Zarządzanie pamięcią w Pythonie - jak działa i czy warto go dotykać?

Czy w Pythonie zdarzają się wycieki pamięci? Jaki jest właściwie koszt działania garbage collectora? Kiedy mogą pojawić się problemy z pamięcią i jak możemy próbować je rozwiązać?

Maciej Pytel

codilime®
CREATING VALUE

02/03/2015

O mnie

Software developer @ CodiLime. Piszę (głównie) w Pythonie. Lubię rozumieć jak rzeczy działają "w środku".

Disclaimer

Który Python?

Ten talk jest o Pythonie 2.7.
Mówiąc Python mam na myśli CPython.

Przykłady

`https://github.com/MaciekPytel/python-memory-talk`

1. Trochę teorii
2. Czemu warto o tym wszystkim wiedzieć? Co i jak możemy zrobić?
   - Przykład: RESTful server
   - Wycieki pamięci
3. Kilka przydatnych narzędzi
4. Trochę więcej teorii, czyli jak to właściwie działa

Część 1

Trochę teorii

Zliczanie referencji

- Każdy obiekt ma licznik referencji.
- W momencie gdy liczba referencji osiągnie 0 obiekt jest *natychmiast* dealokowany.
- Co z cyklami referencji?

Garbage collector

Zadanie: usunąć obiekty w nieosiągalnych cyklach referencji.
Rozwiązanie: garbage collector.

- Uruchamiany co jakiś czas przez interpreter Pythona.
- Nie dealokuje obiektów, a jedynie zrywa cykle referencji.
- ”Zawiesza” proces podczas zbierania śmieci.
- Drogi: koszt działania (co najmniej) liniowy od liczby wszystkich referencji w programie.

Generacyjny GC

Weak generational hypothesis

Większość obiektów w programie ma bardzo krótki, albo bardzo długi czas życia.

Czy to może nam pomóc w optymalizacji GC?

Generacyjny GC

- Dzielimy obiekty na 3 generacje (0, 1, 2).
- Każdy nowo alokowany obiekt trafia do generacji 0.
- Dłużej istniejące obiekty znajdują się w wyższych generacjach.
- Na generację 0 składa się niedużo obiektów, które z dużym prawdopodobieństwem można zwolnić (zgodnie z hipotezą powyżej).

Generacyjny GC

- Garbage collection n-tej generacji to taki, w którym GC przeanalizował obiekty generacji 0..n.
- Obiekt, który przeżył GC n-tej generacji jest promowany do generacji n + 1 (technicznie min(2, n + 1)).
- GC generacji 0 jest tanie - przeprowadzamy je często.
- GC wyższych generacji są droższe - przeprowadzamy je znacznie rzadziej.

Słabe referencje

- Referencje nie zwiększające licznika referencji wskazywanego przez siebie obiektu.
- Istnienie słabej referencji nie zapobiega dealokacji obiektu.
- W Pythonie dostarczane przez moduł *weakref*.
- Dereferencja słabej referencji do zwolnionego obiektu zwraca None.

Część 2

Czemu warto o tym wszystkim wiedzieć? Co i jak możemy zrobić?

Punkt 1

Przykład: RESTful server

Opis eksperymentu

- Serwer:
  - Flask-RESTful + gunicorn.
  - Dwie klasy: Task i Subtask.
  - Relacja jeden-do-wielu (każdemu Task'owi odpowiada wiele Subtask'ów).
  - API pozwala dodawać, odczytywać i usuwać Taski i Subtaski.
- Test: uruchomić kilku klientów wysyłających w pętli żądania i mierzyć wydajność.

Task

```
class Task(object):
    def __init__(self, name):
        self.name = name
        self._subtasks = {}

    def add_subtask(self, name):
        self._subtasks[name] = Subtask(name, self)

    def get_subtask(self, name):
        return self._subtasks.get(name, None)
```

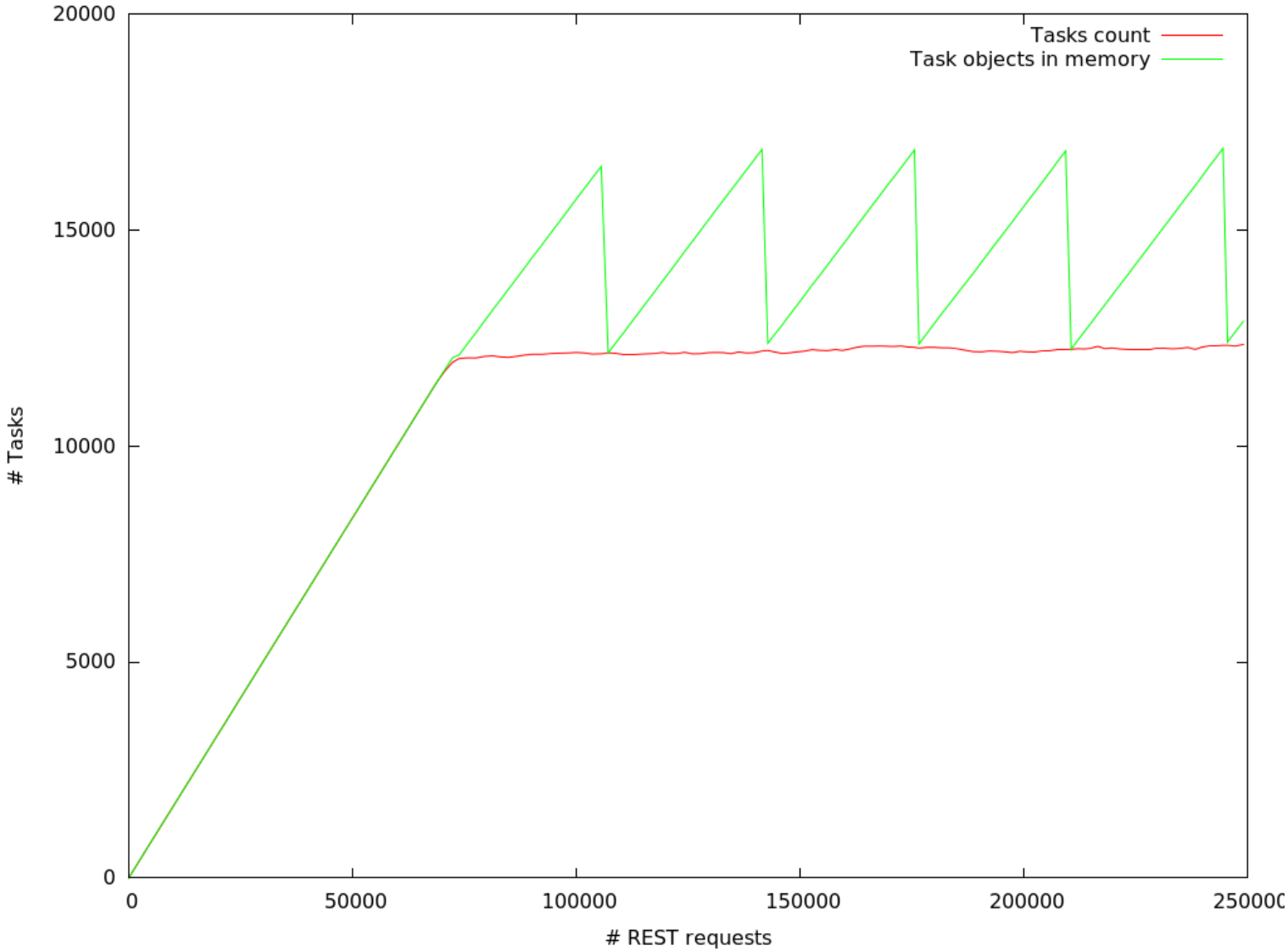
Subtask wersja 1

```
class Subtask(object):
    def __init__(self, name, task):
        self.name = name
        self._task = task

    def get_task(self):
        return self._task
```

REST API

- **GET /tasks/{task_id}/** - zwróć listę Subtask'ów powiązanych z Task'iem o danym id (JSON)
- **PUT /tasks/{task_id}/** - stwórz Task o zadanym id
- **DELETE /tasks/{task_id}/** - usuń Task i wszystkie jego Subtask'i
- **GET /subtasks/{subtask_id}/** - zwróć Task, do którego przypisany jest dany Subtask
- **PUT /subtasks/{task_id}/{subtask_id}** - stwórz Subtask o danym id i przypisz do Task'a o danym id

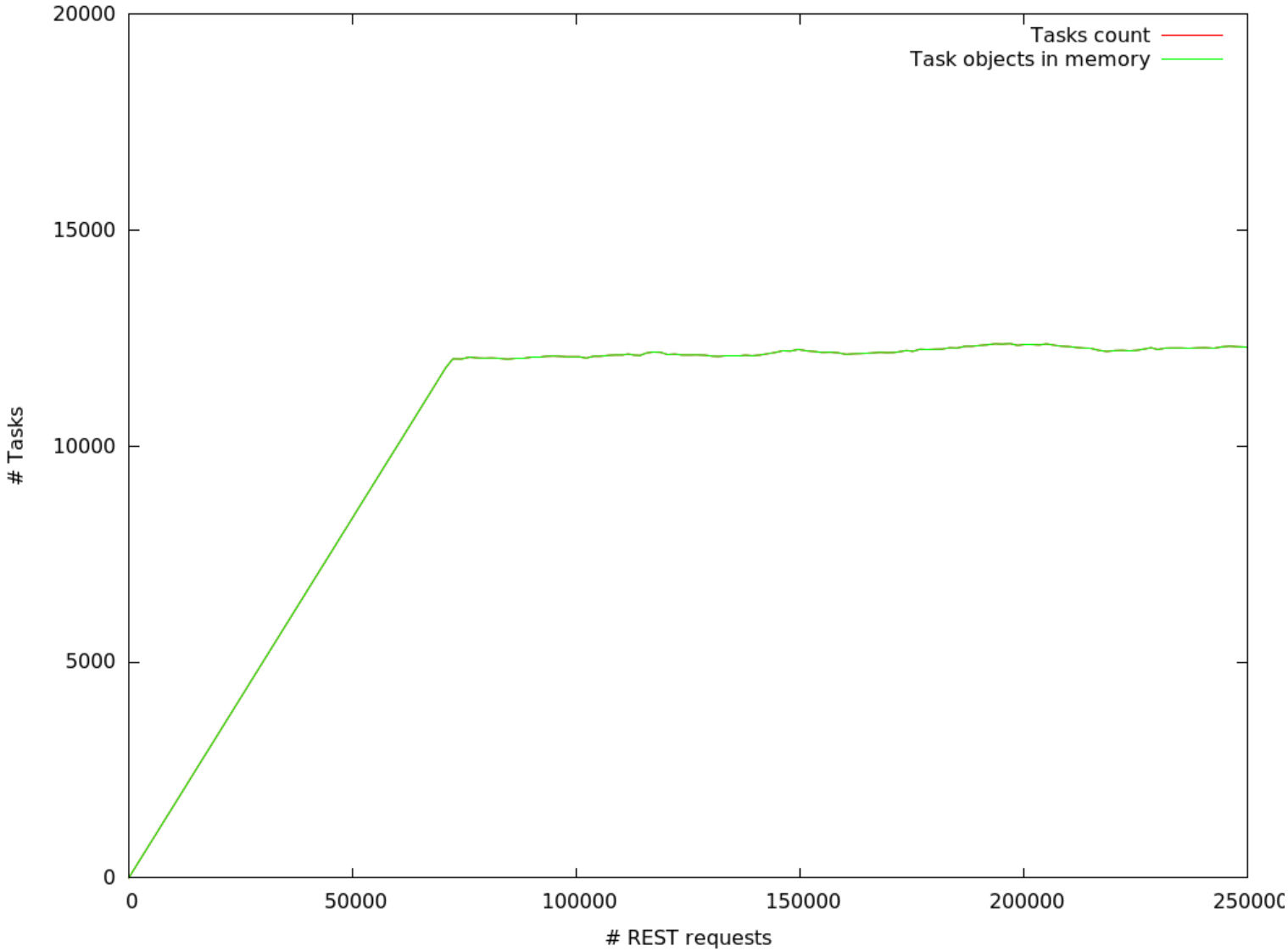
Wyniki


Tasks count
Task objects in memory
Tasks
REST requests
0
5000
10000
15000
20000
0
50000
100000
150000
200000
250000

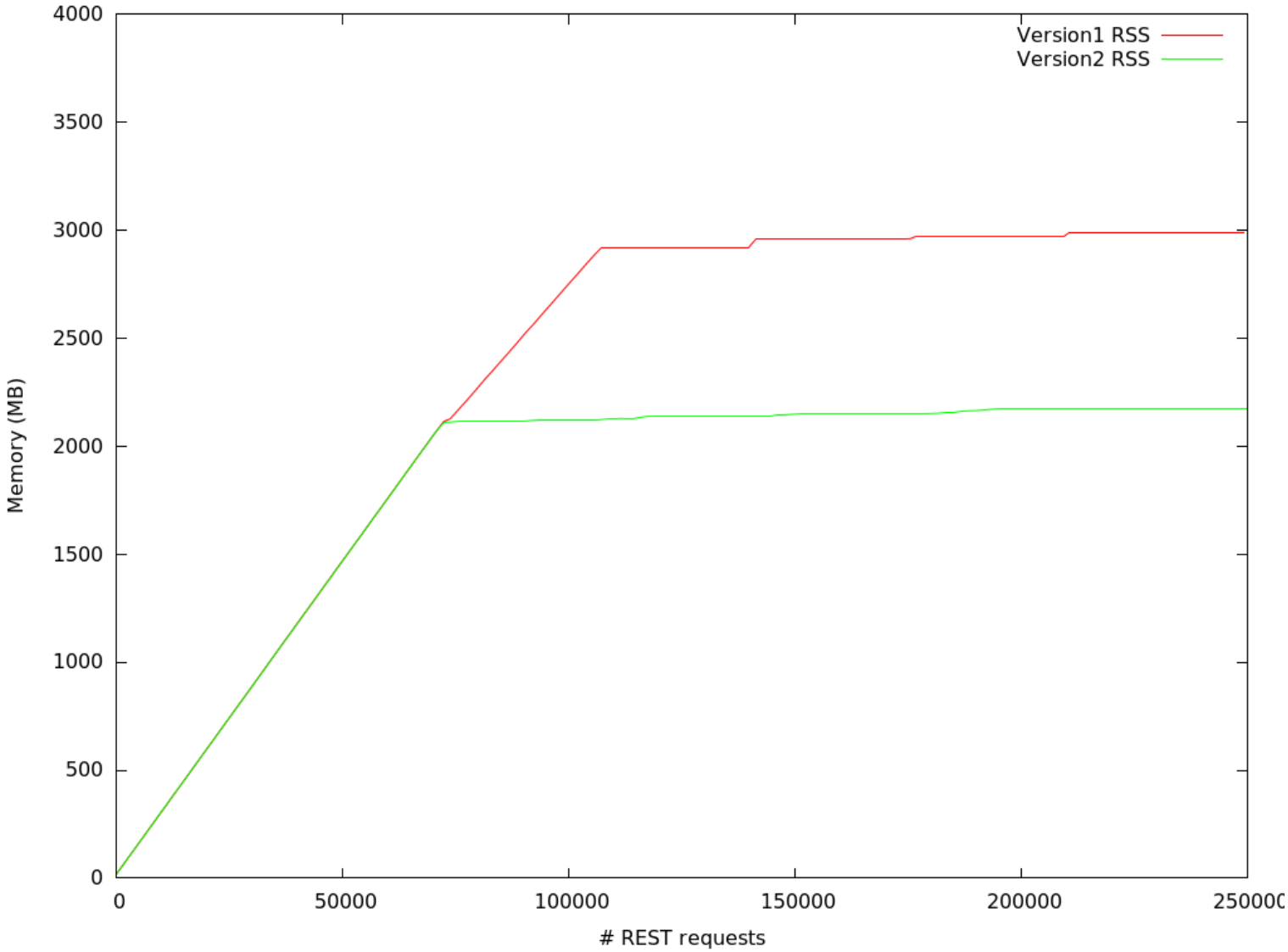
Subtask wersja 2

```
import weakref

class Subtask(object):
    def __init__(self, name, task):
        self.name = name
        self._task = weakref.ref(task)

    def get_task(self):
        return self._task()
```

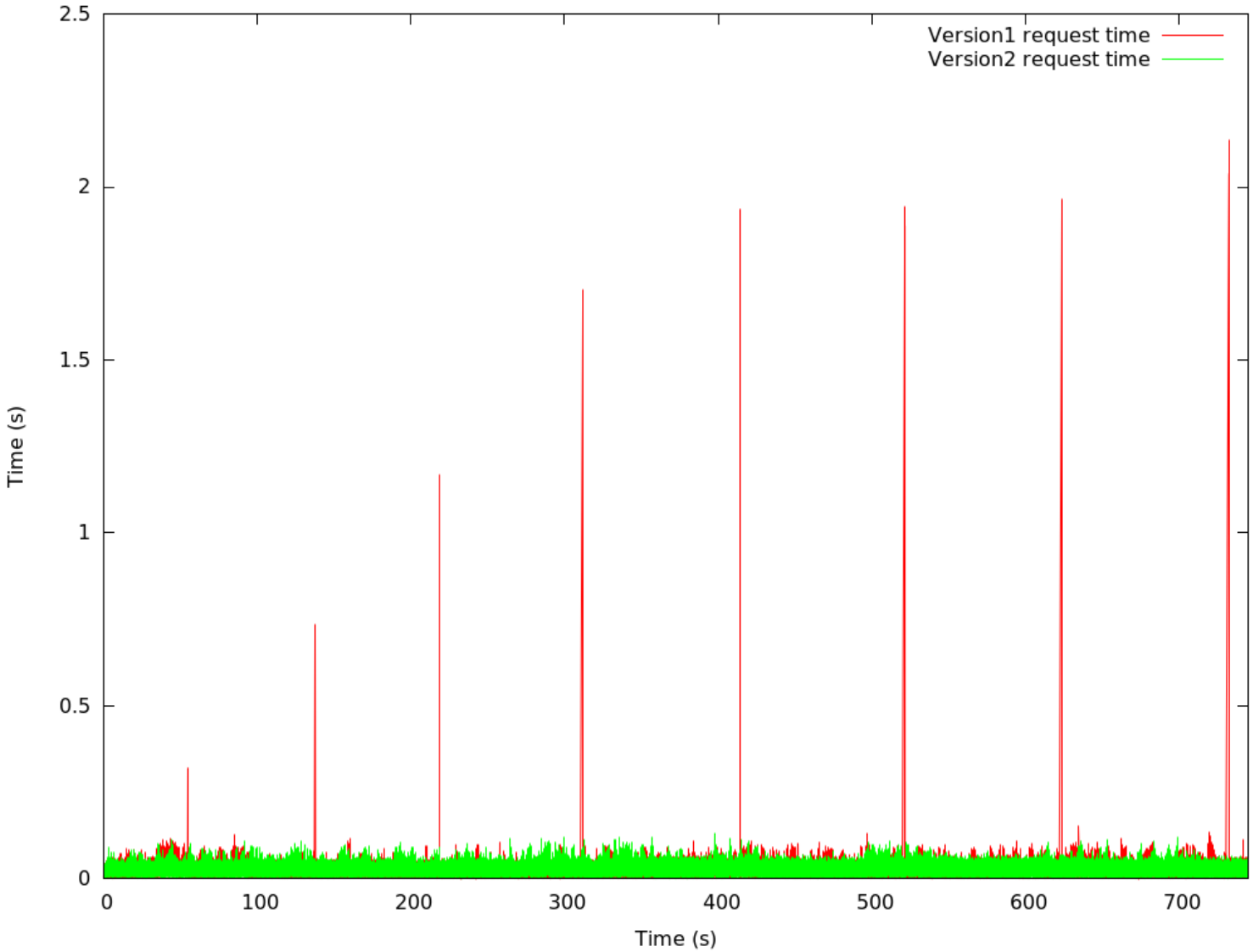
Efekty?


Tasks count
Task objects in memory
Tasks
REST requests
0
5000
10000
15000
20000
0
50000
100000
150000
200000
250000

Efekty cd.


Version1 RSS
Version2 RSS
Memory (MB)
REST requests
0
500
1000
1500
2000
2500
3000
3500
4000
0
50000
100000
150000
200000
250000

Efekty cd.


Version1 request time
Version2 request time
Time (s)
Time (s)
0
0.5
1
1.5
2
2.5
0
100
200
300
400
500
600
700

Wnioski

- Poleganie na GC jest kosztowne.
- W szczególności GC drugiej generacji trwa naprawdę długo (2-2.5s dla serwera z przykładu).
- Koszt wywołań GC niższych generacji jest wyraźnie niższy (nie widać wyraźnego wpływu na czas odpowiedzi serwera).
- GC drugiej jest uruchamiany bardzo rzadko.
- Python potrafi ponownie używać zwolnioną pamięć, ale nie wydaje się jej zwracać do systemu operacyjnego.

Zwalnianie pamięci

- Python ma własny alokator pamięci (szczegóły wychodzą poza zakres tego talka).
- Alokator potrafi zwracać pamięć do kernela*, ale:
  - Pamięć jest alokowana w blokach i będzie zwrócona tylko gdy cały blok jest wolny (fragmentacja!).
  - Dla wielu obiektów (np. tuple) zwalniane obiekty trafiają do puli, do ponownego użycia.
  - Pamięć raz zarezerwowana na inty nigdy nie będzie użyta na nic innego.
  - To samo dla float'ów.
  - ...

Ogólne wnioski

Jeśli mamy problemy ze zużyciem pamięci przez nasz program:

- Warto usuwać referencje do obiektów natychmiast, gdy przestaną być potrzebne.
- Warto unikać *zbędnych* cykli referencji przy pisaniu aplikacji.
- W większości pozostałych przypadków da się użyć słabych referencji.
- Albo explicite zrywać cykle referencji gdy obiekty przestaną być potrzebne (np. definiując i wołając metodę *delete*).
- *xrange* zamiast *range* w długich pętlach robi różnicę. Serio.

Punkt 2

Wycieki pamięci

Wycieki pamięci

Jest kilka możliwych przyczyn wycieku pamięci:

1. Wyciek pamięci w module napisanym w c/c++.
2. Pamięć przeznaczona na int'y, lub float'y nigdy nie będzie zwolniona, lub przeznaczona na cokolwiek innego.
3. Trzymanie "zapomnianych" referencji do struktur danych.
4. Garbage collector nie zwolni *żadnego* obiektu będącego częścią cyklu, w którym choć jeden obiekt definiuje finalizer (__*del*__).

”Zapomniane” referencje

Warto pamiętać o kilku nieoczywistych miejscach w których mogą pozostać referencje do naszych obiektów:

1. sys.exc_info() zwraca informacje o ostatnim wyjątku obsłużonym w obecnej ramce (frame) stosu. Jedną z informacji jest obiekt traceback zawierający cały stan stosu w momencie rzucenia wyjątku. Ups.
2. Domknięcia, functools.partial, itp.

Finalizer (__del__)

- Jeśli obiekt definiuje metodę __del__ zostanie ona wywołana tuż przed dealokacją obiektu.
- W __del__ obiekt może stworzyć referencję do siebie - zapobiegnie to jego usunięciu.
- **GC nigdy nie zdealokuje obiektu, który ma zdefiniowane __del__!**
- Innymi słowy cykl referencji w którym występuje obiekt ze zdefiniowanym __del__ nie zostanie nigdy zerwany.
- **Uwaga:** __del__ jest używany w wielu bibliotekach (a nawet w standardowej bibliotece Pythona!).

Finalizer c.d.

Jak widać finalizer tworzy problemy? Jak je rozwiązać?

- Nie używać. Context manager ("with") jest prawie zawsze lepszym rozwiązaniem.
- Jeśli potrzebujemy finalizera - upewnić się że obiekt, który definiuje finalizer nie jest w cyklu referencji (dekompozycja).
- W ostateczności: weakref.ref przyjmuje jako opcjonalny parametr callback, który zostanie wywołany po usunięciu obiektu. Problem: obiekt jest już usunięty w momencie wywołania callbacku, więc musimy gdzieś przechować stan potrzebny tej funkcji.

Finalizer przy użyciu weakref

```
class FileWrapper(object):
    _weakrefs = set()

    @classmethod
    def _delegated_close(cls, file_object, w):
        file_object.close()
        cls._weakrefs.remove(w)

    def __init__(self, name, mode):
        self._f = open(name, mode)
        self._weakrefs.add(weakref.ref(
            self,
            functools.partial(self._delegated_close,
                self._f)))
```

Część 3

Kilka przydatnych narzędzi

pdb

Debugger, część standardowej biblioteki Pythona. Pozwala poruszać się po stosie, wykonywać kod i ewaluować wyrażenia w kontekście dowolnej ramki, analizować wyjątki, ...

gc

Moduł ze standardowej biblioteki Pythona.

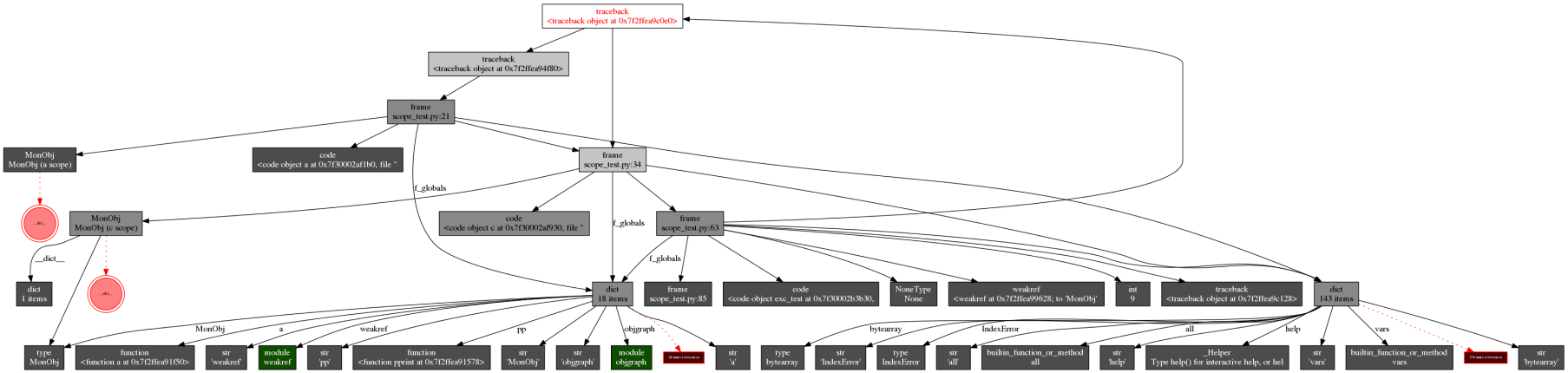
- **gc.collect(generation=2)** - ręczne wywołanie GC.
- **gc.enable()** / **gc.disable()** - włącz/wyłącz automatyczne GC.
- **gc.garbage** - lista wszystkich obiektów, które powinny zostać zwolnione, ale mają zdefiniowane __del__.
- Pozwala ustawić flagi dla garbage collectora, powodujące np. wypisanie statystyk po zakończeniu GC.
- Potrafi podać listę wszystkich obiektów trzymających referencję do naszego obiektu.

objgraph

Biblioteka do analizy obiektów w pamięci i szukania wycieków pamięci.

- Dostępna przez pip.
- Wizualizacja grafów referencji pomiędzy obiektami.
- Informacja o liczbie obiektów poszczególnych typów w pamięci.
- Zmiana ilości obiektów danego typu w czasie.
- Wystarcza do zdiagnozowania 99% problemów.

objgraph


traceback
<traceback object at 0x7f2ffea9c0e0>
traceback
<traceback object at 0x7f2ffea94f80>
frame
scope_test.py:21
MonObj
MonObj (a scope)
code
<code object a at 0x7f30002af1b0, file ''
frame
scope_test.py:34
f_globals
MonObj
MonObj (c scope)
code
<code object c at 0x7f30002af930, file ''
f_globals
frame
scope_test.py:63
__dict__
dict
1 items
dict
18 items
f_globals
frame
scope_test.py:85
code
<code object exc_test at 0x7f30002b3b30,
NoneType
None
weakref
<weakref at 0x7f2ffea99628; to 'MonObj'
int
9
traceback
<traceback object at 0x7f2ffea9c128>
dict
143 items
MonObj
a
weakref
pp
objgraph
bytearray
IndexError
all
help
vars
type
MonObj
function
<function a at 0x7f2ffea91f50>
str
'weakref'
module
weakref
str
'pp'
function
<function pprint at 0x7f2ffea91578>
str
'MonObj'
str
'objgraph'
module
objgraph
str
'a'
type
bytearray
str
'IndexError'
type
IndexError
str
'all'
builtin_function_or_method
all
str
'help'
_Helper
Type help() for interactive help, or hel
str
'vars'
builtin_function_or_method
vars
str
'bytearray'

guppy/heapy

- Narzędzie do profilowania i analizy problemów z pamięcią.
- Mnóstwo funkcji: analiza ścieżek w grafie zależności, drzewa rozpinające grafu referencji (!), grupowanie obiektów według różnych relacji równoważności (np. klasa, moduł z którego pochodzą), ...
- Da się dodać śledzenie własnych obiektów z modułu napisanego w C.
- Nienajlepsza dokumentacja i brak dobrych tutoriali w sieci.
- Na ten 1% sytuacji, gdy objgraph nie wystarczy :)

Część 4

Trochę więcej teorii, czyli jak to właściwie działa

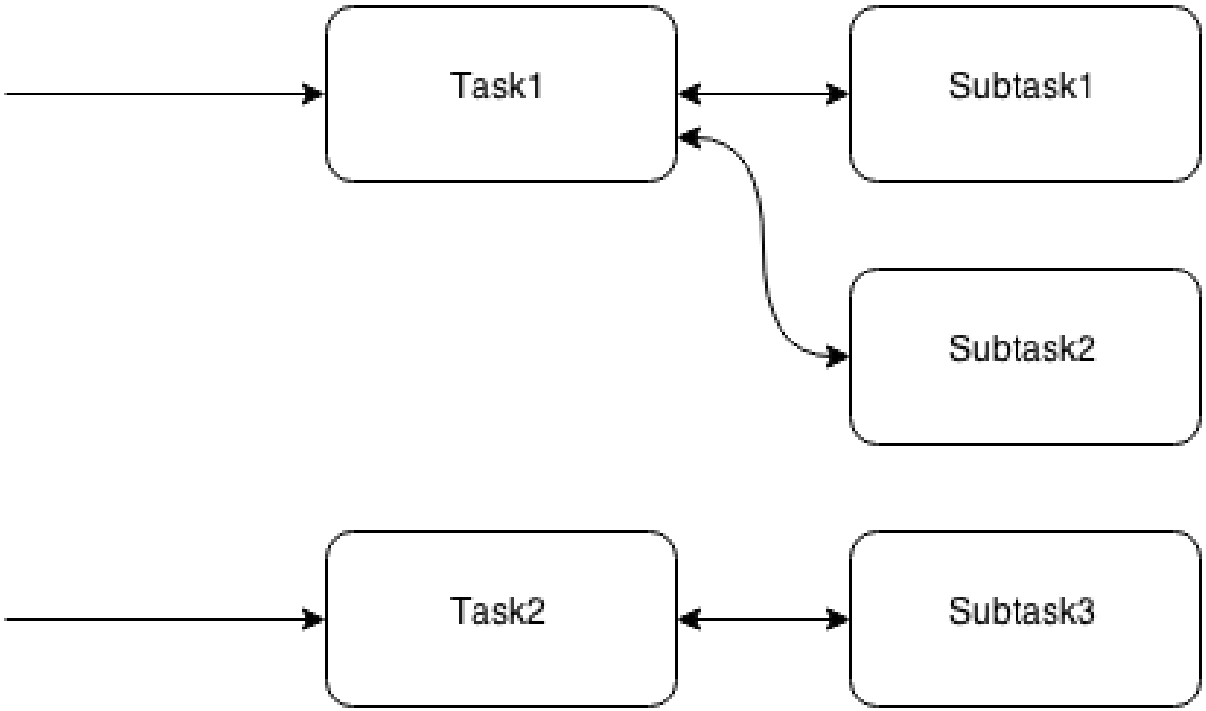
Generacje

- Każda generacja jest listą dwukierunkową.
- Nowo alokowany obiekt jest dodawany do generacji 0.
- Przed GC n-tej generacji listy obiektów niższych generacji są doklejane do listy obiektów n-tej generacji.
- Po GC n-tej generacji lista żywych obiektów jest doklejana do listy obiektów generacji n + 1.

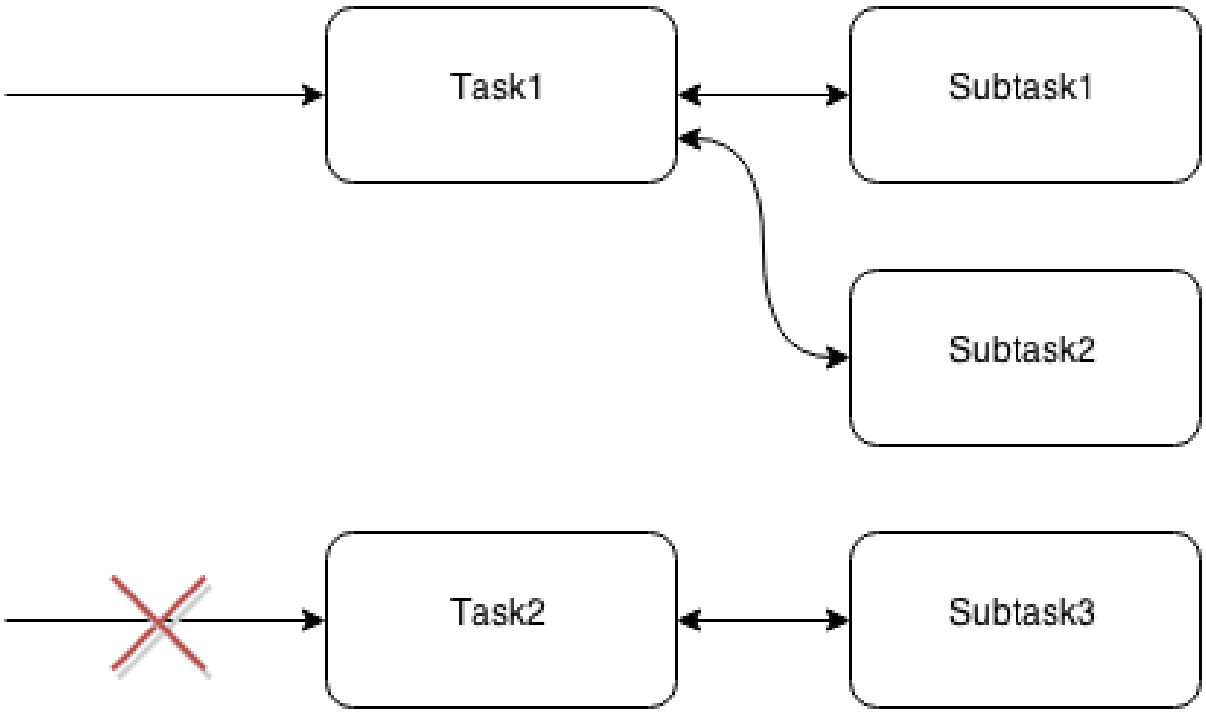
Algorytm GC

1. (Dla generacji 1 i 2) Dołącz obiekty niższych generacji (scalanie list dwukierunkowych).
2. Przeiteruj po obiektach na liście. Sprawdź wychodzące z nich referencje i jeśli prowadzą do innych obiektów na liście zmniejsz licznik referencji tych obiektów.
3. Przeiteruj po obiektach na liście. Jeśli ich licznik referencji wynosi 0 przenieś je na listę *unreacheable*.
4. Przeiteruj po obiektach pozostałych na liście. Sprawdź wychodzące z nich referencje i przenieś z powrotem każdy bezpośrednio osiągalny obiekt na liście *unreacheable*.
5. Przywróć oryginalne wartości liczników referencji.
6. Dołącz listę osiągalnych obiektów do listy obiektów generacji n + 1.
7. Usuń wszystkie referencje wychodzące z obiektów na liście *unreacheable*.

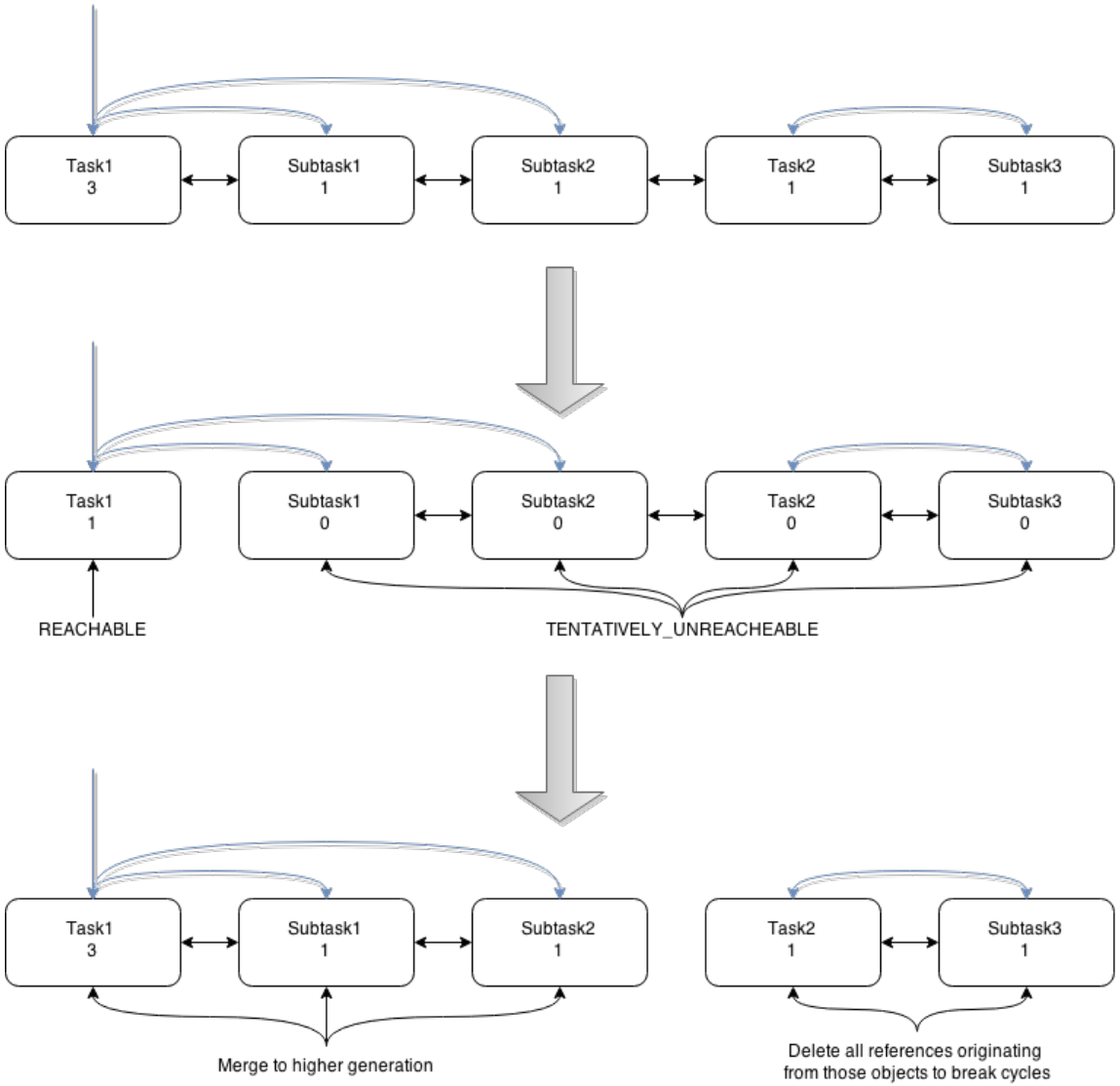
Przykład



Przykład



GC - wizualizacja



Uwagi

- Przedstawiany algorytm oddaję ogólną ideę działania GC w Pythonie. Rzeczywista implementacja jest trochę bardziej skomplikowana.
- Szczegóły w kodzie źródłowym Pythona, plik /Modules/gcmodule.c - bardzo dużo komentarzy.

Cytat na koniec

Programmers waste enormous amounts of time thinking about the speed of noncritical parts of their programs (...). We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

- Donald Knuth