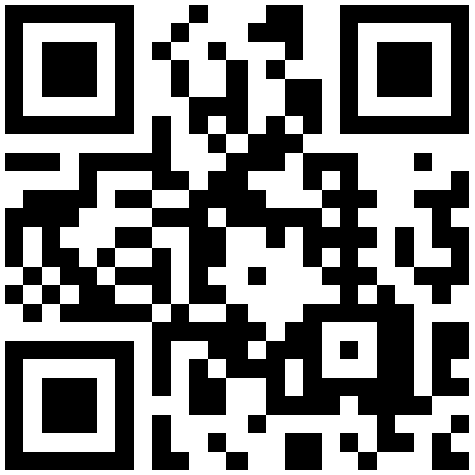
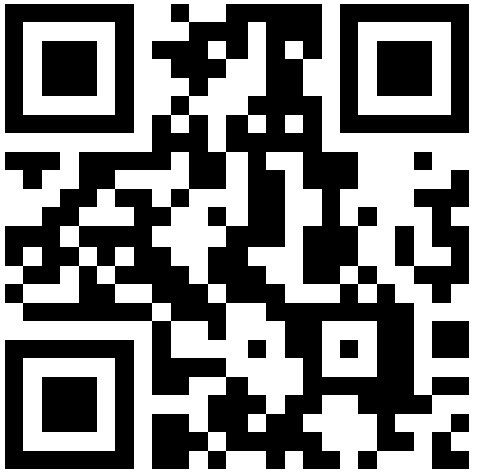
El módulo “tracemalloc”



Jesús Cea Avión
jcea@jcea.es
@jcea
httpS://www.jcea.es/
httpS://blog.jcea.es/



Jesús Cea Avión

- Programando Python desde 1996 (Python 1.4).
- *Core Developer* desde 2008 (Python 2.6 y 3.0).
- Fundador de Python Madrid y Python España, instrumental para la creación de Python Vigo.
  - Miembro de las dos Juntas Directivas.
  - Listas de correo, repositorio Mercurial, calendario de eventos, Twitter, OpenBadges.
- Consultor y *Freelance* a tu disposición.

tracemalloc

- Contabiliza el consumo en cada línea de código.
- Podemos comparar el uso de memoria en dos momentos diferentes.
- Permite identificar el “traceback” del punto en el que se ha creado un objeto concreto.
- Información capturada ajustable.
- Cero sobrecarga cuando no se usa.
- Buen complemento para el módulo estándar ***GC***.

tracemalloc

- La información se puede volcar a disco para analizarla en diferido o con otras herramientas.
- Utiliza los hooks del PEP 0445, muy interesante.
- Comparado con otros sistemas de profiling de memoria:
  - Incluído en la librería estándar.
  - Permite identificar dónde se ha creado un objeto concreto.
  - No clasifica la memoria por tipo de objeto.

tracemalloc

- Módulo en la librería estándar desde Python 3.4.0.
- Disponible para Python 2.7 y 3.3.
  - Requiere una versión parcheada del intérprete. Hace falta recompilarlo.
  - https://pypi.python.org/pypi/pytracemalloc/

Ejemplo: régimen estacionario (1)

```
import tracemalloc, threading, time, gc

def print_malloc() :
    while True :
        time.sleep(60)
        gc.collect()  # Probar a no hacerlo
        snapshot = tracemalloc.take_snapshot()
        top_stats = snapshot.statistics('lineno')
        print("OVERHEAD:",
              tracemalloc.get_tracemalloc_memory())
        print("MEMORIA:",
              tracemalloc.get_traced_memory())
        for stat in top_stats[:10] :
            print(stat)

tracemalloc.start()
t = threading.Thread(target=print_malloc)
t.setDaemon(True)
t.start()
```

Ejemplo: régimen estacionario (2)

- Ejecutando como “python3”:

```
OVERHEAD: 1157856
MEMORIA: (1968680, 2156452)
<frozen importlib._bootstrap>:222: size=622 KiB, count=1747,
average=365 B
<frozen importlib._bootstrap_external>:473: size=478 KiB,
count=5054, average=97 B
[...]
```

- Ejecutando “PYTHONTRACEMALLOC=1 python3”

```
OVERHEAD: 2689736
MEMORIA: (4583184, 5045464)
<frozen importlib._bootstrap_external>:473: size=1446 KiB,
count=16515, average=90 B
<frozen importlib._bootstrap>:222: size=806 KiB, count=2927,
average=282 B
[...]
```

Ejemplo: Volcado para análisis posterior

```
import tracemalloc, threading, time, gc

def print_malloc() :
    for i in range(9999999999) :
        time.sleep(60)
        gc.collect()
        snapshot = tracemalloc.take_snapshot()
        snapshot.dump("z-tracemalloc.%010d.snapshot" %i)

tracemalloc.start()
t = threading.Thread(target=print_malloc)
t.setDaemon(True)
t.start()
```

Ejemplo: *Leak* de memoria

```
>>> import tracemalloc
>>> a=tracemalloc.Snapshot.load("000011.snapshot")
>>> b=tracemalloc.Snapshot.load("000012.snapshot")
>>> for stat in b.compare_to(a, "lineno")[:2] :
...   print(stat)
...
./z.py:10: size=50.7 KiB (+3985 B), count=39 (+3),
average=1330 B
<frozen importlib._bootstrap>:222: size=622 KiB (+0 B),
count=1747 (+0), average=365 B
>>> len(b.compare_to(a, "lineno"))
943
>>> os.stat("000011.snapshot").st_size
85100
```

- El snapshot incluye información de toda la memoria gestionada, aunque no haya cambios.

Ejemplo: Identificar dónde se creó un objeto (1)

```
import tracemalloc
tracemalloc.start()
a="123"
print(tracemalloc.get_object_traceback(a))
a="123"*7
print(tracemalloc.get_object_traceback(a))
```

El resultado es:

```
None
./prueba.py:7
```

El primer caso no reserva memoria, el valor del objeto está directamente en el código compilado.

Ejemplo: Identificar dónde se creó un objeto (2)

```
>>> dis.dis(...)
  6           0 LOAD_CONST               1 ('123')
              3 STORE_FAST               0 (a)
[...]
  8          25 LOAD_CONST               1 ('123')
             28 LOAD_CONST               2 (7)
             31 BINARY_MULTIPLY
             32 STORE_FAST               0 (a)
```

- El primer caso no requiere reservar memoria.
- El segundo caso construye el valor del objeto en tiempo de ejecución.

Sugerencias:

- Volcar los “snapshots” a disco para análisis cómodo.
- Tener un hilo para acceso y exploración interactiva.

  https://docs.python.org/3.5/library/cmd.html
- Dependiendo de la aplicación, es preferible empezar agrupando por “filename” o por “lineno”.
- Tracemalloc se puede activar y desactivar en cualquier momento, podemos analizar secciones del programa de forma selectiva.

Usos prácticos:

- Memory leaks.
- Reducir el consumo de memoria:
  - Caso práctico:

    Reducir el consumo de memoria al 1.6% cambiando una secuencia [True, False, True...] por una matriz de bits:

    https://pypi.python.org/pypi/bitarray/

¿Rendimiento?

- No es para usar en producción en todo tu código:

```
#!/usr/bin/env python3

import timeit
import tracemalloc, gc

gc.disable()

print(timeit.timeit("[i for i in range(10000, 20000)]", number=1000))
tracemalloc.start()
print(timeit.timeit("[i for i in range(10000, 20000)]", number=1000))
tracemalloc.stop()
print(timeit.timeit("[i for i in range(10000, 20000)]", number=1000))
```

- Resultado:

```
0.788975897000455
5.260483931000635
0.771576245999313
```

¿Rendimiento?

- El consumo de memoria depende del número de “frames” que capturamos.
- `tracemalloc.clear_traces()`
- Puede no ser utilizable en entornos limitados (RaspPi) con programas complejos.

¿Preguntas?

- Visibilidad de la memoria de módulos en C.
- Combinación de los módulos tracemalloc y gc.
- Pros y contras entre identificar bloques de memoria e identificar tipos de objetos.
  - Depende de si el programa crea muchos objetos de tipo distinto o no. En Python es típico usar tipos básicos.

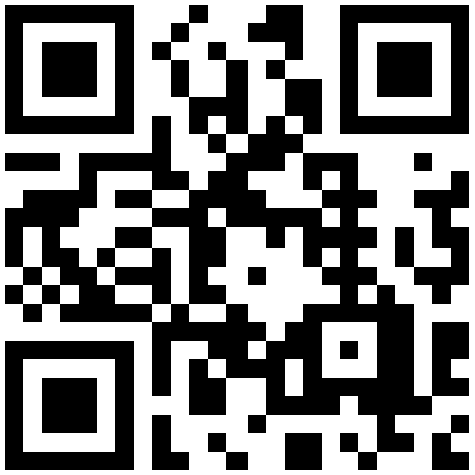
Referencias adicionales

- Documentación:
  https://pytracemalloc.readthedocs.org/
  https://docs.python.org/3.5/library/tracemalloc.html
  https://www.python.org/dev/peps/pep-0454/

  https://docs.python.org/3.5/library/gc.html

  https://www.python.org/dev/peps/pep-0445/
  https://docs.python.org/3.5/library/cmd.html
- Otras opciones de “profiling” de memoria:
  https://stackoverflow.com/questions/110259/

¡Gracias!



Jesús Cea Avión
jcea@jcea.es
@jcea
httpS://www.jcea.es/
httpS://blog.jcea.es/