Dokąd idą pythonowe obiekty po śmierci?

… czyli o tym czego (prawdopodobnie) nie wiedziałeś o Garbage Collectorze

Mateusz Mazurek

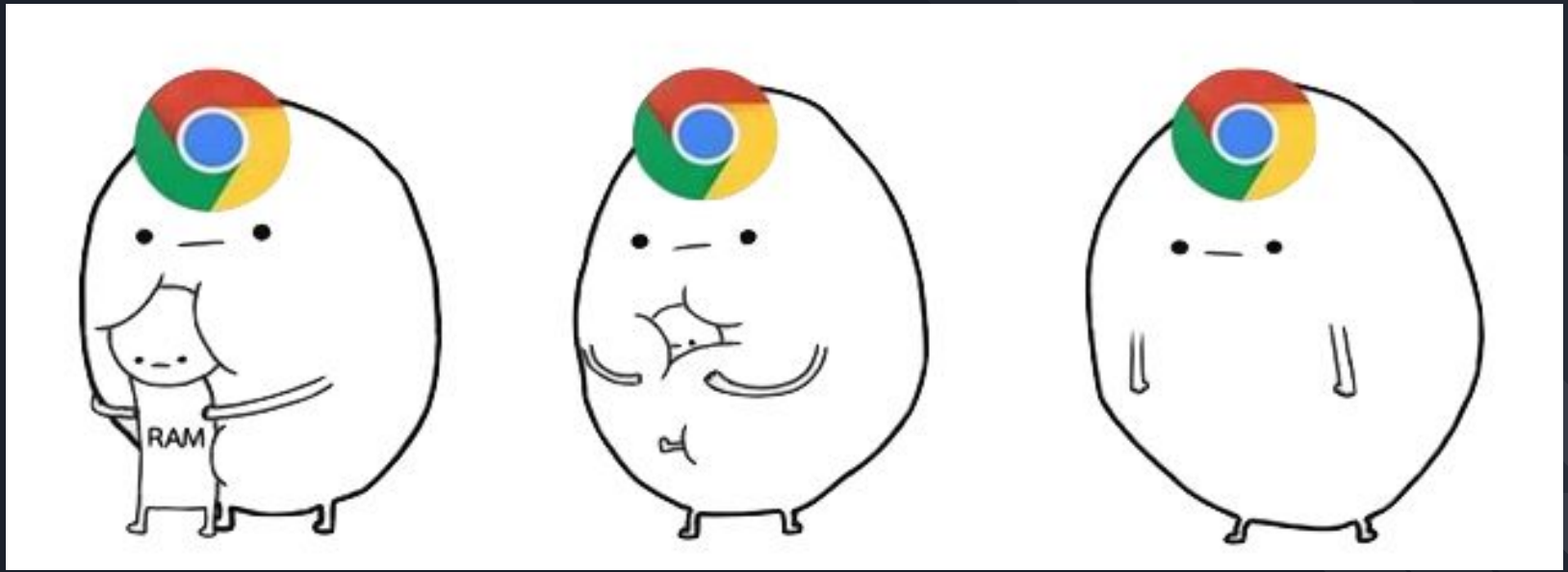
- 10+ lat doświadczenia
- PHP/JS
- Java/JS
- LUA/MySql/Postgresql/Redis
- Python

CloudFerro

CloudFerro

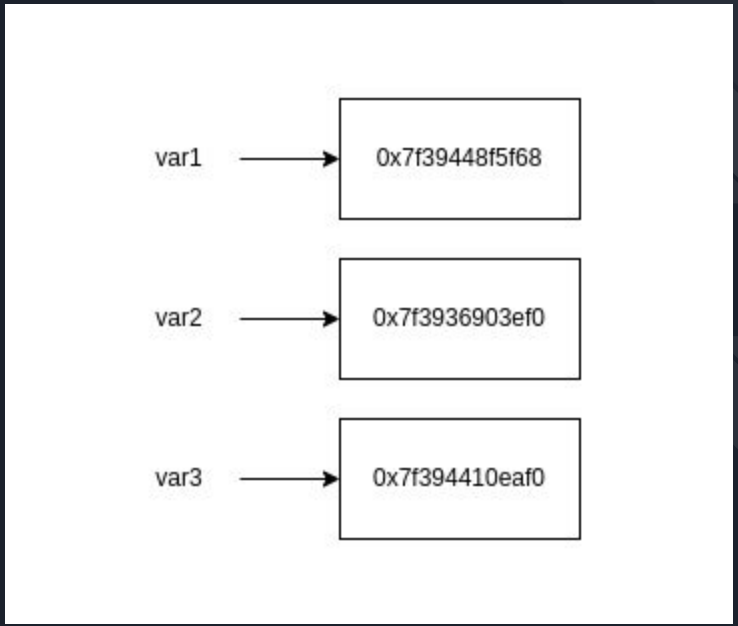
Pamięć RAM

```c
// C Program to dynamically allocate an int ptr
#include <stdio.h>
#include <stdlib.h>

int main() {
    // dynamically allocated variable, sizeof(char) = 1 byte
    char *ptr = (char *)malloc(sizeof(char));

    if (ptr == NULL) {
        printf("Memory Error!\n");
    } else {
        *ptr = 'S';
        printf("%c", *ptr);
    }

    // deallocating memory pointed by ptr
    free(ptr);

    printf("\n%c ", *ptr);

    // assign NULL to avoid garbage values
    ptr = NULL;

    return 0;
}
```

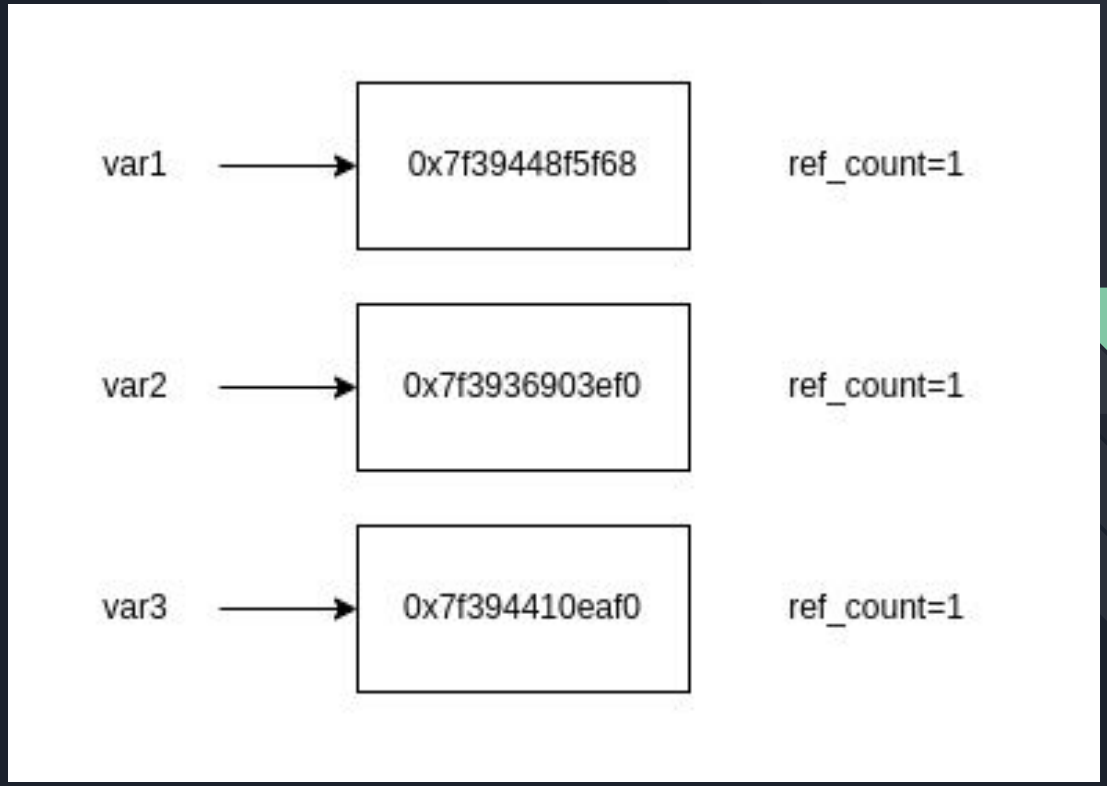

RAM

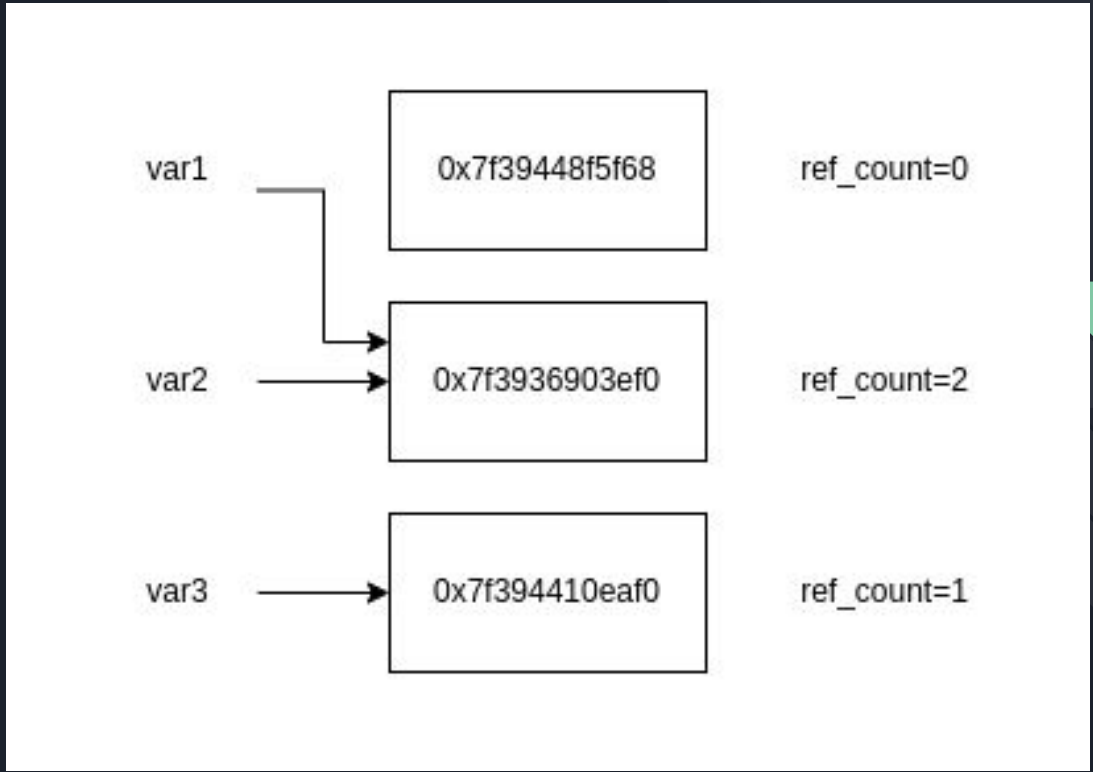
Jak zarządzanie pamięcią działa w Pythonie?

Główny algorytm: zliczanie referencji.


var1
0x7f39448f5f68
var2
0x7f3936903ef0
var3
0x7f394410eaf0

```python
var1 = [1, 2, 3]

var2 = "1"

var3 = range(10)

print(hex(id(var1)), hex(id(var2)), hex(id(var3)))

# output: 0x7f89a4137300 0x7f89a4900840 0x7f89a410eac0
```

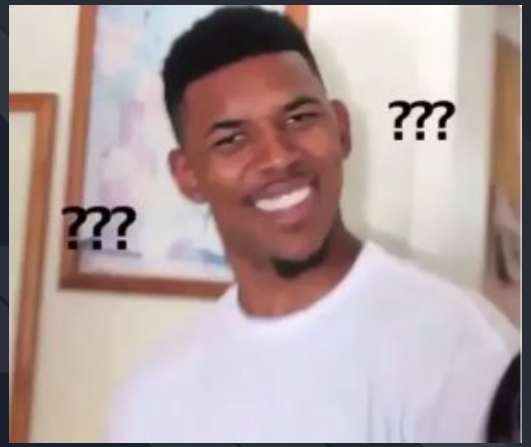

var1
0x7f39448f5f68
ref_count=1
var2
0x7f3936903ef0
ref_count=1
var3
0x7f394410eaf0
ref_count=1


var1
var2
var3
0x7f39448f5f68
0x7f3936903ef0
0x7f394410eaf0
ref_count=0
ref_count=2
ref_count=1

```python
import sys

var1 = 10

print(sys.getrefcount(var1))
```

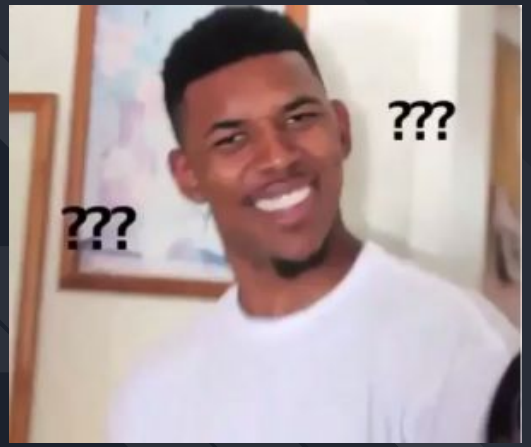
```python
import sys

var1 = 10

print(sys.getrefcount(var1))

#output: 1000000011
```

```python
import sys

var1 = 10

print(sys.getrefcount(var1))

#output: 1000000011
```



```python
import sys

var1 = object()

print(sys.getrefcount(var1))
```

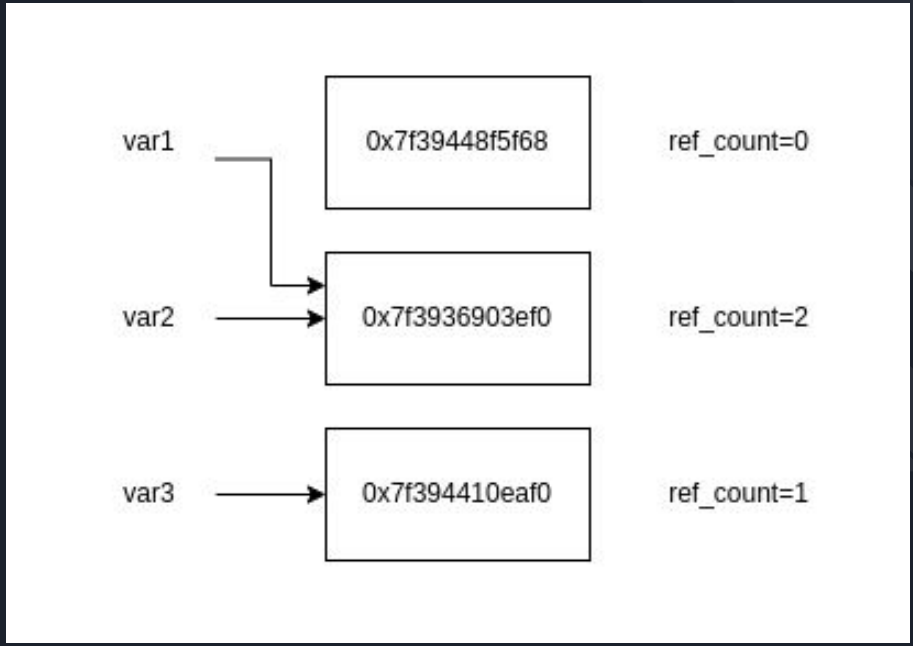
```python
import sys

var1 = object()

print(sys.getrefcount(var1))

# output: 2
```



```python
import sys

var1 = object()

var2 = var1

var3 = var1

print(sys.getrefcount(var1))
```

```python
import sys

var1 = object()

var2 = var1

var3 = var1

print(sys.getrefcount(var1))

# output: 4
```

Główny algorytm: zliczanie referencji.


var1
var2
var3
0x7f39448f5f68
0x7f3936903ef0
0x7f394410eaf0
ref_count=0
ref_count=2
ref_count=1

```python
import sys

a = []
a.append(a)

print(sys.getrefcount(a))
```

```python
import sys

a = []
a.append(a)

print(sys.getrefcount(a))

# output: 3
```

```python
import sys

a = []
a.append(a)

print(sys.getrefcount(a))

del a
```

Algorytm zliczania referencji nie radzi sobie z cyklami!

Garbage Collector

Większość obiektów w naszych programach żyje albo bardzo krótko albo bardzo długo.

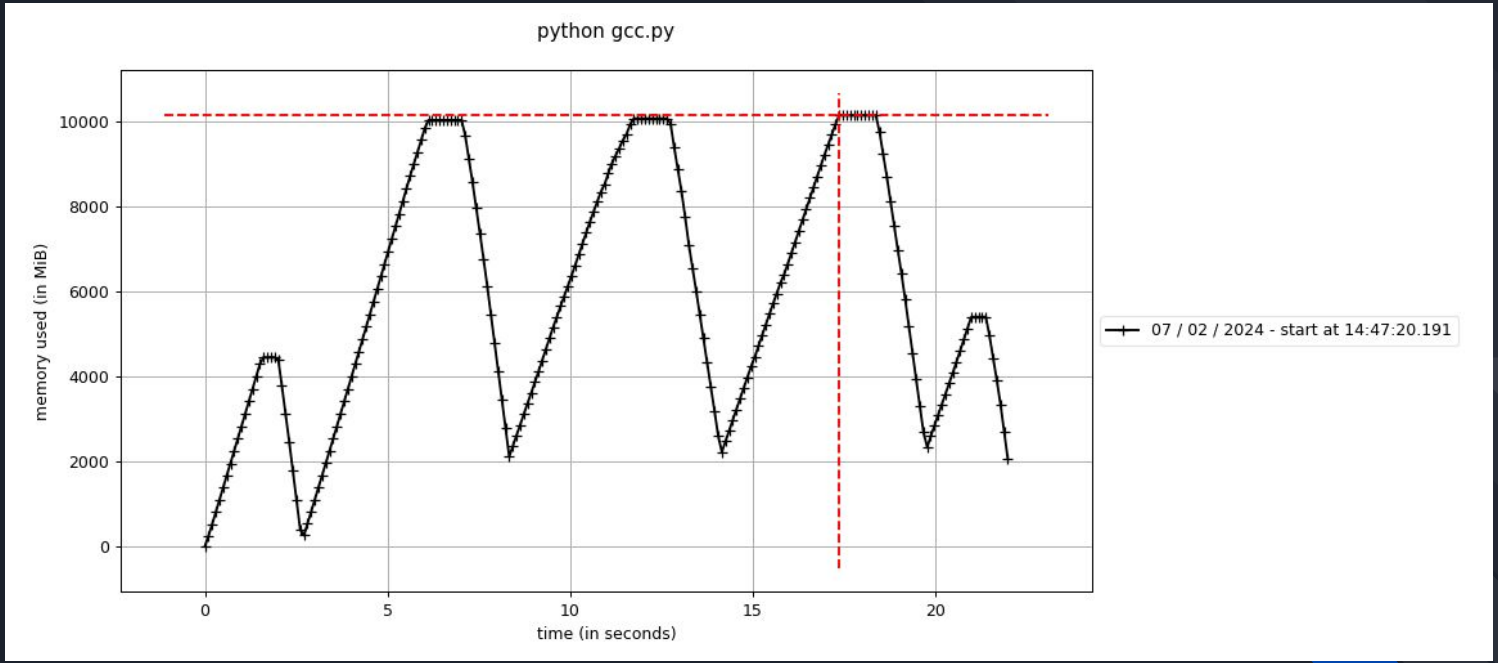
G1								
G2								
G3								

```python
import gc

class Test:
    pass

test = Test()

print(test)  # <__main__.Test object at 0x7fa75fe19110>

print(test in gc.get_objects(generation=0))  # True

gc.collect(generation=0)  # manualne odpalenie garbage collector

print(test in gc.get_objects(generation=0))  # False

print(test in gc.get_objects(generation=1))  # True
```

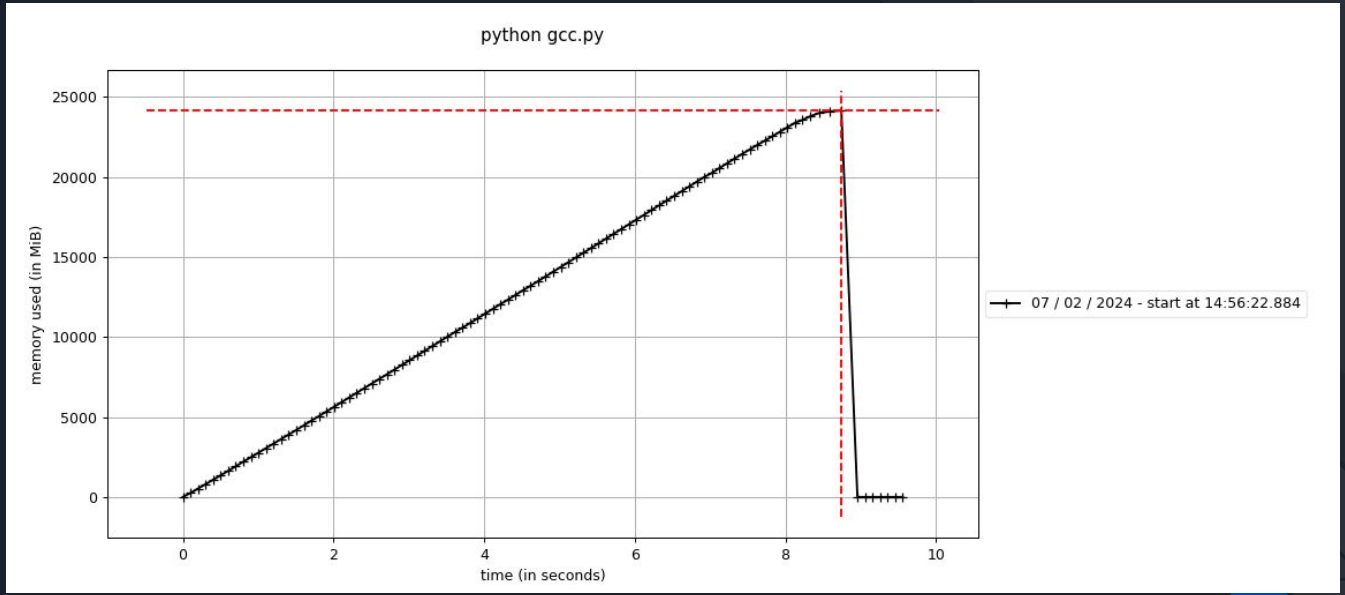
```python
import gc


class Test:
    def __del__(self):
        print("Usuwam obiekt")


gc.collect()  # czyścimy pamięć, żeby mieć pusty tan początkowy
print(gc.get_count())  # (0, 0, 0)

test = Test()
test.tt = test

print(gc.get_count())  # (3, 0, 0)

del test

print(gc.get_count())  # (3, 0, 0)


gc.collect()  # truggeruje printa "Usuwam obiekt"
print(gc.get_count())  # (0, 0, 0)
```

```python
import gc

print(gc.get_threshold())  # 700, 10, 10
```

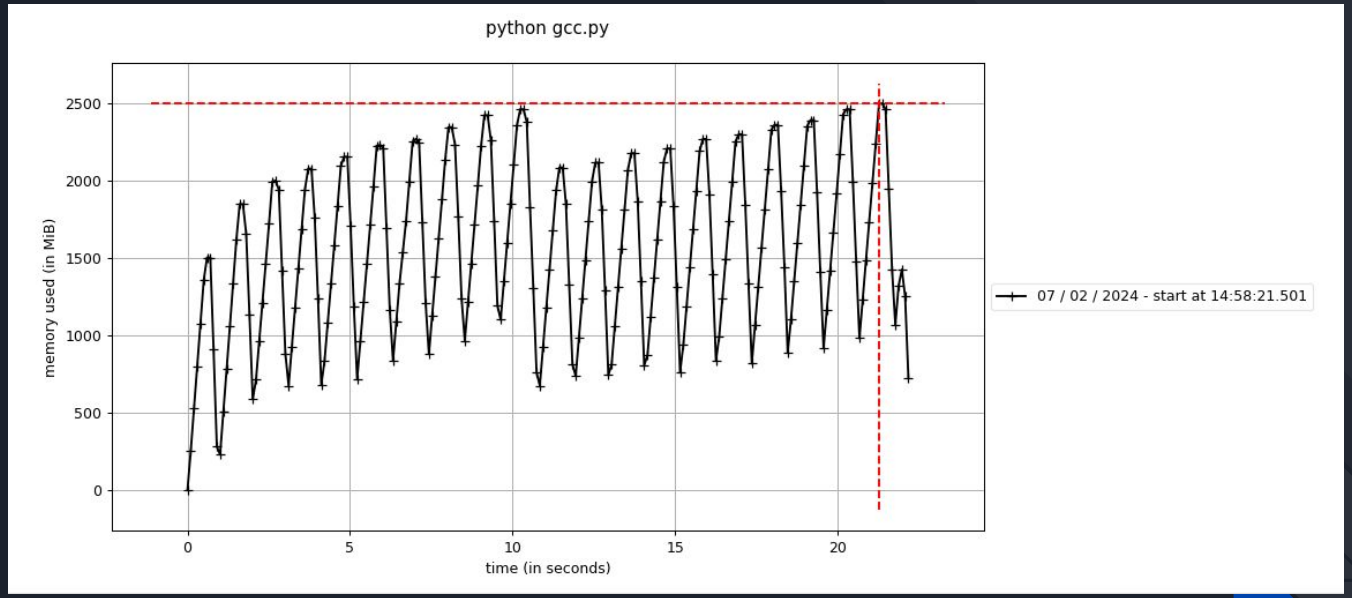
```python
from time import sleep
import gc


class A:
    def __init__(self):
        self.vars = list(range(1000000))


gc.callbacks.append(lambda x, y: print(x, y))

for _ in range(1000):
    a = A()
    a.self_ref = a
    sleep(0.00001)


"""
start {'generation': 0, 'collected': 0, 'uncollectable': 0}
stop {'generation': 0, 'collected': 414, 'uncollectable': 0}

start {'generation': 0, 'collected': 0, 'uncollectable': 0}
stop {'generation': 0, 'collected': 777, 'uncollectable': 0}

start {'generation': 0, 'collected': 0, 'uncollectable': 0}
stop {'generation': 0, 'collected': 774, 'uncollectable': 0}

start {'generation': 0, 'collected': 0, 'uncollectable': 0}
stop {'generation': 0, 'collected': 777, 'uncollectable': 0}

start {'generation': 2, 'collected': 0, 'uncollectable': 0}
stop {'generation': 2, 'collected': 262, 'uncollectable': 0}
"""
```


python gcc.py
memory used (in MiB)
10000
8000
6000
4000
2000
0
0
5
10
15
20
time (in seconds)
07 / 02 / 2024 - start at 14:47:20.191

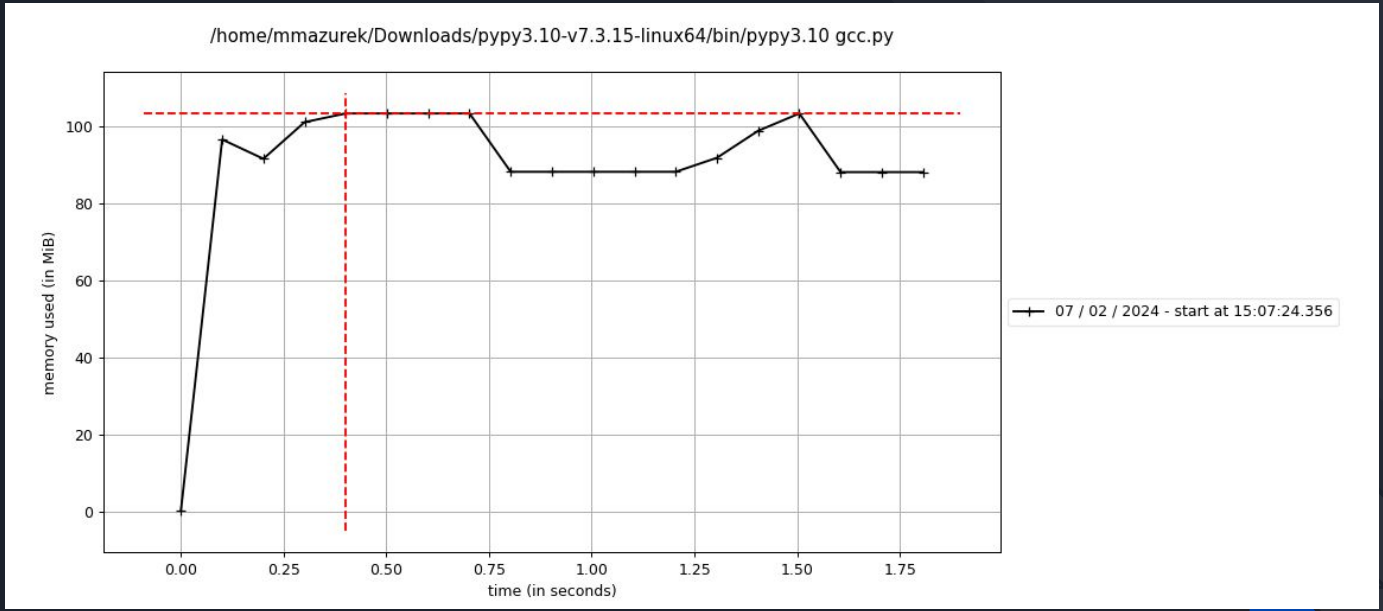
```python
from time import sleep
import gc


class A:
    def __init__(self):
        self.vars = list(range(1000000))


gc.disable()

for _ in range(1000):
    a = A()
    a.self_ref = a
    sleep(0.00001)
```

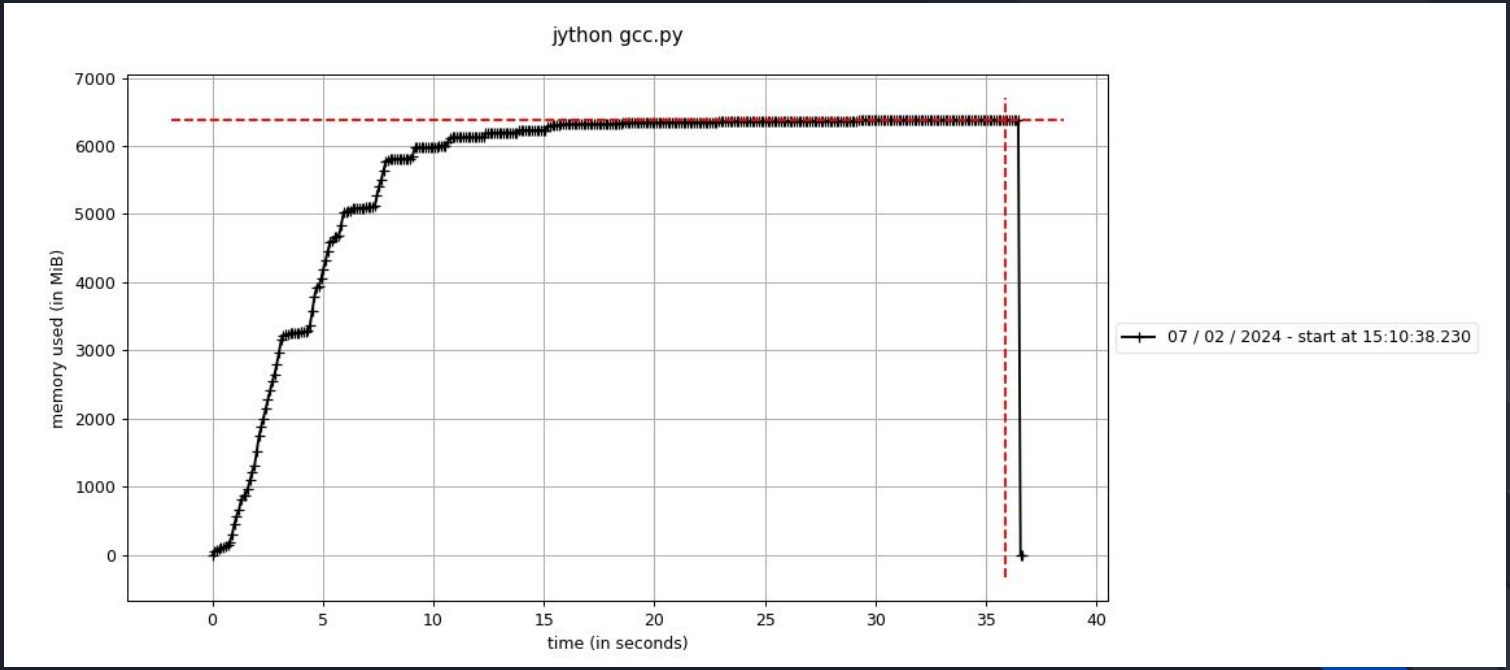

python gcc.py
memory used (in MiB)
25000
20000
15000
10000
5000
0
0
2
4
6
8
10
time (in seconds)
07 / 02 / 2024 - start at 14:56:22.884

```python
from time import sleep
import gc


class A:
    def __init__(self):
        self.vars = list(range(1000000))


gc.set_threshold(100, 10, 10)

for _ in range(1000):
    a = A()
    a.self_ref = a
    sleep(0.00001)
```


python gcc.py
memory used (in MiB)
2500
2000
1500
1000
500
0
0
5
10
15
20
time (in seconds)
07 / 02 / 2024 - start at 14:58:21.501

Czy GC w każdym Pythonie działa tak samo?


/home/mmazurek/Downloads/pypy3.10-v7.3.15-linux64/bin/pypy3.10 gcc.py
memory used (in MiB)
100
80
60
40
20
0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
time (in seconds)
07 / 02 / 2024 - start at 15:07:24.356


jython gcc.py
memory used (in MiB)
7000
6000
5000
4000
3000
2000
1000
0
0
5
10
15
20
25
30
35
40
time (in seconds)
07 / 02 / 2024 - start at 15:10:38.230

W Pythonie nie każda referencja się liczy!

```python
import sys
import weakref


class Test:
    pass


test = Test()

print(test)  # <__main__.Test object at 0x7f5f64841670>

print(sys.getrefcount(test))  # 2

weakref_test = weakref.ref(test)

print(sys.getrefcount(test))  # 2

print(weakref_test)  # <weakref at 0x7f5f6483dae0; to 'Test' at 0x7f5f64841670>

del test  # usuwamy ostatnią silną referencję do obiektu, co się stanie ze słabą referencją?

print(weakref_test)  # <weakref at 0x7f5f6483dae0; dead>
```

```python
import weakref


class Test:
    pass


test = Test()

print(test)  # <__main__.Test object at 0x7f3017a460d0>

weakref_test = weakref.ref(test)
print(weakref_test)  # <weakref at 0x7f3017873a40; to 'Test' at 0x7f3017a460d0>
print(weakref.getweakrefs(test))  # [<weakref at 0x7f3017873a40; to 'Test' at 0x7f3017a460d0>]
print(weakref.getweakrefcount(test))  # 1
```

```python
import weakref


class Test:
    pass


test = Test()

weakref_set = weakref.WeakSet({test, Test()})

print(weakref_set.data)

"""
{<weakref at 0x7f972ec31b80; to 'Test' at 0x7f972edfb0d0>}

Tylko jeden element, bo drugi od razu został sprzątnięty.
"""

del test  # usuwamy ostatnią silną referencję

print(weakref_set.data)

"""
set()
czyli pusto.
"""
```

```python
import weakref


class OwnWeakSet:
    def __init__(self, *args):
        self._inner_set = set()
        for arg in args:
            self.add(arg)

    @property
    def data(self):
        return self._inner_set

    def _remove(self, obj):
        self._inner_set.remove(obj)

    def add(self, obj):
        obj_weakref = weakref.ref(obj, self._remove)  # tu użyty drugi argument
        self._inner_set.add(obj_weakref)


class Test:
    pass


test = Test()

my_weak_set = OwnWeakSet()
my_weak_set.add(test)

print(my_weak_set.data)

"""
{<weakref at 0x7f53f8356c20; to 'Test' at 0x7f53f84472e0>}
"""

del test

print(my_weak_set.data)  # set()
```

Podsumowując!

Dziekuje bardzo za uwagę!

Q&A

Quiz

https://joinmyquiz.com/