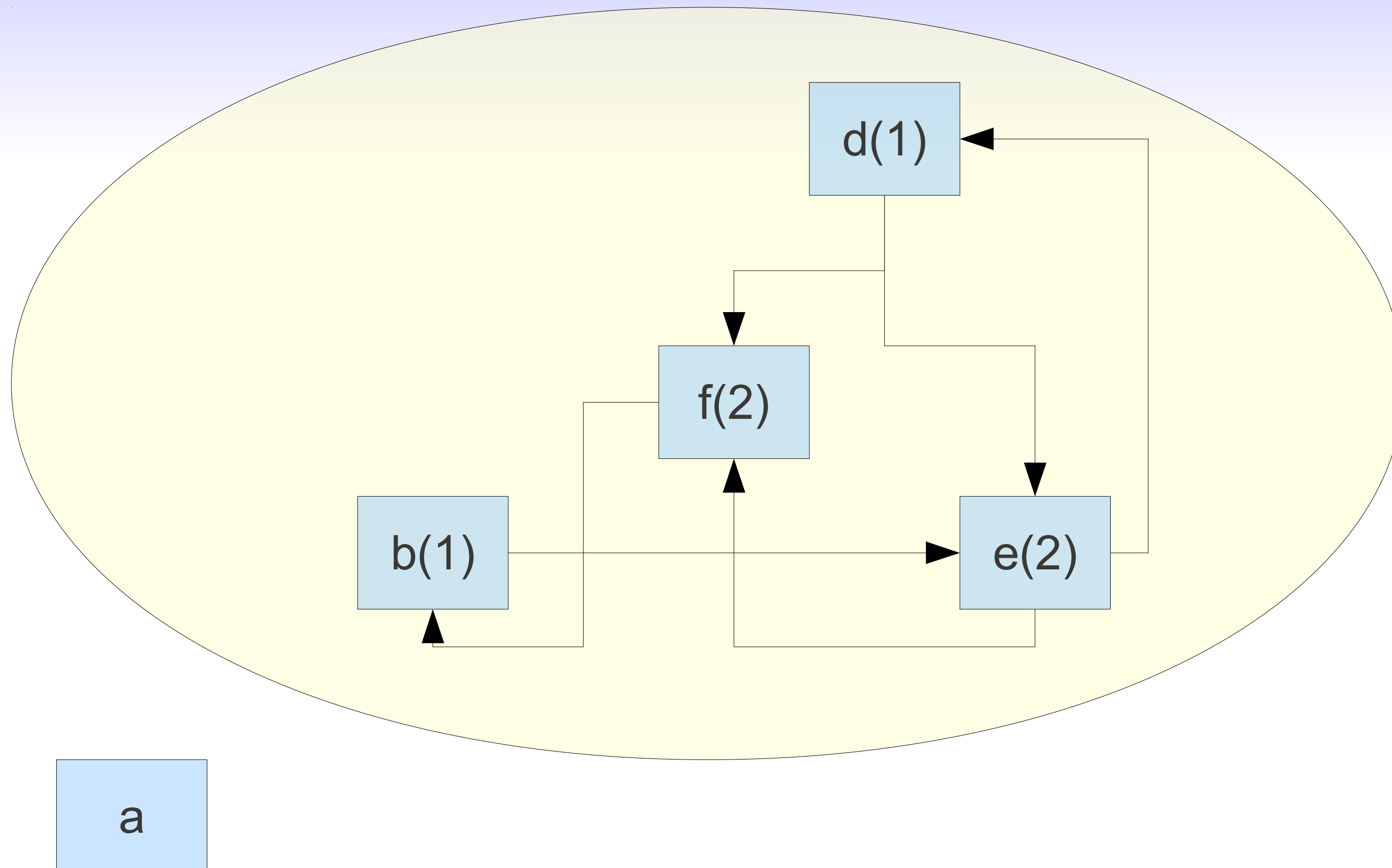
CPython: garbage collector



CPython: garbage collector

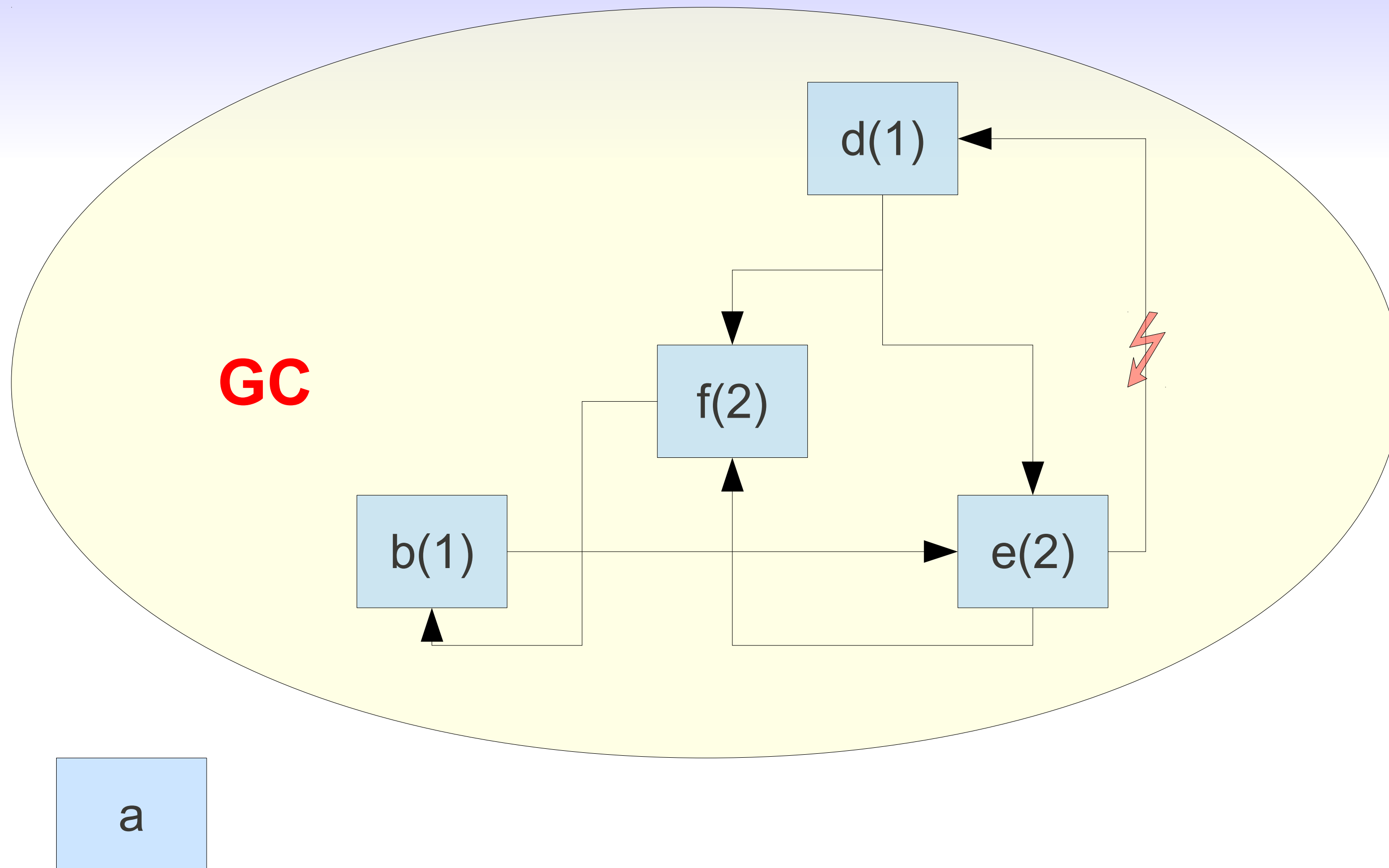


CPython: garbage collector



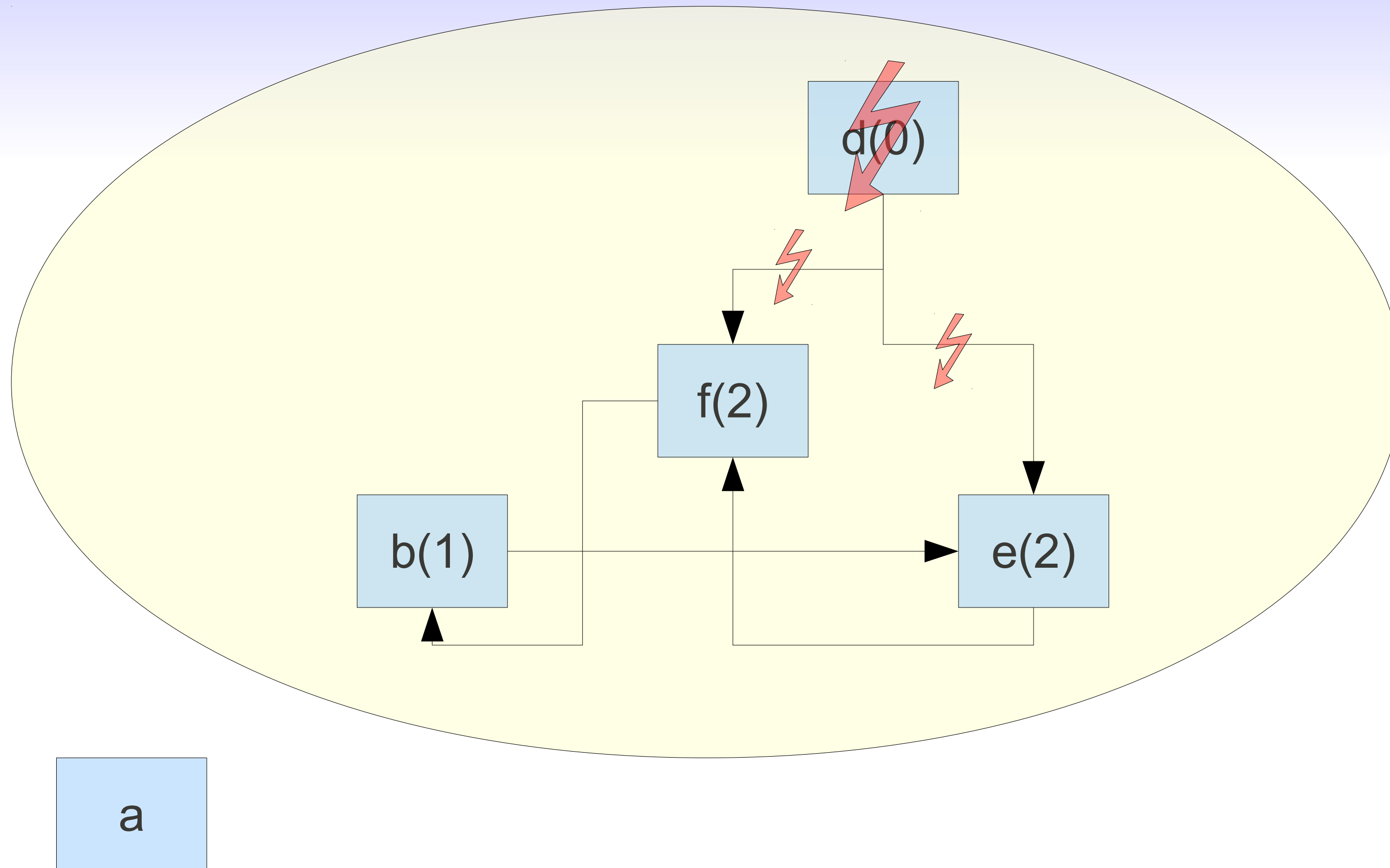
develer

CPython: garbage collector

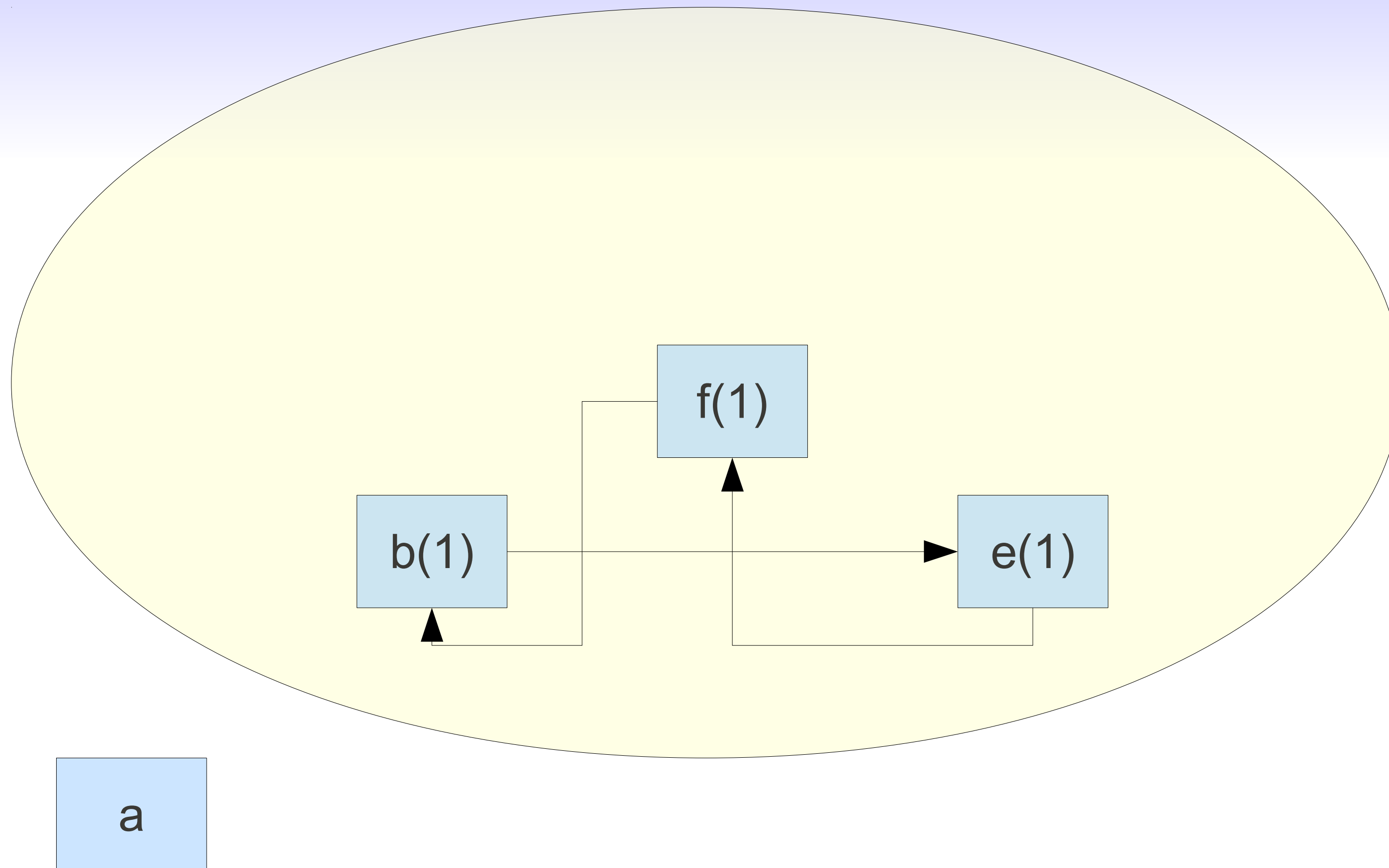


a

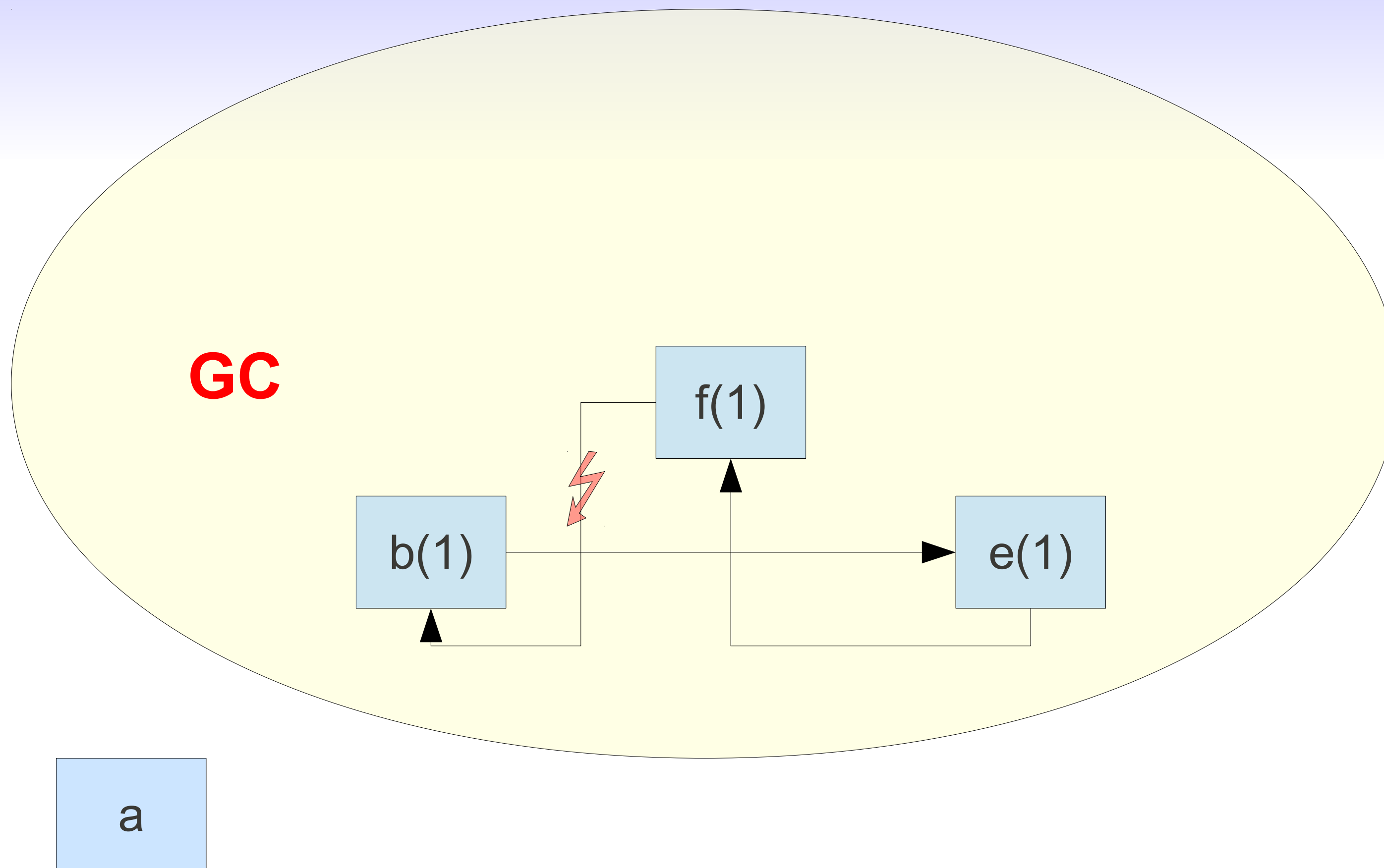
CPython: garbage collector



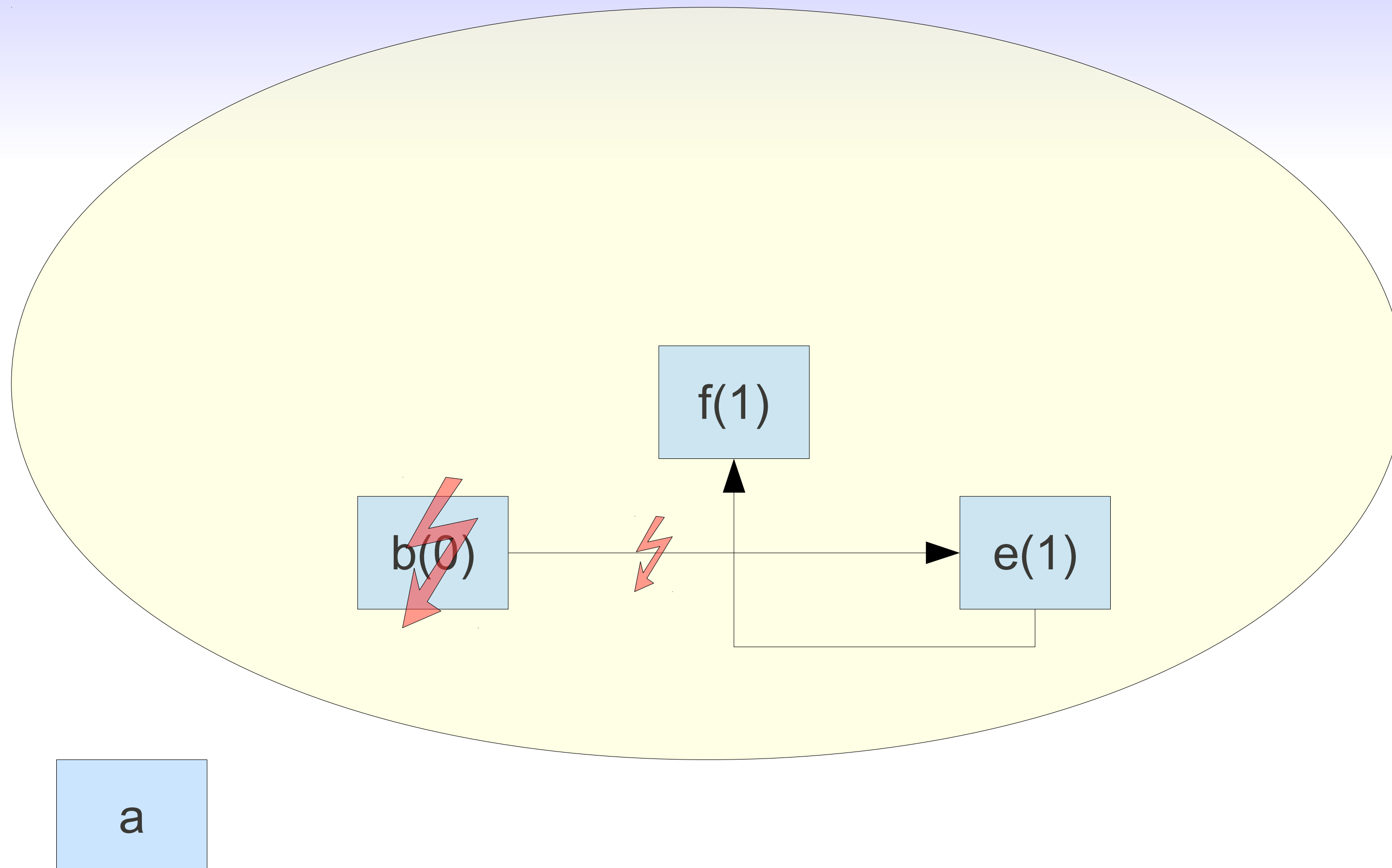
CPython: garbage collector



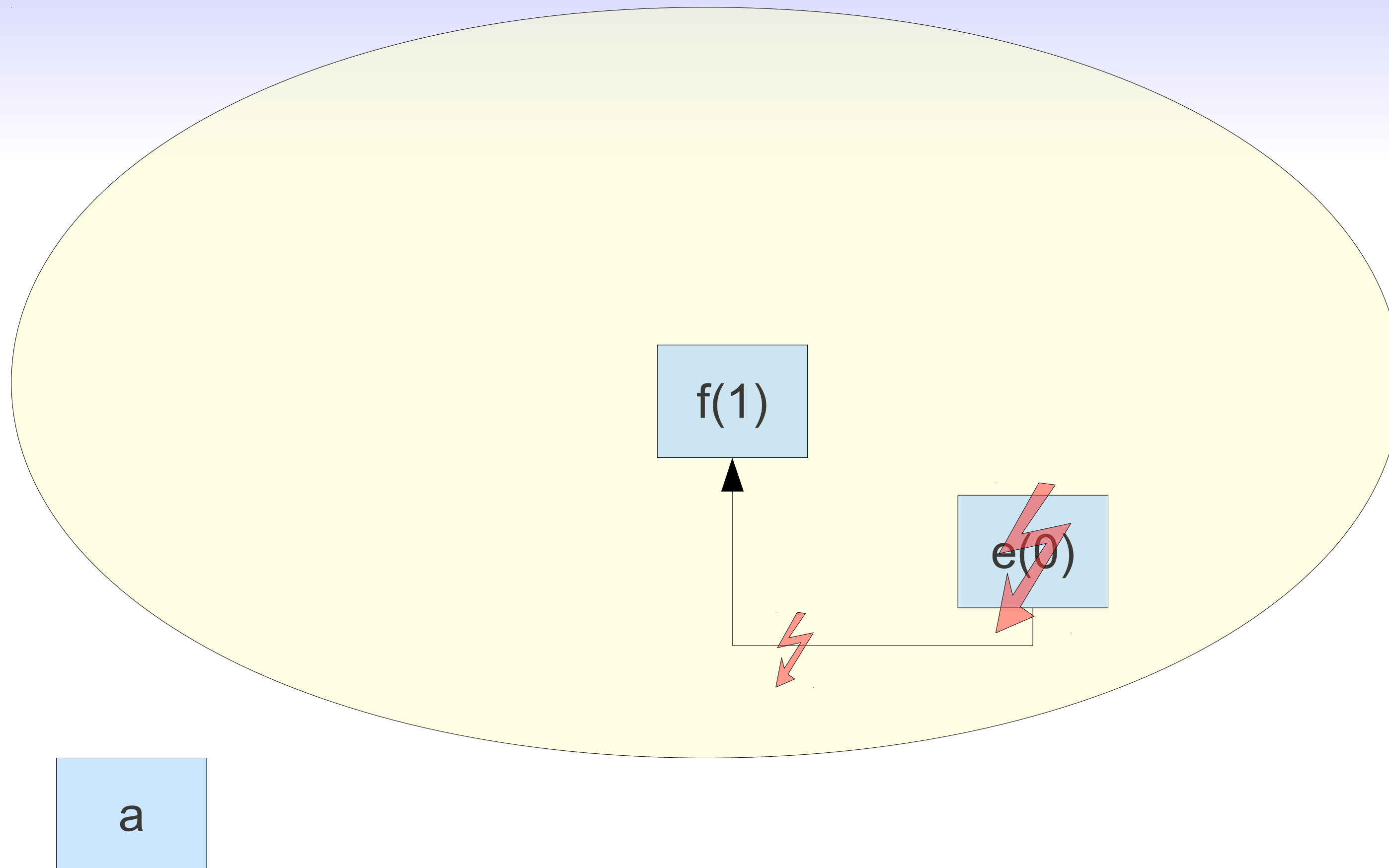
CPython: garbage collector



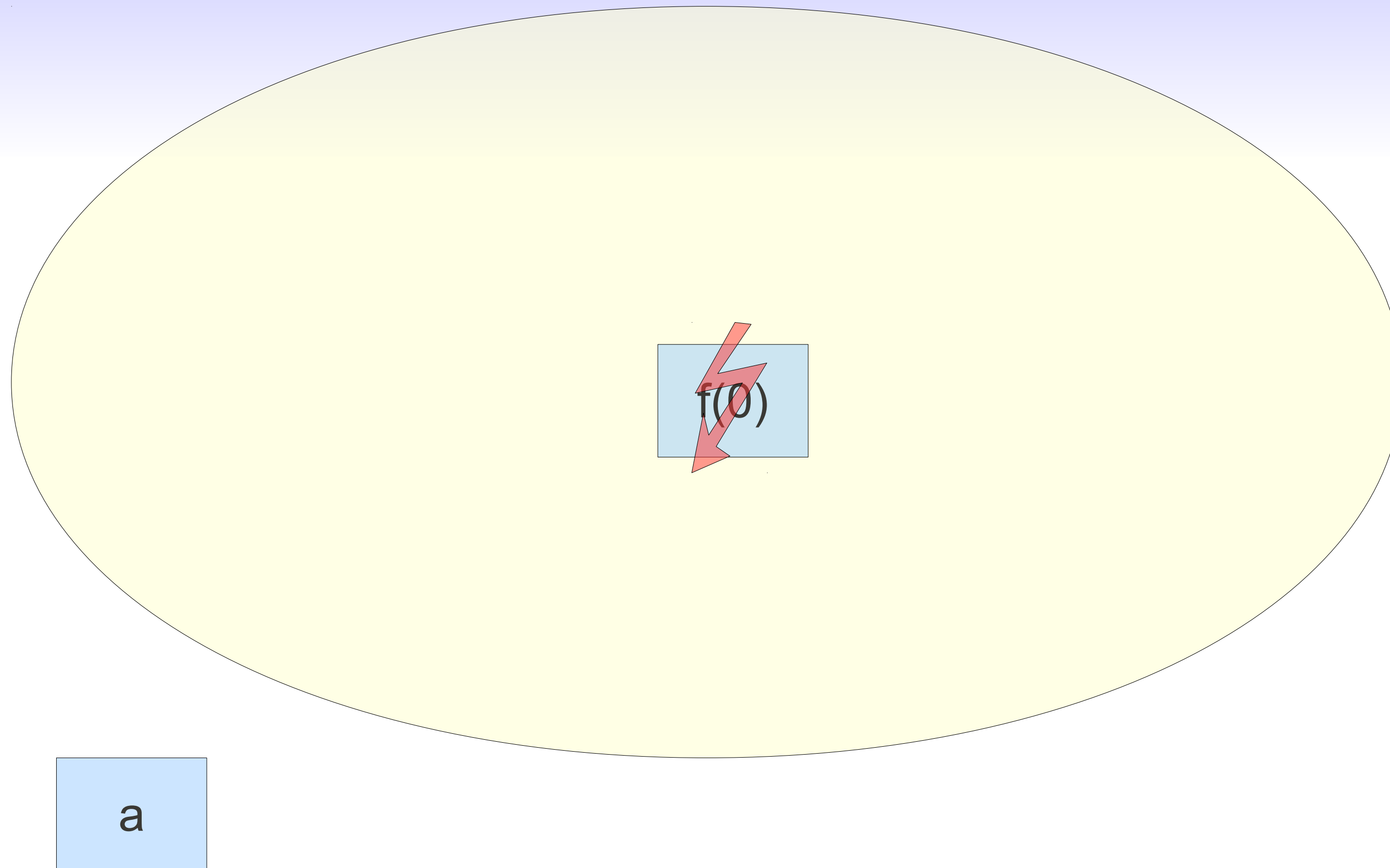
CPython: garbage collector



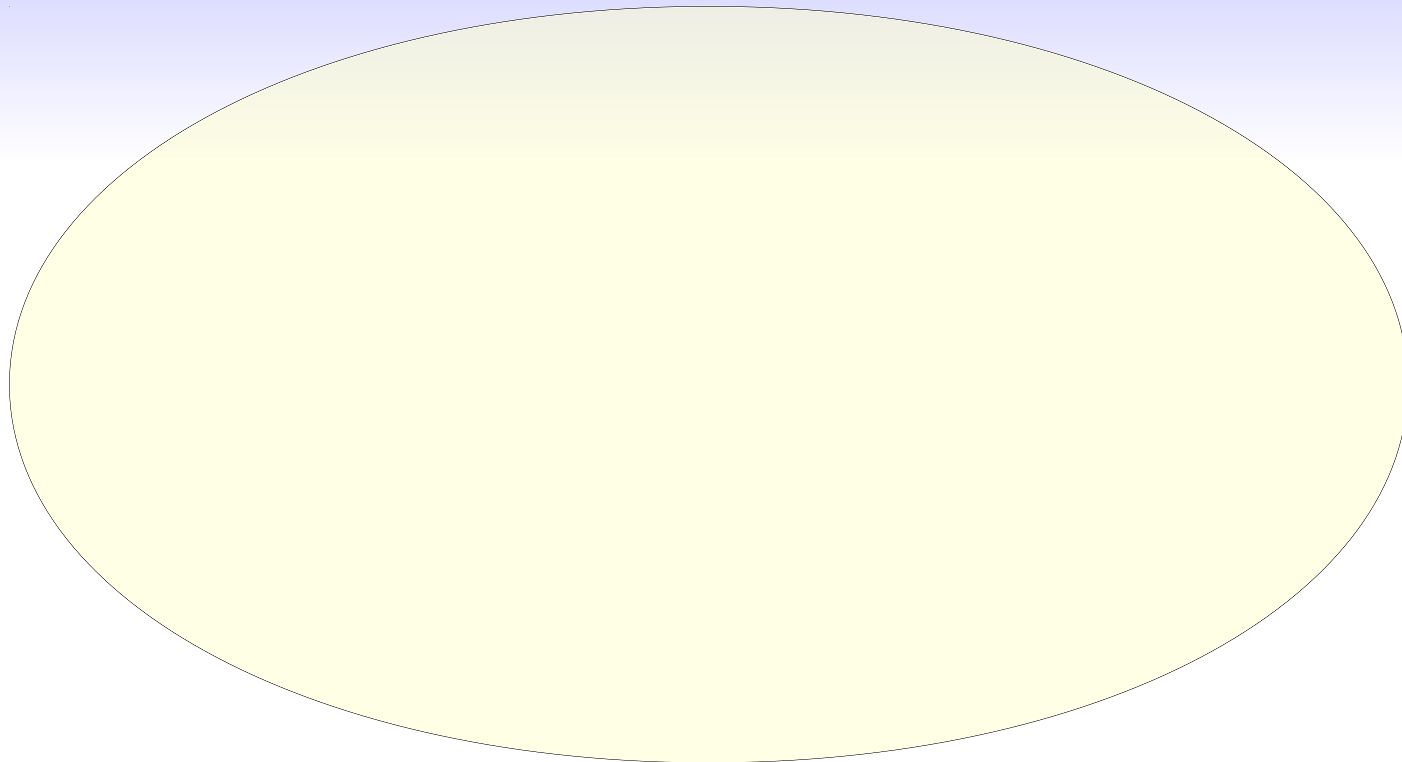
CPython: garbage collector



CPython: garbage collector



CPython: garbage collector



a

... done!

CPython: garbage collector

```
>>> a1 = A()
>>> a2 = A()
>>> a1.xxx = a2
>>> a2.xxx = a1

>>> howmany(A)
2
>>> del a1
>>> del a2
>>> howmany(A)
2

>>> gc.collect()
4
>>> howmany(A)
0
```

1) Loop creation

2) GC execution

- 4 objects collected (2 instances + 2 __dict__)

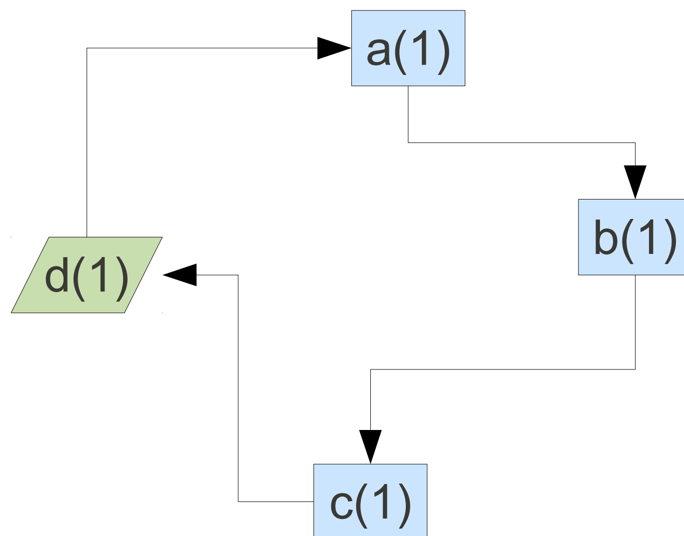
3) No more As

Destructors (__del__)

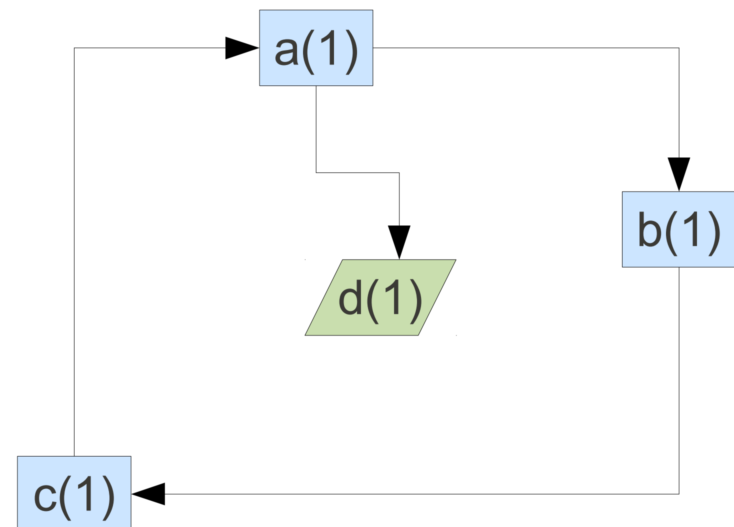
- GC **doesn't work** on objects with finalizers
  - Can't break references in random order anymore
  - True object leak!
  - Can still use __del__ on leafs

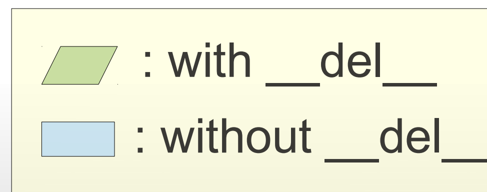
Destructors (__del__)

Leak!



OK!





Destructors (__del__)

- Use __del__ on “leafs” only
  - Eg: wrapper of C-style APIs
  - open/close, lock/release, etc.
  - “with” stmt only work for short-lived objects

```
class GLDisplayList:
    def __init__(self):
        self.id = glAllocLists(1)
    def __del__(self):
        glFreeList(self.id)
```

Destructors (__del__)

- gc.garbage
  - List of non-collectable objects
  - "gc.collect(); assert not gc.garbage"
  - Look here if in trouble...

Finalizers without __del__

- Use weakref
  - Callback when object is collected
  - Finalize in the callback
  - Pay attention to loops...

Finalizers without __del__

```
class GLDisplayList:
    def __init__(self):
        self.id = glAllocLists(1)
    def __del__(self):
        glFreeList(self.id)

assert howmany(GLDisplayList) == 0
x = GLDisplayList()
assert howmany(GLDisplayList) == 1
del x
assert howmany(GLDisplayList) == 0
```

How do we rewrite it without __del__?

Finalizers without __del__

```
class GLDisplayList:
    def __init__(self):
        self.id = glAllocLists(1)
        self._wr = weakref.ref(self,
            lambda: glFreeList(self.id))
```

- weakref to self
- __del__ → weakref callback
- But... does it work?

Finalizers without __del__

```
class GLDisplayList:
    def __init__(self):
        self.id = glAllocLists(1)
        self._wr = weakref.ref(self,
            lambda: glFreeList(self.id))

x = GLDisplayList()
del x
print howmany(GLDisplayList)   # prints 1!
gc.collect()
print howmany(GLDisplayList)   # prints 0
```

- Loop!
  - self → weakref → callback → **nested scope** → self
- QoI regression wrt __del__

Finalizers without __del__

```
class GLDisplayList:
    def __init__(self):
        self.id = glAllocLists(1)
        self._wr = weakref.ref(self,
            lambda id=self.id: glFreeList(id))

x = GLDisplayList()
del x
assert howmany(GLDisplayList) == 0
```

- Avoid the nested scope
  - Use default arguments
- Virtually perfect substitution for __del__

Finalizers without __del__

```
def finalize(obj, callback, _registry=[]):
    def wrapper(wr):
        _registry.remove(wr)
        callback()
    _registry.append(ref(obj, wrapper))
```

- finalize() utility
- Keep the weakrefs outside (cleaner)

```
class GLDisplayList:
    def __init__(self):
        self.id = glAllocLists(1)
        finalize(self,
           lambda id=self.id: glFreeList(id))
```

Finalizers without __del__

```
def debugdeath(o):
    def cb(r=repr(o)):
        print "Dead:", r
    finalize(o, cb)
```

- debugdeath() utility
- Let you know **when** an object is collected
  - No modification to object's code

Leaks in C/C++ extensions

- Leaks can happen on any Python object they handle
- Deep into C/C++:
  - Py_INCREF w/o Py_DECREF
  - CPython API is bug-prone
  - Trust binding generators more

Roadmap

- Memory leaks: definitions and basic analysis
- Memory leaks: garbage collection
- **Python code profiling**
- Debugging C/C++ extensions

Microprofiling

- timeit
  - Small to medium code snippets
  - Measure average execution time
  - Dynamic precision
  - Can be used from shell

```
python -m timeit -s [setup] [code...]

python -m timeit -s "L = [0]" \
                 "L*100"

python -m timeit -s "import core" \
                 -s "core.load('foo.dat')" \
                 -s "K = core.finditems()" \
                 "[item.compute() for item in K]"
```

Microprofiling

```
$ python -m timeit \
         -s "from PyQt4.QtGui import QMatrix"
         -s "m=QMatrix()" \
         "m.inverted()"

100000 loops, best of 3: 5.93 usec per loop
```

- Down to microsecs
- Use to compare different implementations
  - No hints on what's a bottleneck

Profiling

- Two main types
  - Deterministic: cProfile/profile
  - Statistic: hotshot
- cProfile is the hot new guy

Profiling: execution

```
def test_foo():
    [...]

import cProfile
cProfile.run('test_foo()')
```

Within your program

```
def test_foo():
    [...]

test_foo()
```

```
$ python -mcProfile foo.py
```

Outside your program

Profiling: output example

```
         389523 function calls in 2.637 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.072    0.072    2.637    2.637 <string>:1(<module>)
        1    0.848    0.848    2.564    2.564 voronoi.py:126(voronoi_geo2d)
        1    0.000    0.000    0.000    0.000 voronoi.py:51(__init__)
        1    0.013    0.013    0.018    0.018 voronoi.py:57(vertices)
        1    0.169    0.169    0.234    0.234 voronoi.py:78(connected)
    39946    0.278    0.000    0.313    0.000 voronoi.py:92(addArc)
        1    0.000    0.000    0.000    0.000 {built-in method resized}
        1    1.041    1.041    1.041    1.041 {geo2d._geo2dcpp.voronoi}
        2    0.000    0.000    0.000    0.000 {len}
   109888    0.051    0.000    0.051    0.000 {method 'add' of 'set' objects}
    39946    0.014    0.000    0.014    0.000 {method 'append' of 'list' objects}
        1    0.005    0.005    0.005    0.005 {method 'keys' of 'dict' objects}
    39947    0.017    0.000    0.017    0.000 {method 'pop' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'pop' of 'set' objects}
   159784    0.129    0.000    0.129    0.000 {round}
```

Profiling: increase readibility

- Sorting:
  - cumtime: time spent within a function, w/o children
    - python -m cProfile **-s internal**
  - tottime: time spent within a function, w/ children
    - python -m cProfile **-s cumulative**
- Try both!

Cumulative sorting

```
700778 function calls (700775 primitive calls) in 1.807 CPU seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    1.807    1.807 <string>:1(<module>)
     1    0.008    0.008    1.807    1.807 {execfile}
     1    0.202    0.202    1.799    1.799 voronoi.py:3(<module>)
100000    0.519    0.000    1.582    0.000 voronoi.py:311(RP)
200000    0.289    0.000    1.063    0.000 random.py:211(randint)
200000    0.686    0.000    0.774    0.000 random.py:147(randrange)
200000    0.088    0.000    0.088    0.000 {method 'random' of
                                            '_random.Random' objects}
     1    0.001    0.001    0.008    0.008 __init__.py:4(<module>)
     1    0.004    0.004    0.004    0.004 offset.py:3(<module>)
     1    0.004    0.004    0.004    0.004 {range}
     1    0.002    0.002    0.002    0.002 heapq.py:31(<module>)
```

Internal sorting

```
700778 function calls (700775 primitive calls) in 1.822 CPU seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
200000    0.683    0.000    0.767    0.000 random.py:147(randrange)
100000    0.509    0.000    1.577    0.000 voronoi.py:311(RP)
200000    0.301    0.000    1.068    0.000 random.py:211(randint)
     1    0.219    0.219    1.812    1.812 voronoi.py:3(<module>)
200000    0.083    0.000    0.083    0.000 {method 'random' of
                                              '_random.Random' objects}
     1    0.010    0.010    1.822    1.822 {execfile}
     1    0.006    0.006    0.006    0.006 {range}
     1    0.004    0.004    0.004    0.004 offset.py:3(<module>)
     1    0.001    0.001    0.008    0.008 __init__.py:4(<module>)
     1    0.001    0.001    0.001    0.001 random.py:39(<module>)
     1    0.001    0.001    0.001    0.001 heapq.py:31(<module>)
```

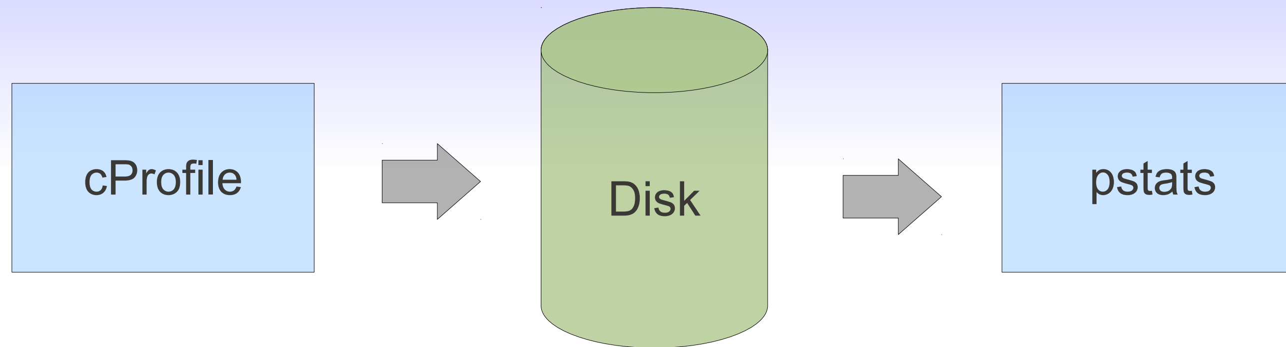
Diving through the numbers...

- One profile session is **never enough**
- Generate lots of data
  - Try both cProfile and hotshots
  - Try different data inputs
  - Pay attention to overhead and absolute numbers
    - cProfile increases Python execution time wrt C/C++ extensions.
    - Use time.time() to get back to reality

Diving through the numbers...

- If you don't see anything glaring
  - Split into multiple functions (warn: overhead)
  - Use C-level profiling (oprofile)
- Don't lose focus
  - Attack bottlenecks and solve them
  - Micro-optimizations don't pay back

"pstats" module



- Save data to file
  - -o [output-file] / .run('foo', 'output-file')
- Post-profiling analysys
  - Test different sorting
  - Filter out useless data

“Quick” things to try

- Psyco
  - Dynamic JIT of Python code
  - Works on basic data types
  - Easy to try (10 mins), might work
- Cython
  - Decorate Python code with types, then compile
  - Can take a few hours to do an attempt
  - Worth a look

Roadmap

- Memory leaks: definitions and basic analysis
- Memory leaks: garbage collection
- Python code profiling
- **Debugging C/C++ extensions**

C/C++ extensions debugging

- How to debug C/C++ extensions
  - Binding code (Python C API)
  - Underlying C/C++ code
- Goal:
  - Post-mortem debugging (segfault)
  - Step-by-step debugging

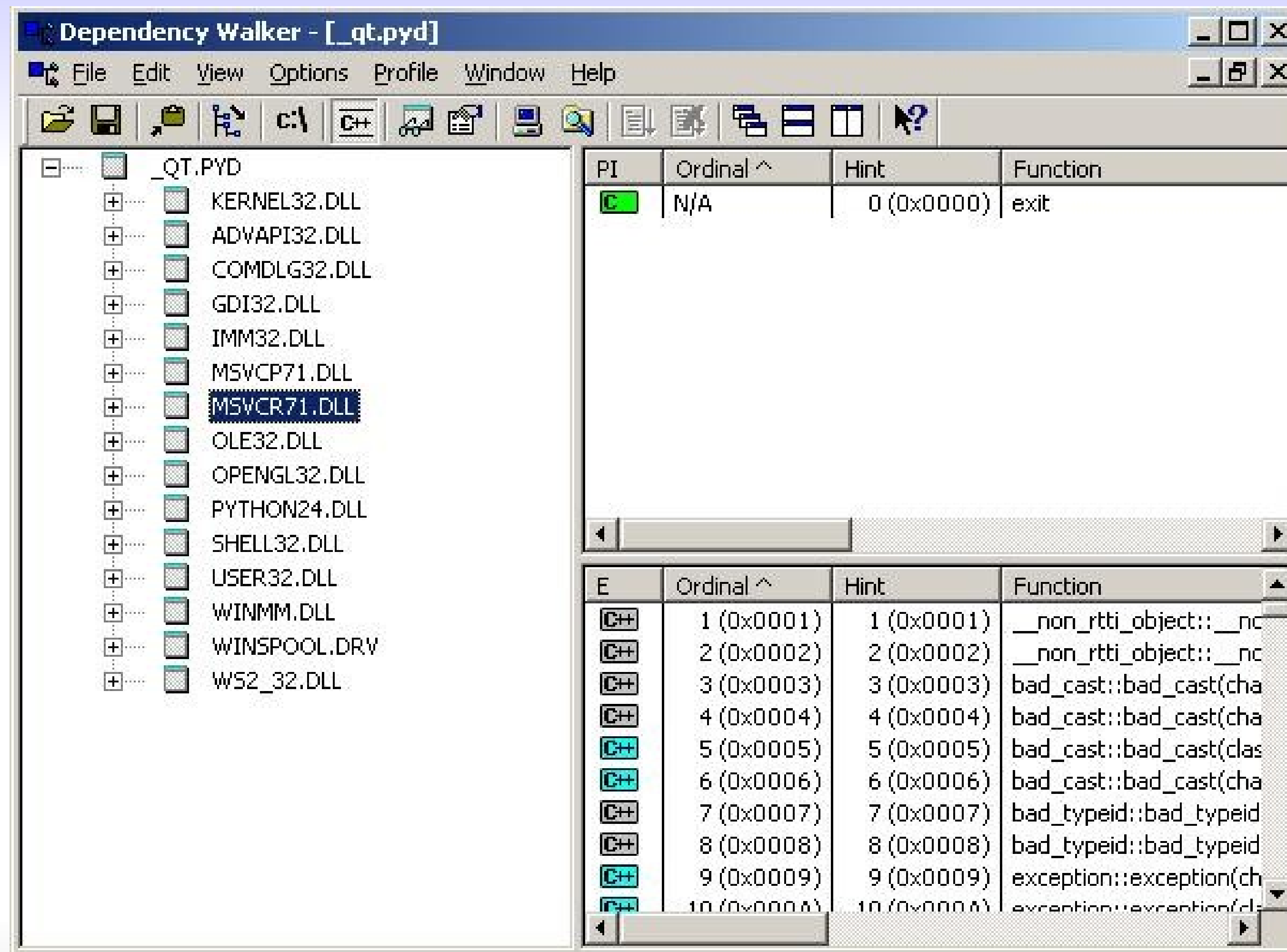
Windows: ABI troubles

- ABI tied to CRT (msvcr*)
  - Changes between debug and release builds
  - Changes among Visual Studio versions
- CRT = C runtime dynamic libraries:
  - Es: msvcr**71d**.dll
    - **71** = Visual Studio 7.1 (aka .NET 2003)
    - **d** = debug

Windows: ABI troubles

- ABI check:
  - depends.exe (http://www.dependencywalker.com/) (GUI)
  - PyInstaller's bindepend.py (cmdline)
- Check which msvcr* is used
  - **Must be only one!**
  - The same of pythonXX.dll

DependecyWalker



Runtime versions

Python 2.3	VC 6.0	MSVCRT.DLL
Python 2.4+	VC 7.1 (2003)	MSVCR71.DLL
Python 2.6+	VC 9.0 (2008)	MSVCR90.DLL

Also: try special MinGW build for Python developers (maintained by Develer)

- http://www.develer.com/oss/GccWinBinaries
- Integration with distutils

Getting debug symbols

- Must keep ABI stable
  - Same CRT as release
  - Eg: msvcr71.dll, **not msvcr71d.dll**
- Visual Studio IDE
  - Ususally, “debug” → debug CRT
  - Runtime to use: “Multithreaded release”
  - Cmdline options: “/MD /Zi” (not /MDd)

Getting debug symbols

- Don't use “–debug” in distutils
  - That's “Py_DEBUG” mode + debug CRT
  - Two ABI changes...
- Hand-edit setup.py

```
setup(
    name = “FooBar”,
    ext_modules = [
        Extension(“foobar”,
            [...]

            extra_compile_args = ["/Od", "/Zi"],
            extra_link_args = [“/DEBUG”],
        ),
    ],
)
```

Getting debug symbols

- Recompile pythonXX.dll too
  - Change Visual Studio “Release” target
  - Manually add options (like distutils)
  - Just copy the new DLL in the project app
- Since Python 3.2, pdbs can be downloaded

Attach the debugger

- Segfault → press “debug” to hop into Visual Studio
  - Full stack-trace with debug symbols
- Auto-generated project
  - Change the startup command
    - “python c:\src\foo\start.py”
    - Current directory: “c:\src\foo”
  - Restart under debug with F5
  - Breakpoint, watch, etc.

Linux: much better!

- Less troubles:
  - Just one ABI (no CRT mess)
  - Most distros ship debug symbols
- GDB can be easily attached
- Still... you need to avoid "Py_DEBUG" mode in distutils:

```
extra_compile_args = ["-O0", "-g3"],
extra_link_args = ["-g"],
```

faulthandler

- Get a stacktrace on segfault

```
>>> import faulthandler
>>> faulthander.enable()

>>> faulthandler._sigsegv()
Fatal Python error: Segmentation fault

Traceback (most recent call first):
  File "<stdin>", line 1 in <module>
Segmentation fault
```

faulthandler

- Get a stacktrace on signal (UNIX)

```
>>> faulthander.register(signal.SIGUSR1,
all_threads=True)
```

- Get a stacktrace in a while

```
>>> faulthander.dump_tracebacks_later(10.0)
```

- Python 3.3 (or PyPI today, 2.x as well!)

That's all, folks!

Questions?

Giovanni Bajo
<rasky@develer.com>
@giovannibajo
http://giovanni.bajo.it

CC BY NC SA