

Tecniche di debugging e di profiling



By Giovanni Bajo <rasky@develer.com>

Roadmap

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

Una nota

- Si parlerà quasi esclusivamente di CPython
 - Dettagli molto diversi tra implementazioni (Jython, IronPython)
 - Memory leak: l'implementazione dei garbage collector è totalmente diversa
 - Estensioni: package esterni Java / .NET hanno problematiche diverse

Roadmap

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

Leak di memoria e di risorse

- Tecnicamente diversi (C/C++)
 - Memory leak: malloc senza free
 - Resource leak: non ho chiuso un file
- Coincidenti in Python (spesso)
 - Memoria allocata tramite creazione di oggetti Python
 - Distruttori usati per chiudere le risorse (es: `file.__del__` → `file.close`)
- Un solo tipo di leak: “leak di oggetti”

Object leak

- Python rilascia automaticamente ogni oggetto che “non serve più”
 - Nessuno lo riferenzia
- Object leak: un oggetto è “logicamente” morto, ma non viene allocato
 - Riferimento “dimenticato” o “nascosto”
 - Bug in estensioni C/C++
 - Implementazione Python “incompleta”

Contare gli oggetti allocati

- `gc.get_objects()`
 - Lista di tutti gli oggetti esistenti
 - Utile per trovare se ci sono oggetti di un certo tipo in vita
- Controllo “prima & dopo”
 - Se il numero cresce in continuazione, ci sono object leak

Contare gli oggetti allocati

```
>>> 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
```


Riferimenti “dimenticati”

- `sys.getrefcount(obj)` [... -1!]
 - Prima di “distruggere” un oggetto, controllare il suo refcount
 - Chiamare `gc.collect()` per sicurezza, prima
 - Se > 2 , c'è qualcun altro che lo punta
- Da usare anche in test automatizzati
 - Controllo implementazione di oggetto senza refleak
- Come trovo il riferimento “dimenticato”?

Riferimenti “dimenticati”

- `gc.get_referrers(*objs)`
 - Elenco oggetti con riferimento **diretti** agli oggetti specificati
 - Un livello potrebbe non essere sufficiente
 - `__dict__` di un'istanza
 - containers
 - Analizzare ricorsivamente (con debugger e/o con `print`)

Esempio: analisi riferimenti

```
>>> 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)>}]
```

Che cos'è questo dizionario?

Esempio: analisi riferimenti

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

- Il riferimento “a portata di mano” è nel contesto
 - Contesto: `sys._getframe().f_locals` (dizionario)
 - Maneggiare con cura...

Esempio: analisi riferimenti

```
>>> 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

- Uso di reference counting
- Distruzione **immediata** quando il contatore va a zero
 - Es. esplicito: “del a.ref”
 - Es. implicito: uscita da una funzione
 - Garantito da CPython (in eterno)
 - Non è Python “ufficialmente”
- Sfruttabile per scrivere codice più compatto

CPython: reference counting

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

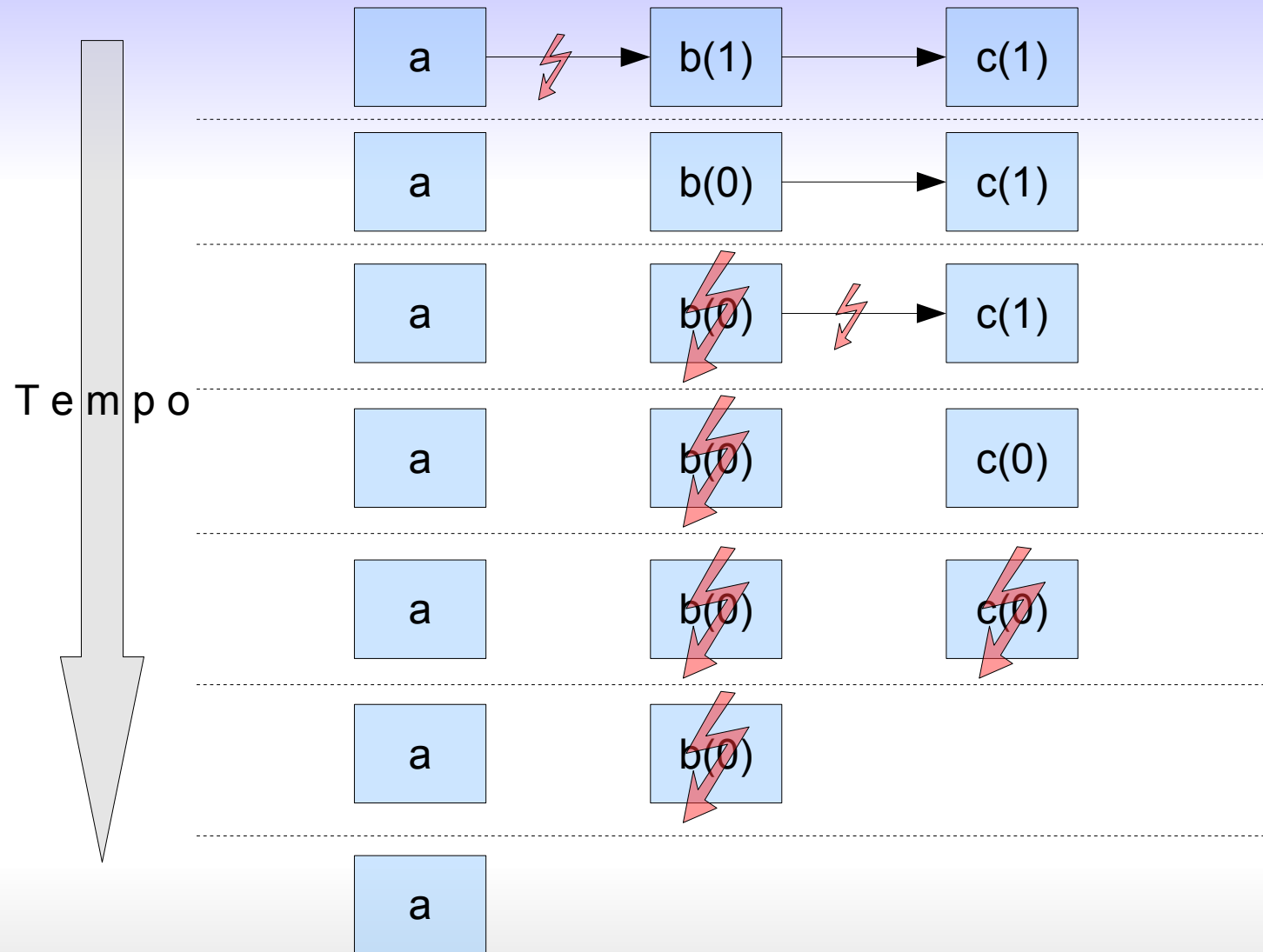
```
data = open(fn).read()
```

- Nessun leak in CPython
 - File chiuso al momento dell'uscita della funzione
 - Non “un po' dopo”
- Object (resource) leak in IronPython & friends:

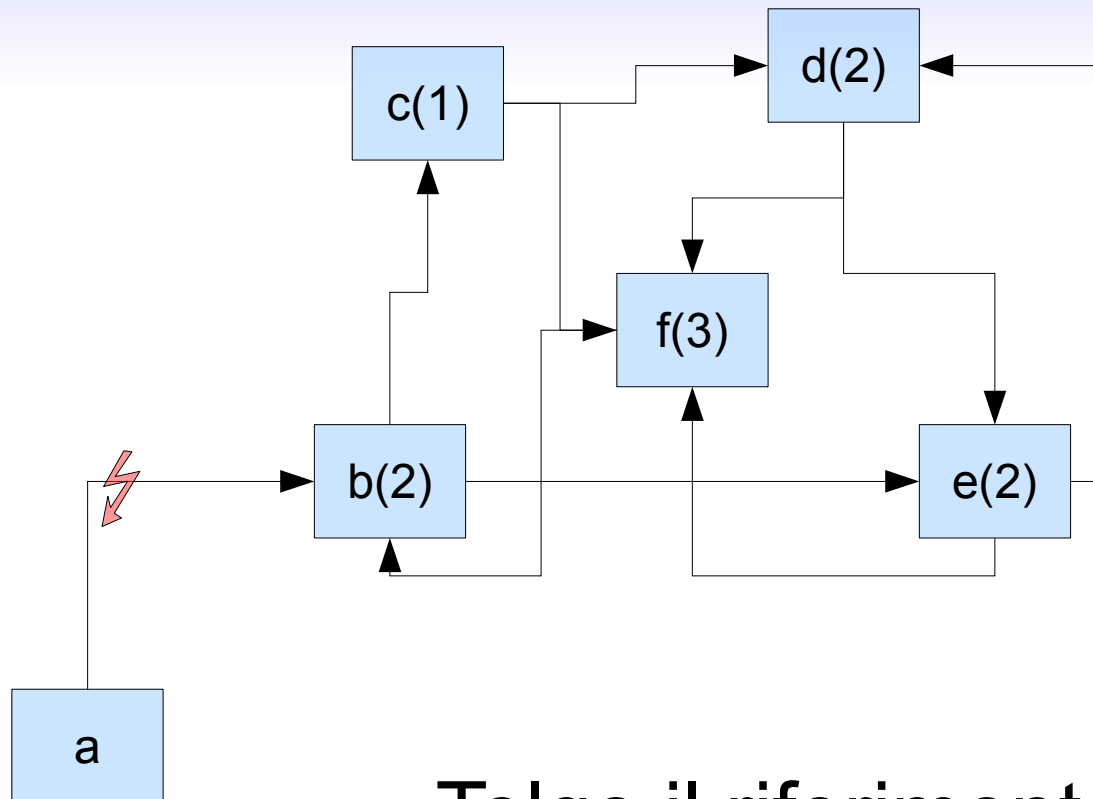
```
f = open(fn)  
try:  
    data = f.read()  
finally:  
    f.close()
```

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


CPython: reference counting

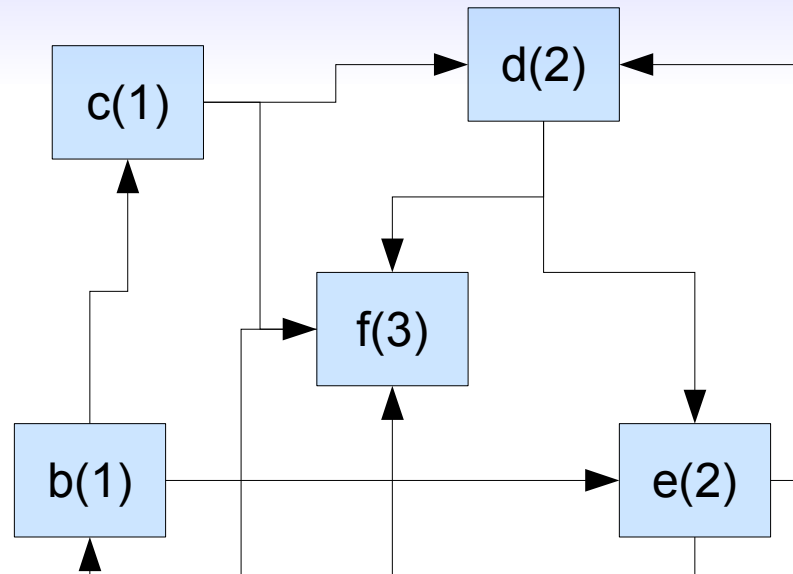


CPython: creazione di cicli



Tolgo il riferimento ad un gruppo di oggetti...

CPython: creazione di cicli



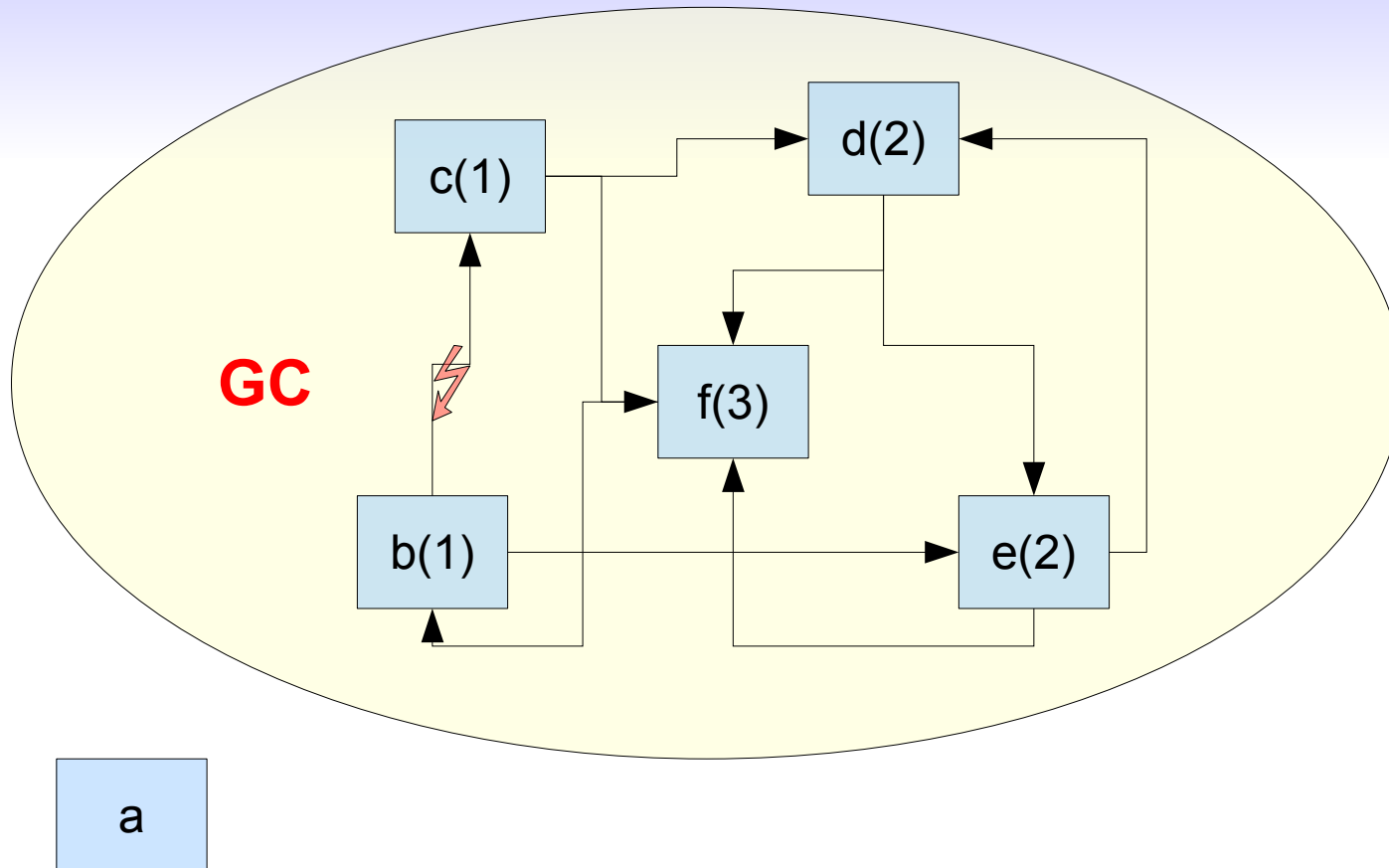
a

... ma nessun contatore va
a zero!

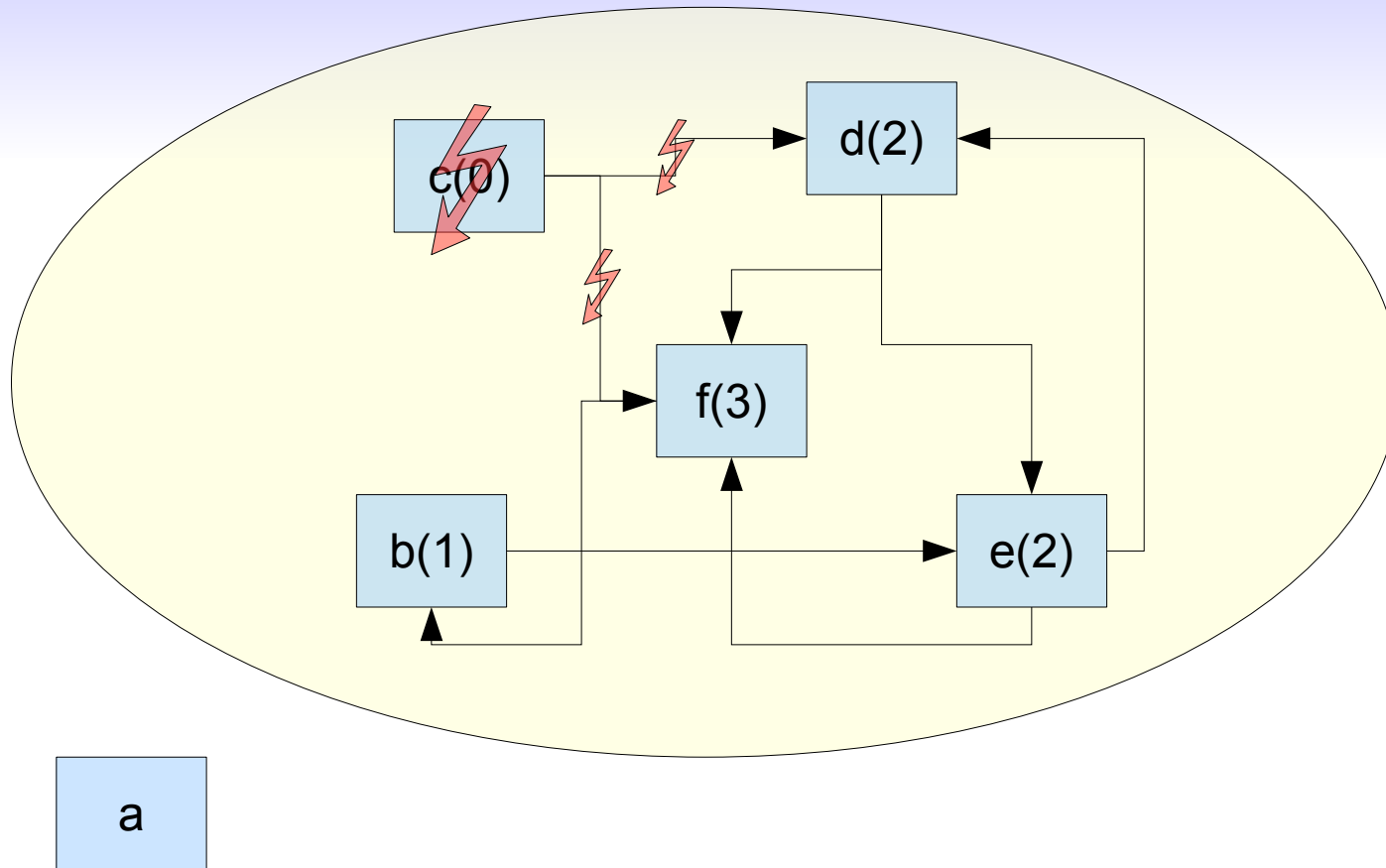
CPython: garbage collector

- Garbage collector
 - Usato **solo** per liberare gli oggetti in cicli
 - Trova cicli di oggetti che si auto-referenziano senza alcun puntatore da “fuori”
 - Distrugge riferimenti a caso, finché i loop non scompaiono
- Invocato automaticamente “ogni tanto”
 - `gc.collect()`: immediato
 - `gc.disable()` / `gc.enable()`

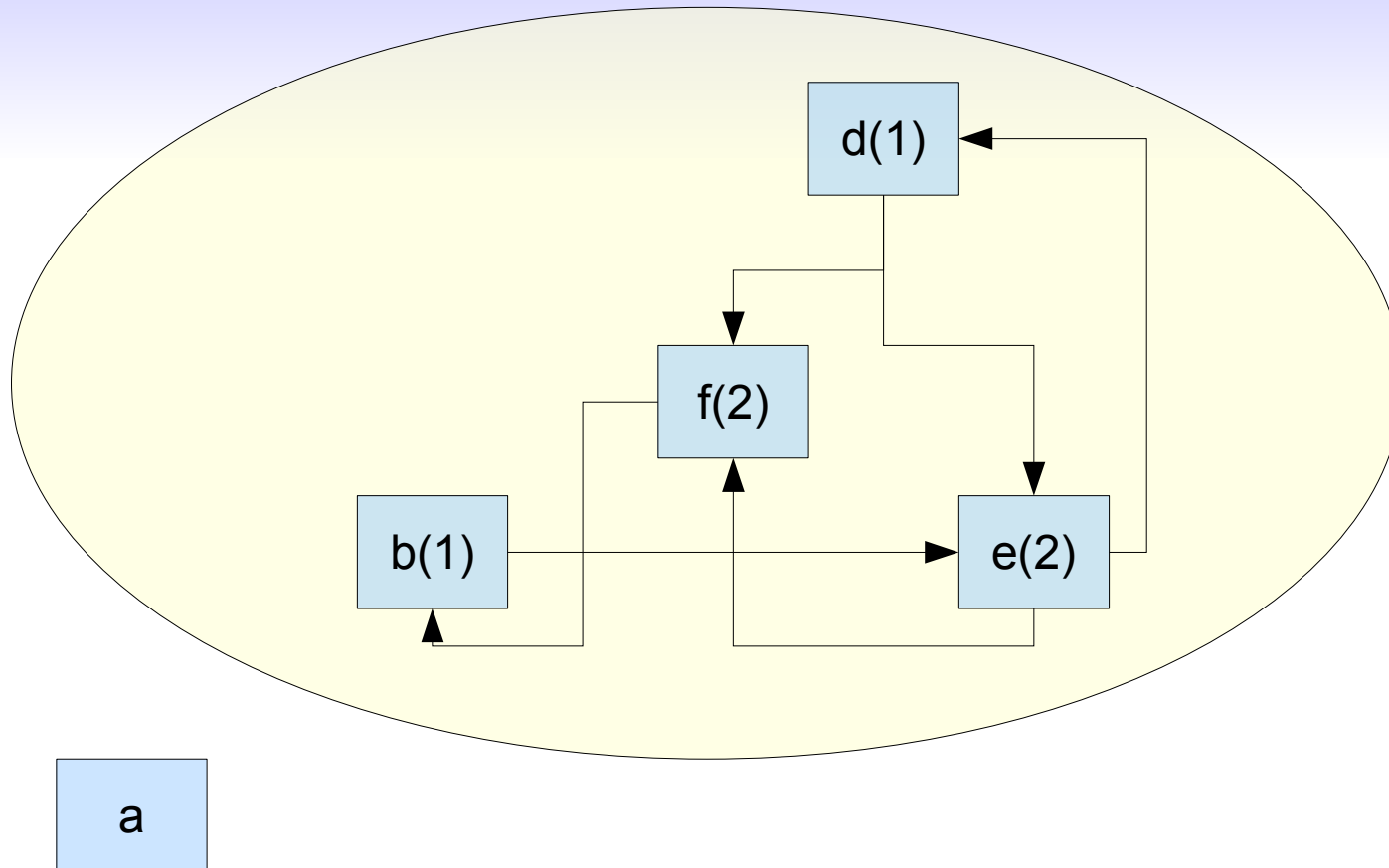
CPython: garbage collector



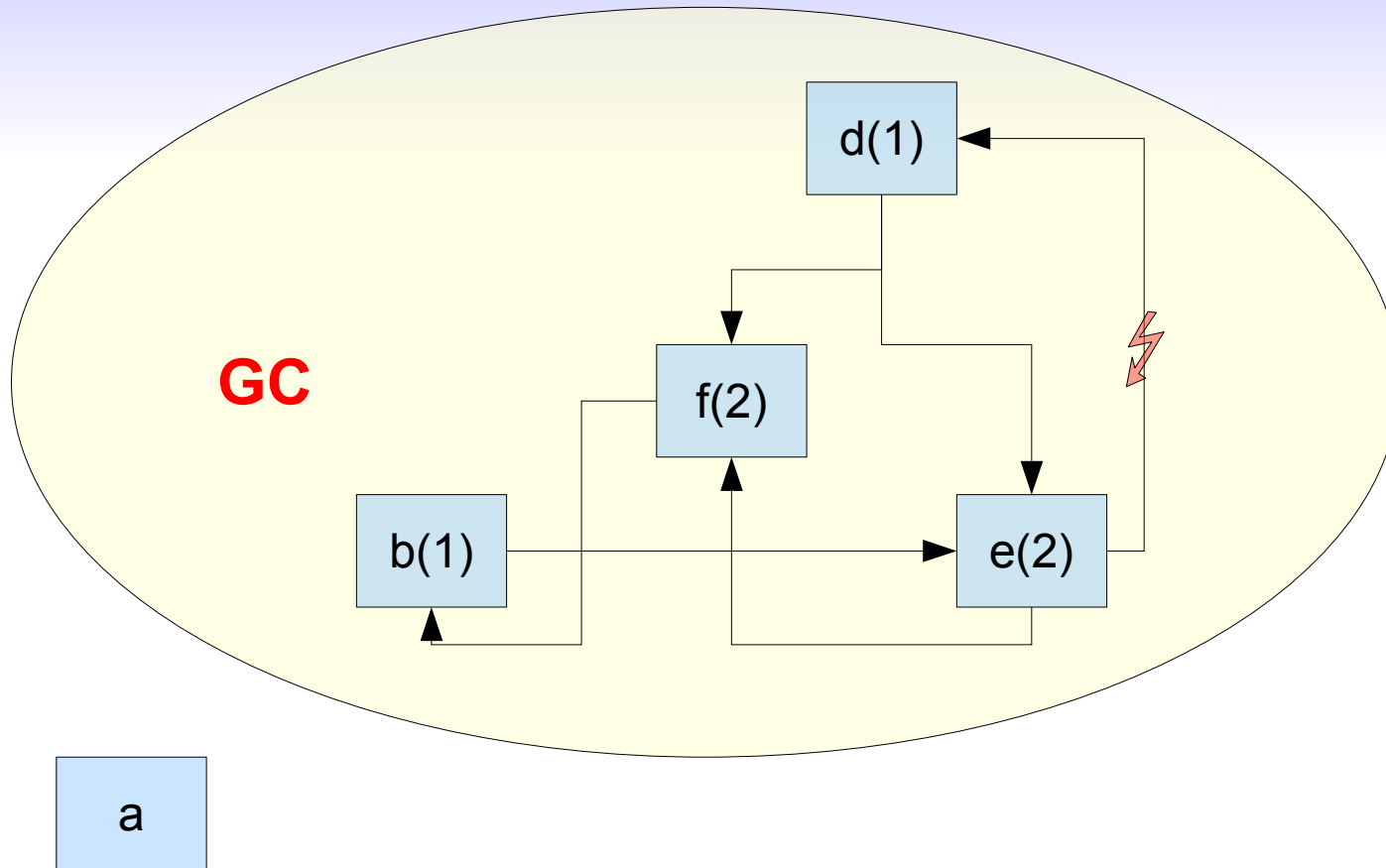
CPython: garbage collector



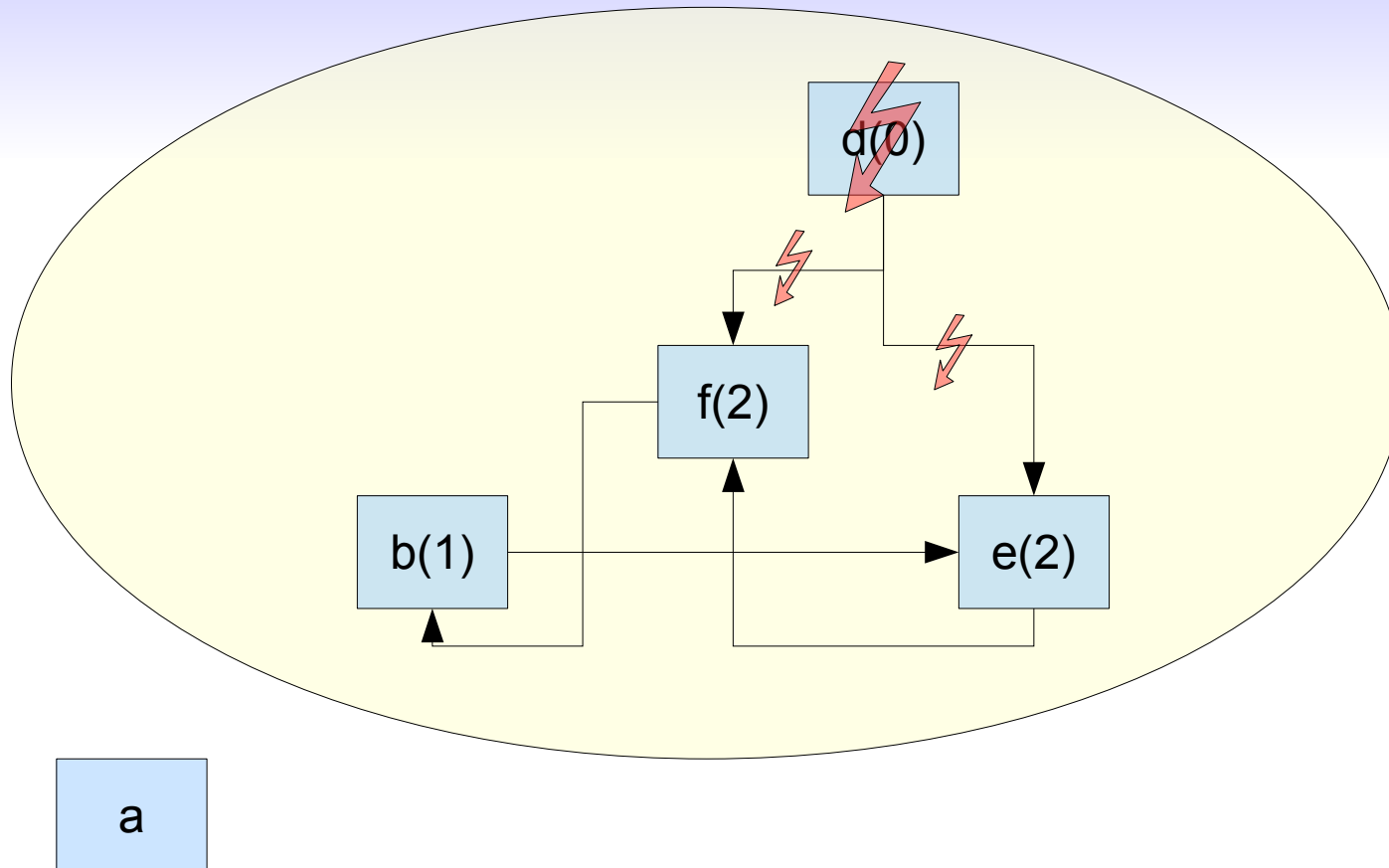
CPython: garbage collector



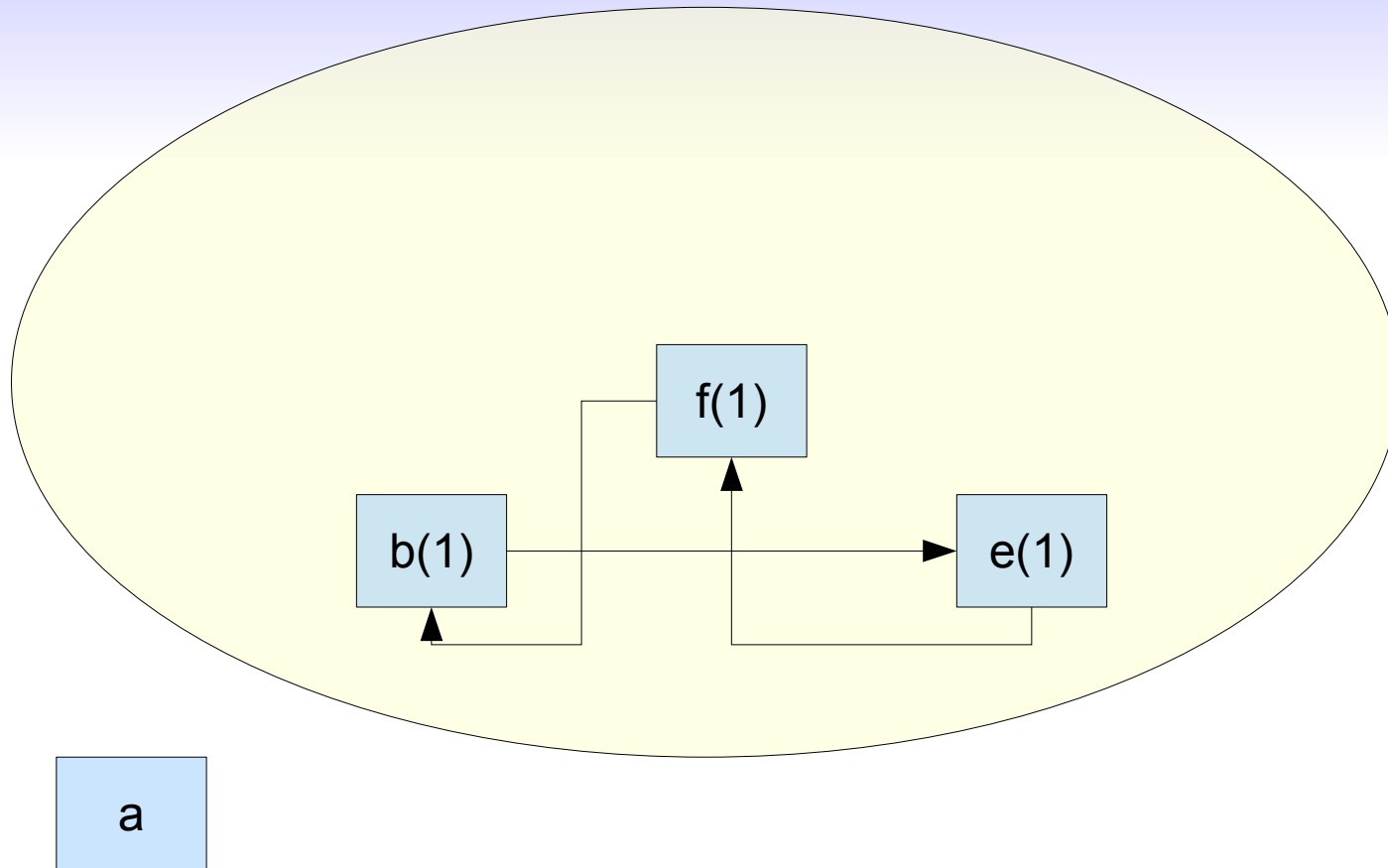
CPython: garbage collector



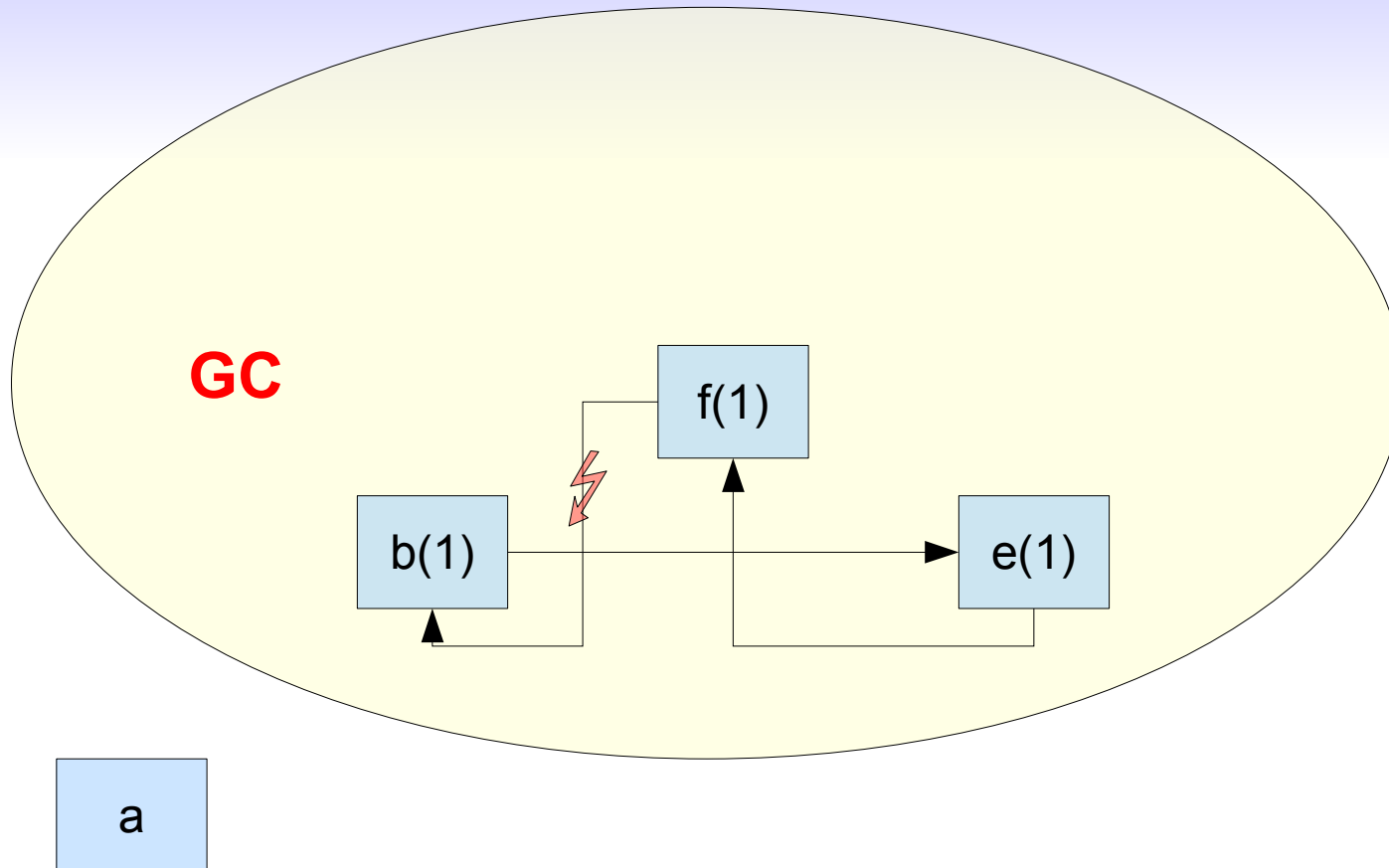
CPython: garbage collector



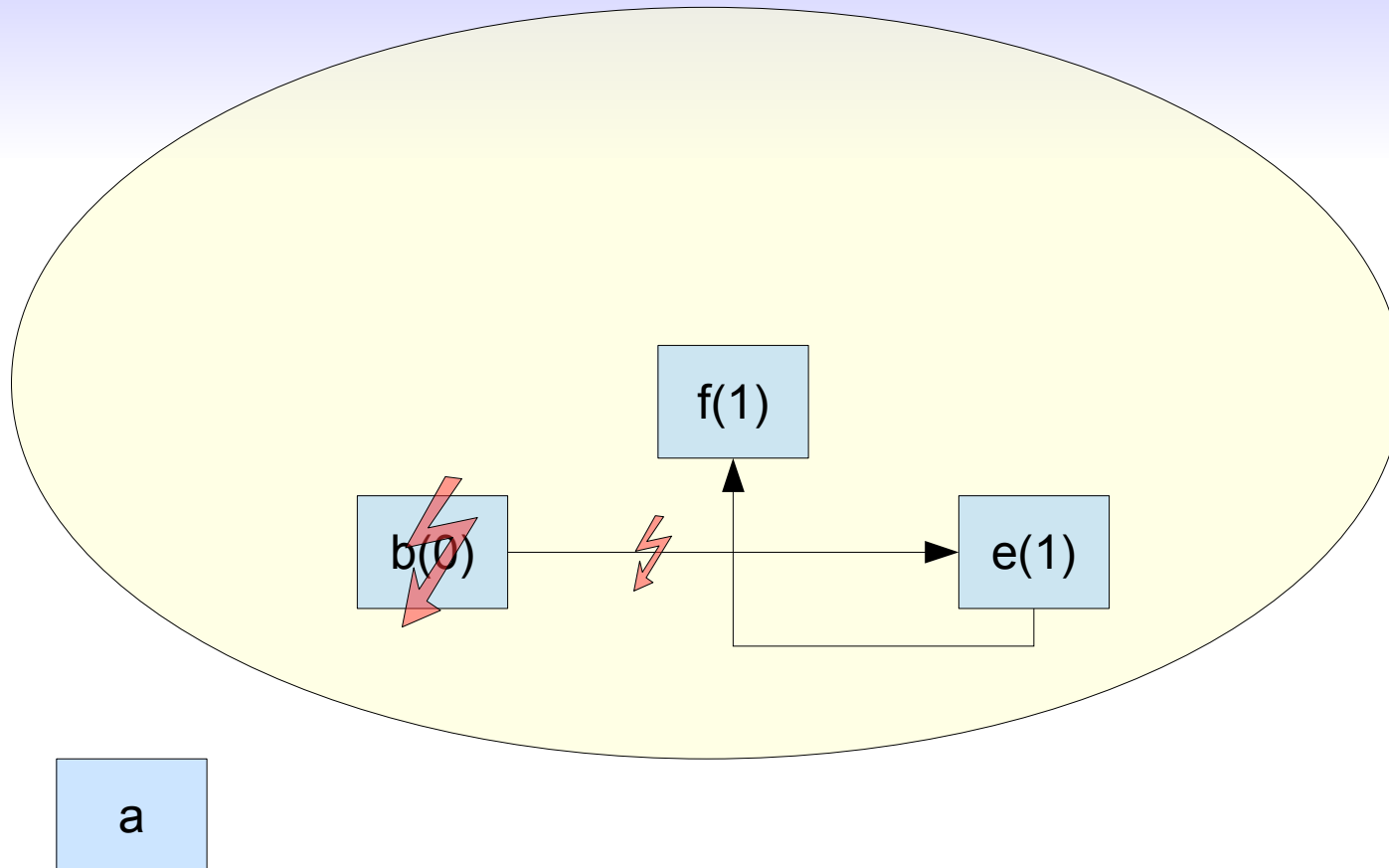
CPython: garbage collector



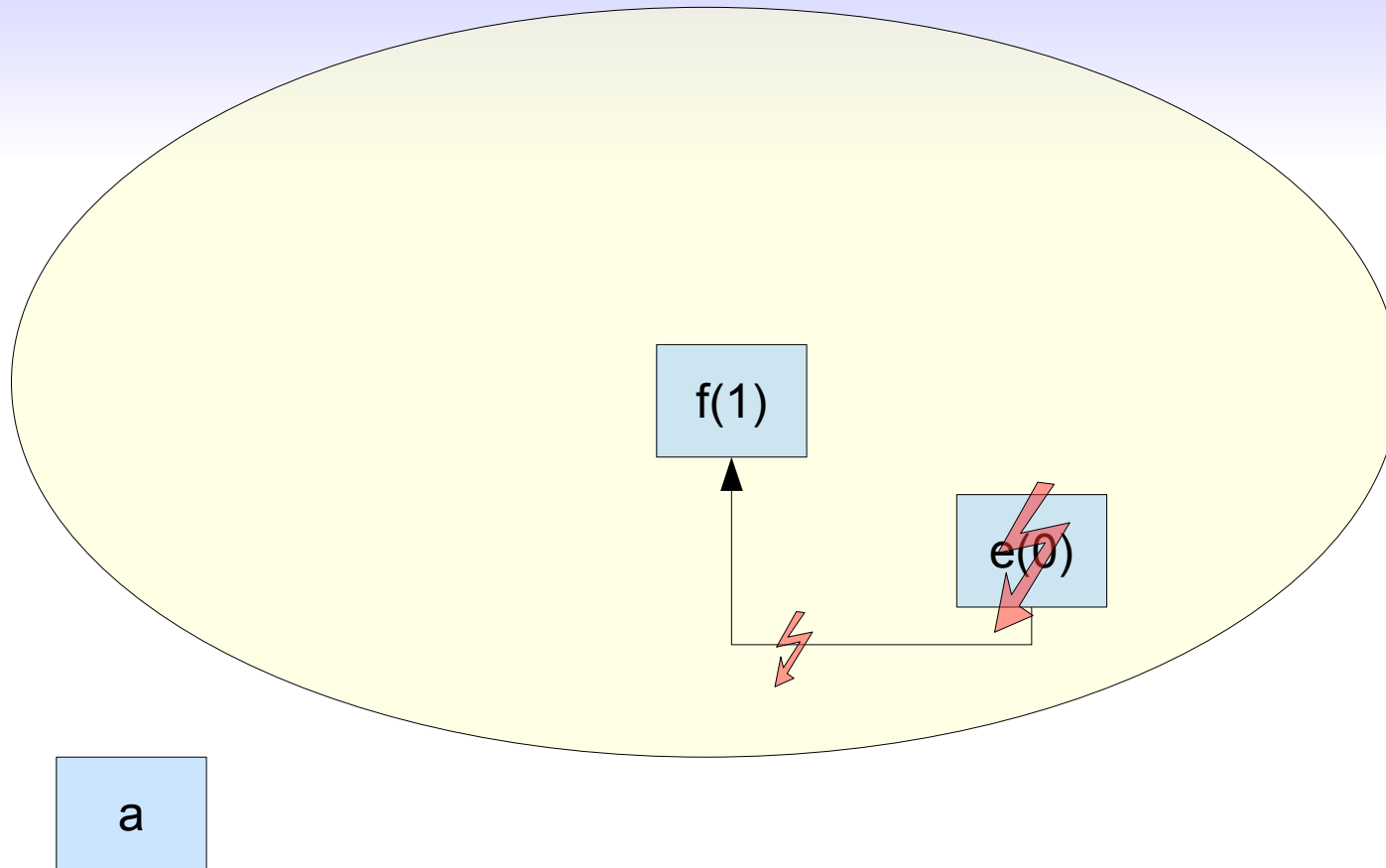
CPython: garbage collector



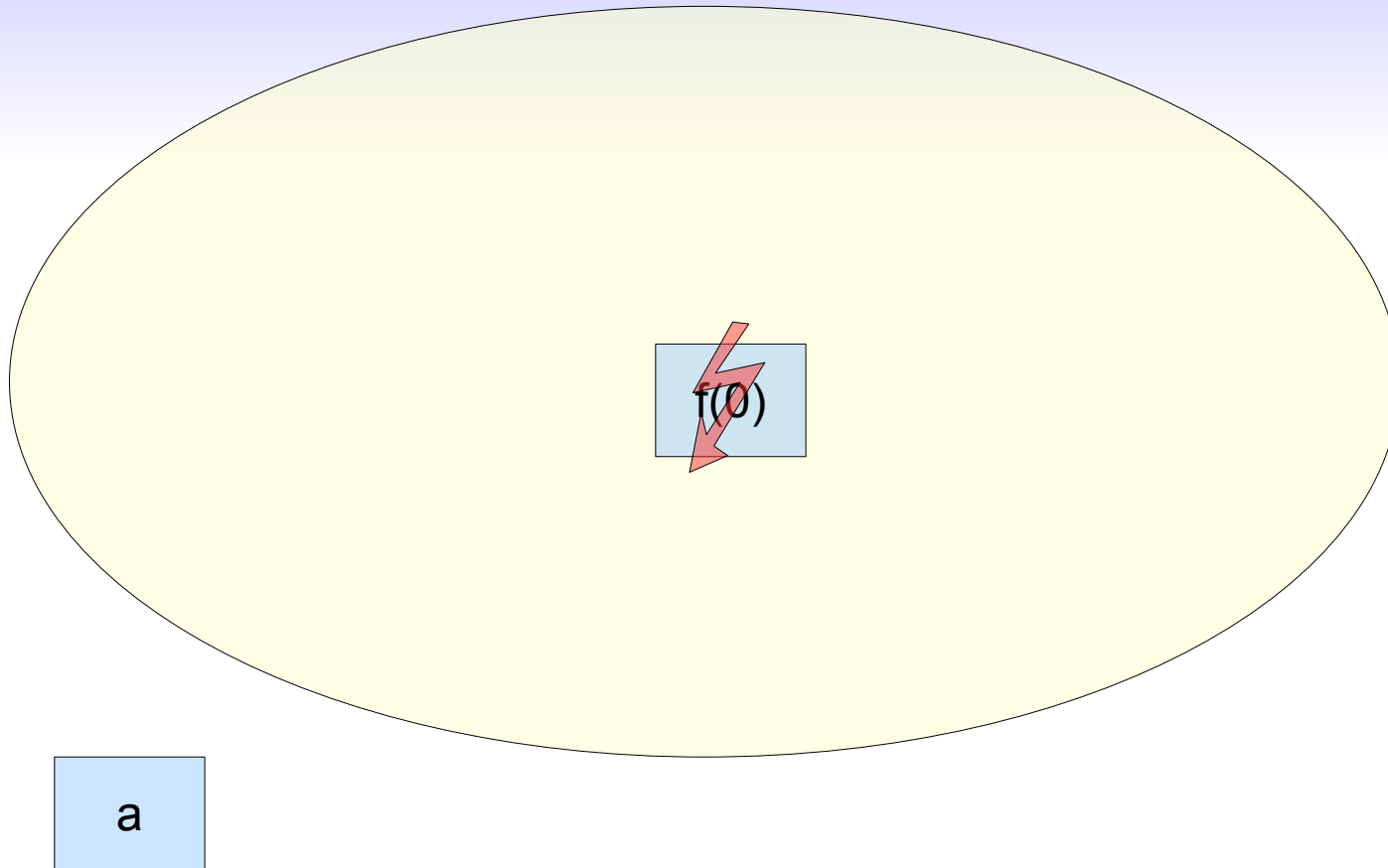
CPython: garbage collector



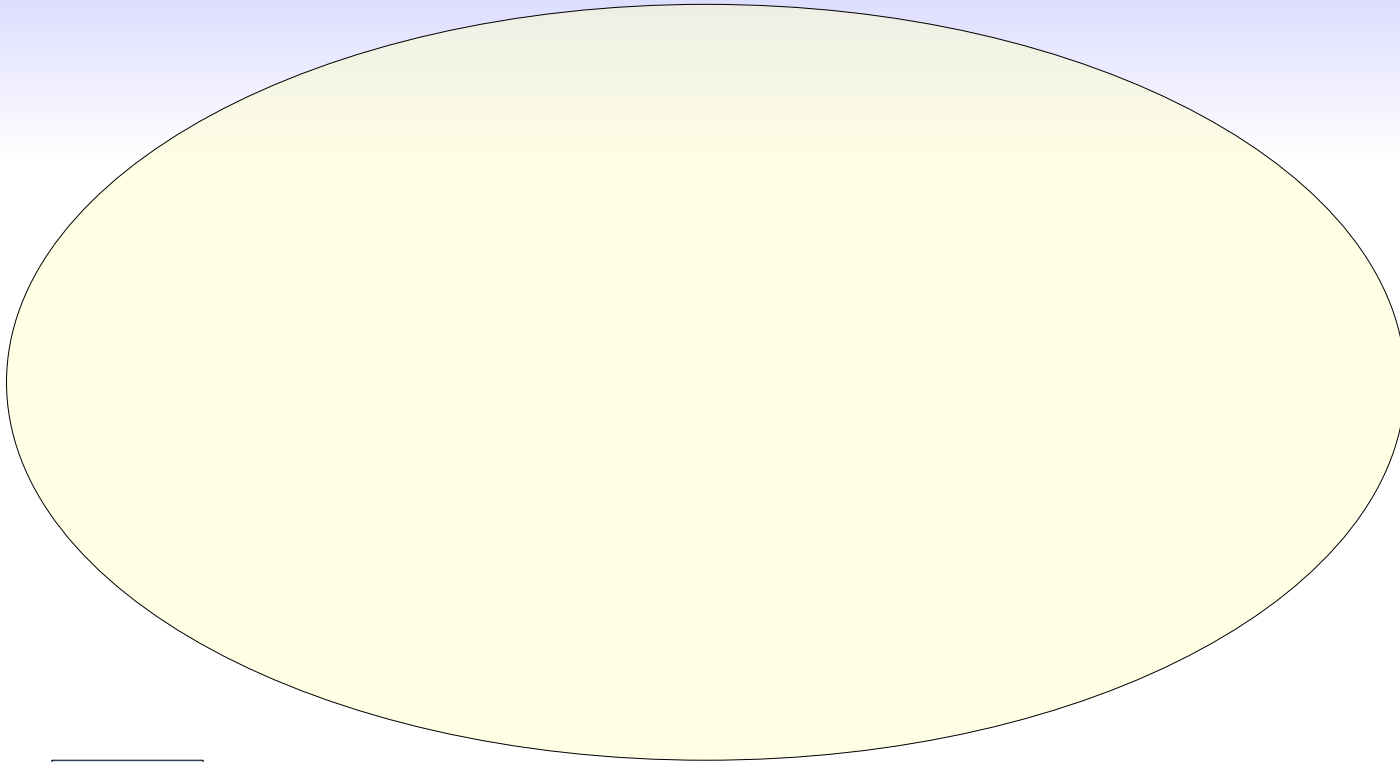
CPython: garbage collector



CPython: garbage collector



CPython: garbage collector



a

... finito!

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) Creazione di loop

- Riferimenti ancora in memoria

2) Esecuzione GC

- 4 oggetti distrutti
 - 2 istanze e 2 `__dict__`

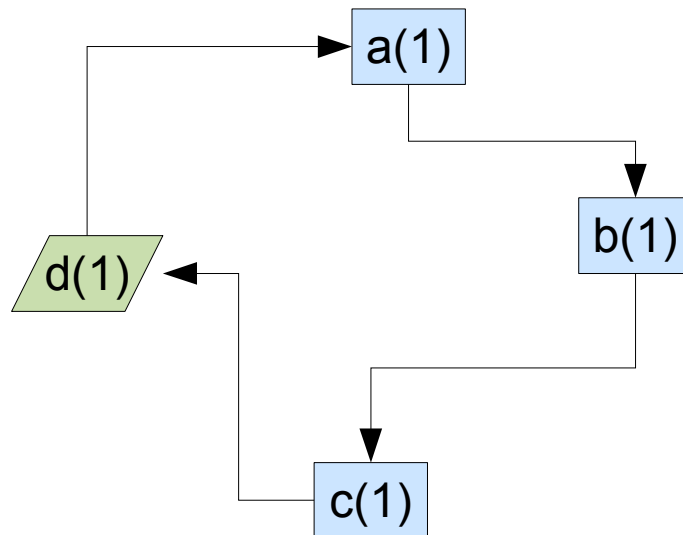
3) Nessun A rimasto!

Distruttori (__del__)

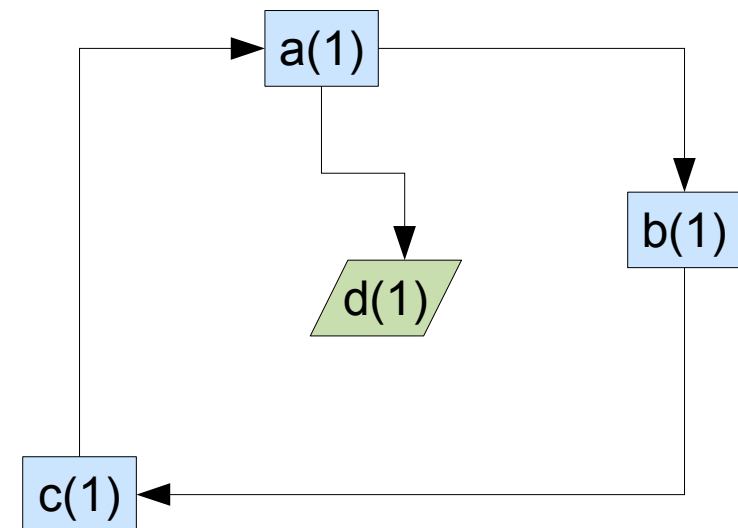
- GC **non funziona** su oggetti con distruttori
 - Motivo: interazione non chiara tra rimozione di riferimenti forzata e codice nella `__del__`
 - Vero object leak!
 - Solo se parte integrante del loop


Distruttori (`__del__`)

Leak!



OK!



 : con `__del__`

 : senza `__del__`

Distruttori (__del__)

- Usare `__del__` solo su classi “foglia”
 - In molti casi, non c'è bisogno
 - Casi reali: semplici wrapper di API open/close, lock/release, begin/end
 - Anche se ora c'è “with”!
- `gc.garbage`
 - Lista oggetti non liberabili dal GC
 - “`gc.collect(); assert not gc.garbage`”
 - Guardare qui in caso di guai...

Alternativa a `__del__`

- Distruttore “implicito” tramite weakref
 - weakref invocano un callback quando l'oggetto viene distrutto
 - Eseguire finalizzazione dal callback
 - Attenzione a non creare loop...

Callback dei weakref

```
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
```

- Semplice classe con `__del__`
- ... ma come si riscrive senza `__del__`?

Callback dei weakref

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

- Creo un weakref a self
 - Callback invocato quando self muore... proprio come `__del__`!
- `__del__` diventa il callback del weakref
- Finito? Non proprio...

Callback dei weakref

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

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

- Si è creato un loop!
 - self → weakref → callback → **nested scope** → self
- E' una regressione rispetto a `__del__`

Callback dei weakref

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

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

- Evitare il nested scope
 - Trucco “famoso”: valore di default argomento
- Sostituzione virtualmente perfetta di `__del__`

Memory leak in estensioni

- Memory leak in codice C
 - Py_INCREF senza Py_DECREF
 - Specialmente codice scritto a mano e poco testato
 - Diffidate delle estensioni non mature
 - Più sicure quelle che usano generatori di binding
- Leak creati su qualunque oggetto che maneggiano
 - Non solo quelli creati dall'estensione

Roadmap

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

Profiling codice Python

- Regola #1: ridurre accoppiamento e interdipendenza
 - Separare nettamente modello (core) da vista (GUI)
 - Es: possibilità di caricare/modificare/salvare progetti senza GUI
- Regola #2: creare un “test” automatico da profilare
 - Senza interazione dall'utente
 - Eseguitibile facilmente a linea di comando

Microprofiling

- timeit
 - Piccoli/medi snippet
 - Misura tempo medio di esecuzione
 - Precisione dinamica
 - Semplice uso a linea di comando

```
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**

- Anche su statement molto veloci (usec)
- Utile per confrontare diverse implementazioni
 - Non dice però dove viene perso tempo

Profiling

- Due tipologie
 - Deterministico (cProfile/profile)
 - Statistico (hotshot)
- API simile, funzionamento diverso
 - Deterministico: funzioni decorate, tracciamento chiamata / uscita
 - Statistico: ad intervalli regolari di tempo, guarda dove si trova
- cProfile al momento è più mantenuto

Profiling: esecuzione

```
def test_foo():  
    [...]  
  
import cProfile  
cProfile.run('test_foo()')
```

Da dentro il
codice

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

```
test_foo()
```

Esternamente

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

Profiling: esempio di output

389523 function calls in 2.637 CPU seconds

Ordered by: **standard name**

ncalls	totttime	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: migliorare la lettura

- Ordinamento:
 - cumtime: tempo d'esecuzione della funzione (dall'inizio alla fine)
 - python -m cProfile **-s internal**
 - tottime: tempo speso internamente alla funzione (non in sottofunzioni)
 - python -m cProfile **-s cumulative**
- Provare sempre entrambi
 - Mostrano aspetti diversi

Ordinamento per tempo cumulativo

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

Ordered by: **cumulative time**

ncalls	totttime	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>)

Ordinamento per tempo interno

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

Ordered by: **internal time**

ncalls	totttime	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>)

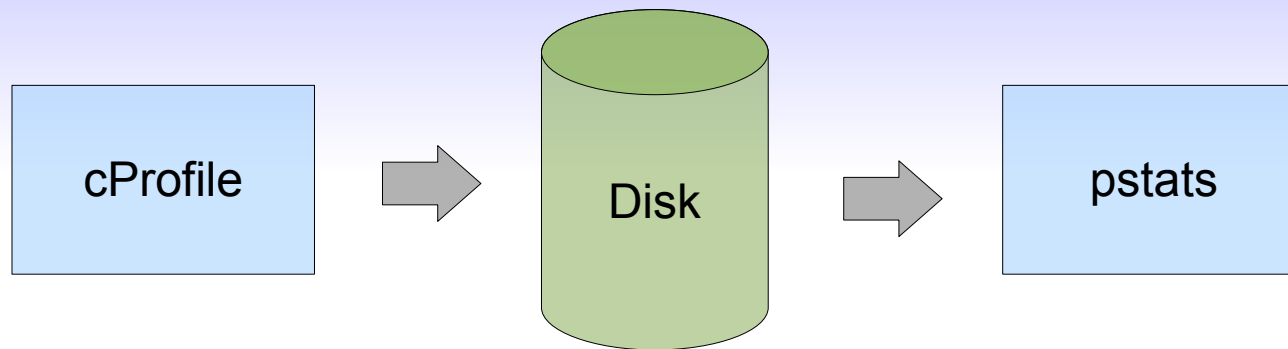
Naufragare tra i numeri...

- Un solo profiling non è **mai sufficiente**
- Produrre tanti dati per evitare falsi positivi
 - Provare cProfile e hotshots
 - Metodi diversi, risultati diversi
 - Provare set di dati diversi per verificare variabilità
 - Input diversi stressano punti diversi del codice
- Considerare overhead Python vs C/C++
 - cProfile tende a “dare molto peso” a Python a causa dell'overhead
 - In caso di dubbio: `time.time()`

Naufragare tra i numeri...

- In caso di poca visibilità:
 - Suddividere le funzioni chiave in più funzioni
 - Attenti overhead chiamata a funzione!
 - Usare un profiler C/C++ per trovare colli di bottiglia invisibili da Python
 - oprofile su Linux è fantastico!
- Attenzione al centro d'interesse
 - Lavorare sui grandi colli di bottiglia o inefficienze
 - Micro-ottimizzazioni spesso inutili

Modulo “pstats”



- Salvataggio risultati su disco
 - -o [output-file] / .run('foo', 'output-file')
- Analisi a posteriori (archivio storico)
 - ordinamenti diversi
 - filtri su funzioni da considerare

Qualche possibile non-ottimizzazione

- Psyco
 - Specializza “al volo” codice Python “simile al C”
 - Utile per codice che usa tipi base
 - Facilissimo da provare, risultati a volte ottimi
- Pyrex / Cython
 - Riscrittura “ottimizzata” di codice Python
 - O uso di librerie C/C++ già pronte (numpy, ecc.)

Roadmap

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

Debugging estensioni C/C++

- Debugging del codice C/C++ invocato da Python
 - Debug vero codice C/C++
 - Debug codice Python C API (binding)
- Post-mortem debugging
 - Segfault del processo: dove è successo?
- Necessità: estensioni Python con informazioni di debug, aggancio debugger interattivo

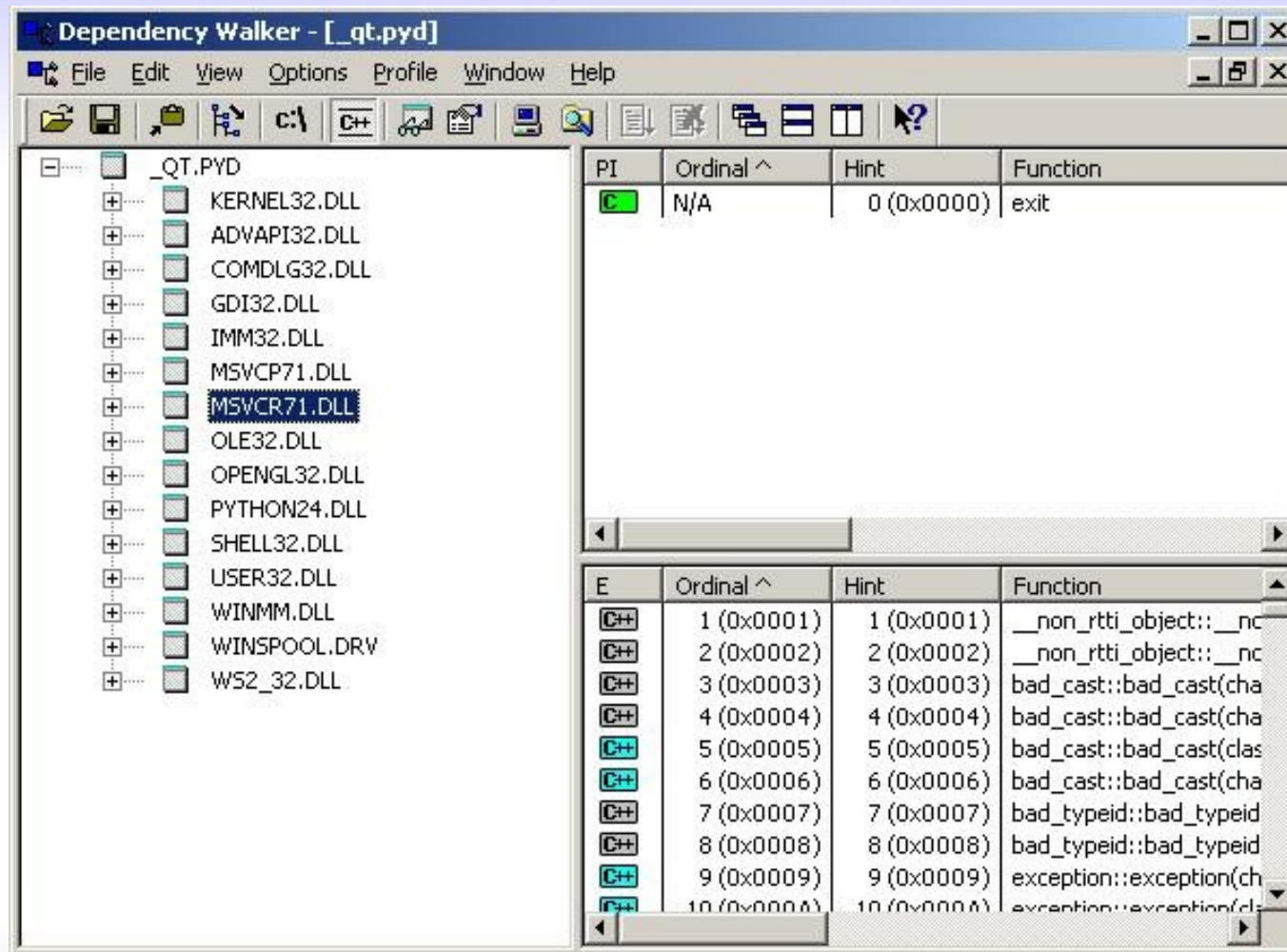
Windows: problemi di ABI

- ABI legato al runtime C (msvcr*)
 - Cambia tra runtime debug e release
 - Cambia tra versioni di Visual Studio (6, 7.1, ecc.)
- Librerie di runtime dinamiche:
 - Es: msvcr**71****d**.dll
 - **71** = Visual Studio 7.1 (aka .NET 2003)
 - **d** = debug

Windows: problemi di ABI

- Controllo di ABI: tool da utilizzare
 - depends.exe (<http://www.dependencywalker.com/>) (GUI)
 - PyInstaller's bindepend.py (cmd line)
- Verificare quale libreria msvc* viene usata
 - **Deve essercene solo una!**
 - La stessa usata da python*.dll
- Python 3.0 non risolverà questa situazione
 - Ma alcune parti (I/O) non usano più il runtime C

DependencyWalker



Versioni del runtime

Python 2.3	VC 6.0	MSVCRT.DLL
Python 2.4+	VC 7.1 (2003)	MSVCR71.DLL
Python 3.0	VC 9.0? (2008)	MSVCR90.DLL?

- La scelta del compilatore è importante
 - Compilatori diversi → runtime diversi
- Uso di MinGW in alternativa (per tutti i runtime)
 - <http://www.develer.com/oss/GccWinBinaries>
 - Integrato automaticamente con distutils

Compilazione in debug

- Necessario compilare in debug, ma con la stessa ABI:
 - Es: msvcr71.dll, **non msvcr71d.dll**
 - Altrimenti bisogna ricompilare tutto...
- Attenzione a Visual Studio (IDE)
 - Di solito, la modalità “debug” usa il runtime in debug
 - Nome runtime: “Multithreaded release”
 - Opzioni giuste: “/MD /Zi” (non /MDd)

Compilazione in debug

- Con distutils, non utilizzare “–debug”
 - Attiva “Py_DEBUG” e runtime in debug
 - 2 cambi di ABI non validi!
- Modificare lo script distutils a mano:

```
setup(  
    name = "FooBar",  
    ext_modules = [  
        Extension("foobar",  
            [...]  
  
        extra_compile_args = ["/Od", "/Zi"],  
        extra_link_args = ["/DEBUG"],  
    ),  
],  
)
```

Compilazione in debug

- E' possibile anche ricompilare python2X.dll in debug
 - Modificare progetto Visual Studio, versione “Release”
 - Aggiungere a mano opzioni giuste (come distutils)
 - Ricompilare
 - Copiare la DLL nella directory dalla quale viene lanciato lo script che si vuole debuggare

Attaccare il debugger

- In caso di segfault, “debug” lancia Visual Studio
 - Stack-trace completo con informazioni di debug
- Progetto auto-generato semplice da modificare
 - Specificare comando di avvio
 - “python c:\src\foo\start.py”
 - Current directory: “c:\src\foo”
 - Premere F5: esecuzione completa sotto debugger
 - Breakpoint, watch, ecc.

Debugging estensioni: Linux

- In generale, meno problemi:
 - Una sola ABI (nessuna differenza release/debug)
 - Ciascuna estensione con o senza simboli di debug
 - Forniti spesso dalla distro in un pacchetto separato
- GDB per visualizzare lo stacktrace e fare post-mortem debugging.
- Stesse modifiche a script di distutils richieste:

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

That's all, folks!

Domande?

Giovanni Bajo
<rasky@develer.com>