Python上での Garbage Collectionの仕組み

2024.11.27 ゼミ　三上 陽向

卒論に向けた調査時に発生した疑問


```python
import gc

print("_____GC 0_____")
print(gc.collect())
print(gc.get_stats())

class C:
    pass

x1 = C()
x2 = C()
x1.next = x2
x2.next = x2

del x2

print("_____x2 deleted_____")
print(x1.next)
print(x1.next.next)

print("_____GC 1_____")
print(gc.collect())
print(gc.get_stats())


x1.next = 0
print("_____GC 2_____")
print(gc.collect())
print(gc.get_stats())
```

```
_____GC 0_____
0
[{'collections': 9, 'collected': 0, 'uncollectable': 0},
 {'collections': 0, 'collected': 0, 'uncollectable': 0},
 {'collections': 1, 'collected': 0, 'uncollectable': 0}]
_____x2 deleted_____
<__main__.C object at 0x7f0dc5d16470>
<__main__.C object at 0x7f0dc5d16470>
_____GC 1_____
0
[{'collections': 9, 'collected': 0, 'uncollectable': 0},
 {'collections': 0, 'collected': 0, 'uncollectable': 0},
 {'collections': 2, 'collected': 0, 'uncollectable': 0}]
_____GC 2_____
2
[{'collections': 9, 'collected': 0, 'uncollectable': 0},
 {'collections': 0, 'collected': 0, 'uncollectable': 0},
 {'collections': 3, 'collected': 2, 'uncollectable': 0}]
```

gc.collect() や gc.get_stats() の仕様は?
x1.next = 0 とした後に GC されたのは何?
なぜGCされた?
gc.collect() をしないと GC されないのか?

今回の内容

PythonでのGCの仕組みについて，今後の研究に向けた自分の理解＆備忘録として，今回のゼミで共有します．

1. Garbage Collection 概論
2. Python における GC
3. 冒頭の疑問への答え
4. 今後の方針

今回の内容

1. Garbage Collection 概論
2. Python における GC
3. 冒頭の疑問への答え
4. 今後の方針

1. Garbage Collection 概論

Dangling Reference

使用しなくなったメモリを解放(deallocate)するとき,
それを参照する別のポインタが残っていると,
解放されたセルに新しい値を格納したときに
型安全性が損なわれる可能性がある.

1. Garbage Collection 概論

このようなDangling Reference を回避するために，プログラム上でたどり着かないことが明らかなセルやそれを指すポインタをランタイムシステムが自動的に解放する Garbage Collection システムが多くの言語に実装されている.

[GCが実装されている言語]

ML, Haskel, APLなどの多くの関数型言語

Python, Java, JavaScript, Ruby, Lua, .NET言語　など

[GCが実装されていない言語]

C / C++	プログラマが明示的にメモリ管理 / デストラクタを使う
Rust	Ownershipシステムにより厳密にメモリ管理

https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

1. Garbage Collection 概論

GC の方法は大きく分けて3つ

1. Tracting GC　(Mark-sweep GC, Copying GC, etc)
2. Reference Counting
3. 静的自動メモリ管理 ※今回は触れません
(Escape Analysis, Region inference, etc)

[GC の改良]

+ Generational GC
+ Incremental GC

1.1. Tracting GC

“ヒープが満杯になるまでallocateを続け，満杯になったときにヒープ中のゴミオブジェクト (死んだオブジェクト) を一気に検出して解放する”

生死判定の概要

- ユーザプログラムが現在直接触れるメモリ領域 (レジスタ，スタック，大域変数など (= GCのルート)) からポインタによって指されるオブジェクトは生きている.

- 生きたオブジェクトからポインタによって指されるオブジェクトは生きている.

- それ以外のオブジェクトはゴミであり，解放されるべき.

1.1. Tracting GC

“Tracing GCの主な仕事は、ルートからポインタによって到達可能なオブジェクトを再帰的に探索し、たどられたオブジェクトを何らかの方法で「区別」することである。”

“探索には単純なallocate/free処理よりも
ずっと長い時間が必要”

“これが世の中で「GCは遅い」と言われるゆえんである”

http://matsu-www.is.titech.ac.jp/~endo/gc/gc.pdf

1.1. Tracting GC

1.1.1 Mark-sweep GC

各オブジェクトに1bitのマークビットを割り当てる

マークフェイズ(探索), スイープフェイズ(解放)の繰り返し

マークフェイズ

ルートオブジェクトから到達可能なオブジェクトの

マークビットを1にしていく

スイープフェイズ

すべてのオブジェクトのマークビットを調べ,

0であるオブジェクトを解放する

1.1. Tracting GC

1.1.2 Copying GC

“Copying GCアルゴリズムは、生きたオブジェクトをすきまをつめながら移動するというものである。

多くのSchemeやML処理系で採用されている。”

もっとも単純な例 ヒープを2等分

- 使用しているヒープが埋まったらGCを起動.
- from-space ヒープから to-space ヒープに到達可能なオブジェクトをコピー. GC 終了後は to-space ヒープで動作.
- GCのたびに2つのヒープの役割を交換.

1.2. Reference Counting

“それぞれのオブジェクトに対して Reference count を常に数えておく。オブジェクト a の参照数とは、世の中のどこか(ヒープ中の他のオブジェクト/スタック/大域変数など)から a を指しているポインタの本数である。”

“あるオブジェクト a の参照数が正のときは、a は将来使われるかもしれないのでとっておく。やがて a の参照数が0 になったら、そのオブジェクト a は今後の使い道がないと見なし、それを解放する。”

“この参照数を管理するために、コンパイラが協力するのが主流である。ポインタの増減が起りうる個所全てで、参照数をいじるコードを余分に出力するのである。なお、参照数の置き場所については、オブジェクトのヘッダに記録する処理系が多い。”

1.4. GCの改良

“Tracing GCの欠点は、一気に生きた全オブジェクトを探索するために停止時間が長くなることである。”

Incremental GC

“GCの探索処理を細切れにして、ユーザプログラムと交互に動かす。これにより一回あたりの停止時間を短くするアプローチであり、リアルタイム性の必要なプログラムに適している。”

“交互で動かす最もメジャーな方法は、allocate処理を行なうときに、こっそり少しづつGCを進めるというものである。”

1.4. GCの改良

Generational GC

オブジェクトの寿命に関する傾向を利用

> “古くから生きているオブジェクトはそのまま生き残りやすく、
> 新しく確保されたオブジェクトほどすぐにゴミになりやすい”

1.4. GCの改良

Generational GC

“オブジェクトはまず新世代ヒープに確保される”

“新世代ヒープが一杯になったら、新世代ヒープだけを対象としたGC(minor collection)を行なう。”

“ Minor collectionを繰り返した結果、ある程度以上長寿命なオブジェクトが見つかったなら、そのオブジェクトは旧世代ヒープに移される”

“やがて旧世代ヒープが一杯になったら、そこではじめて、ヒープ全体を探索する GC (major collection) を試みる。”

http://matsu-www.is.titech.ac.jp/~endo/gc/gc.pdf

今回の内容

1. Garbage Collection 概論
2. Python における GC
3. 冒頭の疑問への答え
4. 今後の方針

2. Python における GC

CPython (C言語により処理系が構築されているPython)

・・・ Reference counting 方式 (通常)

2世代 Generational GC

(循環参照の処理 : GC モジュールで実装)

```
>>> import sys
>>> x = object()
>>> sys.getrefcount(x)
2
>>> y = x
>>> sys.getrefcount(x)
3
>>> del y
>>> sys.getrefcount(x)
2
```

Pythonでは循環参照はどこでも起こり得る:
インタプリタが必要とする多くの内部参照が循環参照を生成するため.

→循環参照に対応するためのアルゴリズムがgcモジュールに組み込まれている.

https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

2. Python における GC

2世代 Generational GC : 循環参照の処理のために実装

若い世代には短命のオブジェクトを多く含むという発想から
若い世代の GC を重点的に実施することで効率化を図る

GC が実行されるタイミングは threshold により決まる.
これは各世代のオブジェクトのサイズにより変動する (後述).
ユーザーが設定することもできる.

gc.get_threshold()

現在の検出閾値を、`(threshold0, threshold1, threshold2)` のタプルで返します。

```
>>> gc.get_threshold()
(700, 10, 10)
```

https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection
https://docs.python.org/ja/3/library/gc.html

2.1. 2世代Generational GC

***gen0* (若い世代)**

新しく割り当てられたオブジェクトはすべてgen0に配置

allocation – deallocation > threshold0 のとき GC 実行

***gen1* (中年世代)**

gen1 は Python 3.13 (2024.10.07) より削除された.

threshold2 も無視される.

***gen2* (古い世代)**

gen1 の GC が threshold1 回実行されるたびに

1 / threshold1 の割合でスキャン (Incremental Correction)

☆ gen0 の GC を生き残ったオブジェクトは gen2 へ昇格

https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection
https://docs.python.org/3.14/library/gc.html

2.1. 2世代Generational GC

Incremental collection

若い世代の スキャン時間はスキャンサイズを変更することにより制御可能だが，古い世代はサイズが大きく，その分スキャン時間が大きくなってしまう．これに対処するため，incremental collection が導入されている．

到達不可能なオブジェクトを検出し回収するためには，GC 時にヒープ全体を検査 (full scavenge) しなければならない．incremental collection はヒープ全体を何度かに分けてスキャンするもので，full scavenge は 一連の increments により実行される．

[各increment 単位の構成]

- 若い世代のオブジェクト
- 古い世代のうち最近スキャンされていないオブジェクト
- これらの未スキャンオブジェクトから到達可能なオブジェクト

https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

2.1. 2世代Generational GC

Incremental collection

何がスキャンされているか追跡し続けるため，古い世代は2つのリストを持つ.

1. pending

    スキャンされていないオブジェクトのリスト. full scavenge 開始時は古い世代のすべてのオブジェクトが pending space にいるはず.

2. visited

    スキャンされ，生き残ったオブジェクトのリスト.

ヒープのすべてのオブジェクトの走査が完了したら，visited リストのすべての要素を pending リストに移さなければならない.

すべての要素を移す代わりに，2つのリストの役割を1サイクルごとに交換する操作を GCState 構造体のフィールドを使用して行っている.

https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

2.1. 2世代Generational GC

Incremental collection

GC をいつ実行するか判定するために，コレクタは常に最後の collection 以降の オブジェクトの allocation，deallocation の数を 追跡しており, allocation – deallocation > threshold0 となったときにGC を実行する.

threshold1 は1回の incremental scan で処理される割合を制御する役割を持っている．すなわち，threshold1 の値が大きければ大きいほど，古い世代の GC が実行される頻度が低くなる.

https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

2.1. 2世代Generational GC

Incremental collection

```
>>> gc.get_threshold()
(700, 10, 10)
```

この例の場合,

threshold0 = 700

threshold1 = 10

threshold2 = 10

であるから,

allocation – deallocation > 700 となるたびにGC が実行され,

gen0のGCが10回行われるたびに gen2の1/10 がscanされる.

したがって, 各世代別フルスキャンの回数の割合は,

gen0 ：gen2 = 100 : 1

と考えられる.

https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

2.2. 循環参照への対処

Pythonでは循環参照はどこでも起こり得る:

インタプリタが必要とする多くの内部参照が循環参照を生成するため.

→循環参照に対応するためのアルゴリズムが gcモジュールに組み込まれている.

これは，古い世代の GC (incremental collection) 実行時に自動的に行われるもので，Mark–sweep に似たアルゴリズムが部分的に実行される.

具体的な手順を次に説明する.

https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

2.2. 循環参照への対処

2.2.1. 循環参照の特定とGC

このアルゴリズムは，コンテナオブジェクトを処理する．
(1つ以上のオブジェクトへの参照を持つ可能性があるため)

コンテナオブジェクト

配列，辞書，リスト，カスタムクラスへのインスタンス，
拡張モジュール内のクラス など．

これらのコンテナオブジェクトについて，到達可能なのか
そうでないのかを厳密に区別する必要がある．

https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

2.2. 循環参照への対処

2.2.1. 循環参照の特定と GC

例として，次のようなプログラムを考える.

```
>>> import gc

>>> class Link:
...     def __init__(self, next_link=None):
...         self.next_link = next_link

>>> link_3 = Link()
>>> link_2 = Link(link_3)
>>> link_1 = Link(link_2)
>>> link_3.next_link = link_1
>>> A = link_1
>>> del link_1, link_2, link_3

>>> link_4 = Link()
>>> link_4.next_link = link_4
>>> del link_4

# Collect the unreachable Link object (and its .__dict__ dict).
>>> gc.collect()
2
```

https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

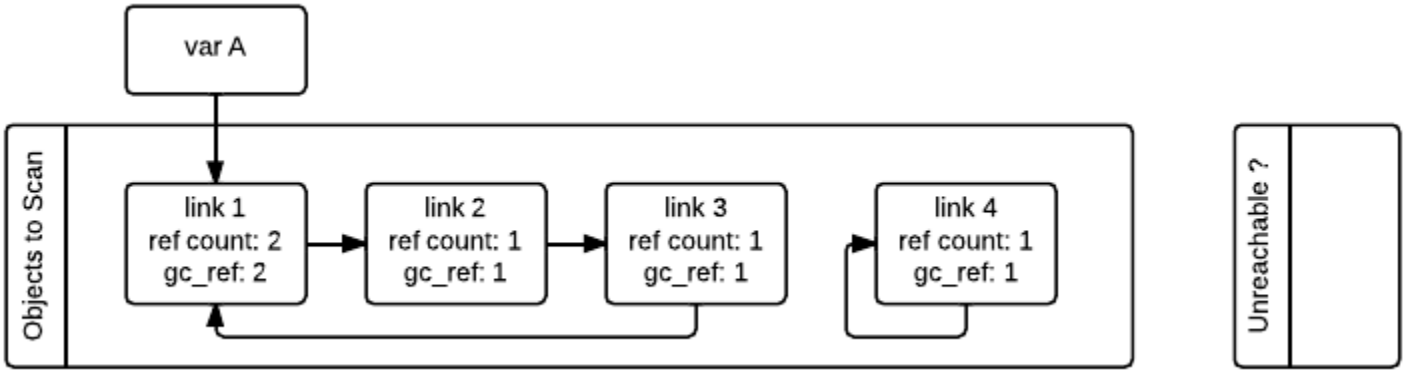
2.2. 循環参照への対処

2.2.1. 循環参照の特定と GC

デフォルトビルドでは，このアルゴリズムのスキャン対象はすべてのコンテナオブジェクトの集合またはそのサブセットである (incremental collection の場合と考えられる) .

アルゴリズムの開始時，それぞれのオブジェクトは追加の参照カウントフィールド `gc_ref` を持つ.

これは `ref count` により初期化されている.



https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

2.2. 循環参照への対処

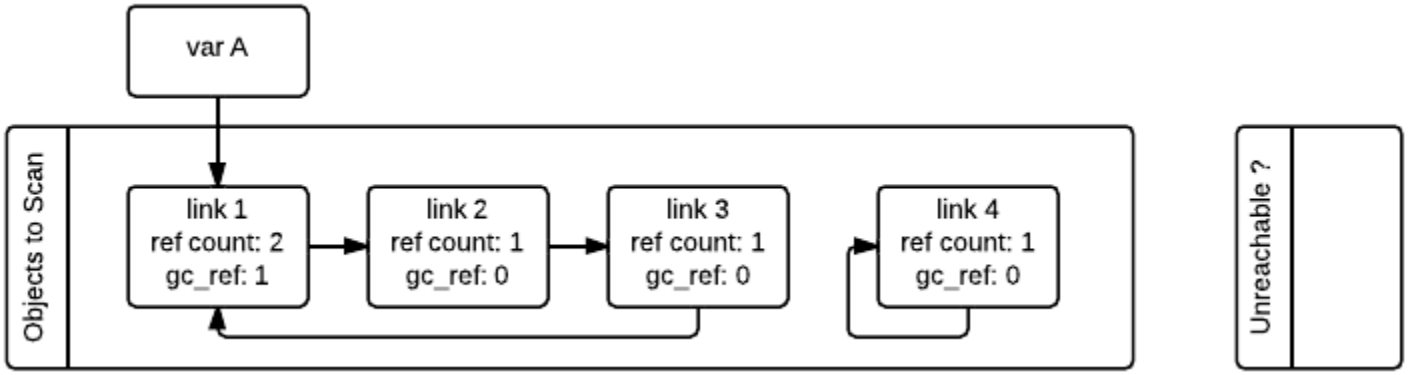
2.2.1. 循環参照の特定と GC

それぞれのコンテナが参照する先の `gc_ref` を1減らす.

すべてのコンテナの処理が終わると，外部から参照されているオブジェクトのみが `gc_ref > 0` となる.

☆必ずしも `gc_ref == 0` ⇒ unreachable ではない

例えば下の図では, link_2 は link_1 により到達可能.



https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

2.2. 循環参照への対処

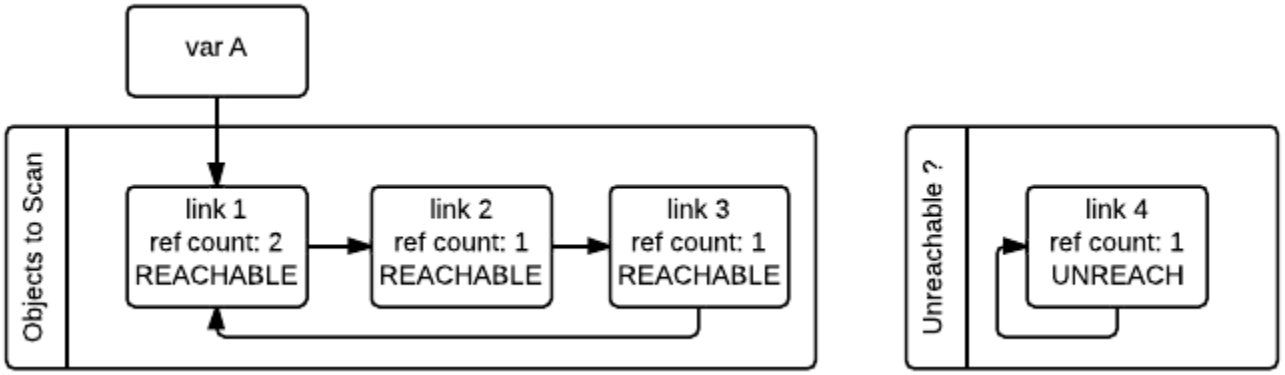
2.2.1. 循環参照の特定と GC

`gc_ref == 0` であるオブジェクトは”暫定的に到達不能”と判定され，別のリストに移動される．

次に，`gc_ref > 0` のオブジェクトを再度走査し，そこから到達可能なオブジェクトを元のリストに復活させる．

復活したオブジェクトの参照も当然走査する．

この走査をしても復活しないオブジェクトこそが真に到達不能として GC によって破棄される．



https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

2.2. 循環参照への対処

2.2.2. アルゴリズムの特徴

一度到達可能と判定されたオブジェクトはgc_ref == 1 に設定される.

GC がこのオブジェクトを何度も走査することがないよう,
走査済みオブジェクトにはフラグがつけられ，区別される.

このアルゴリズムは再帰を必要とせず，追加のメモリを準備する必要もない.

内部のCが必要とするO(1)のストレージを除き，アルゴリズムが必要とする追加のストレージはすべてオブジェクト自身が持っている.

https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

2.2. 循環参照への対処

2.2.3. 走査の”正しさ”

Full – scavenge により到達不能なサイクルが見つかることを保証するには，すべての各サイクルは単一インクリメント内に収まっている必要がある.

部分的なサイクルがインクリメント内に残らないようにするため，最初のインクリメントから到達可能で未スキャンのオブジェクトに対し transitive closure を実行する (?) .

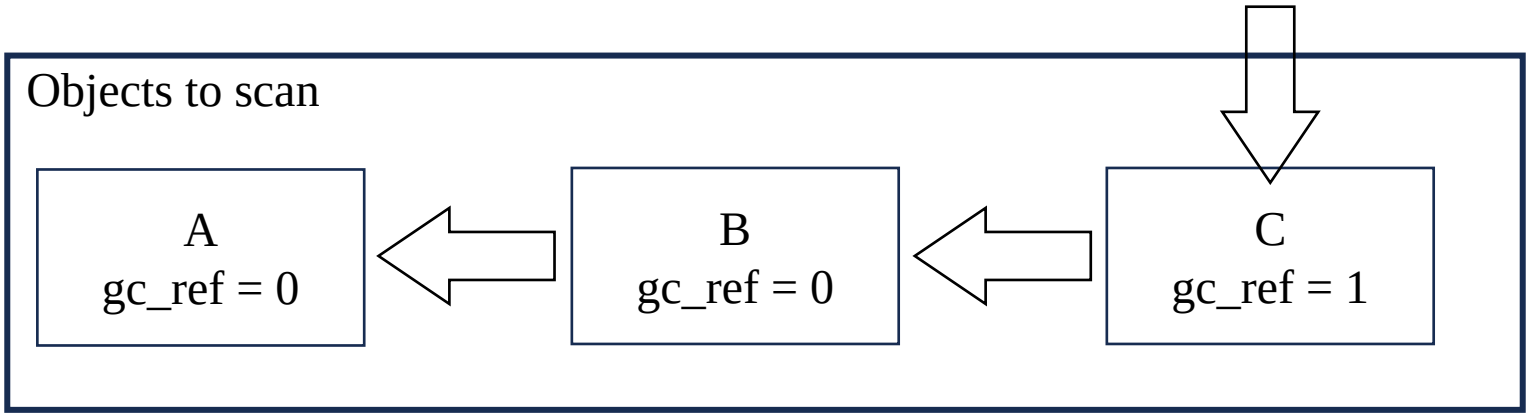
（おそらくある未スキャンのオブジェクトから到達可能なすべてのオブジェクトに対してスキャンを実行することで，サイクルの参照があったとしてもすべてたどることができる という意味だと思われる）

https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

2.2. 循環参照への対処

2.2.4. オブジェクトを移動することが効果的な理由

例えば，オブジェクトA, B, C をこの順で作成し，Cは外部から参照可能とする．また，B は A を，C は B を参照する．

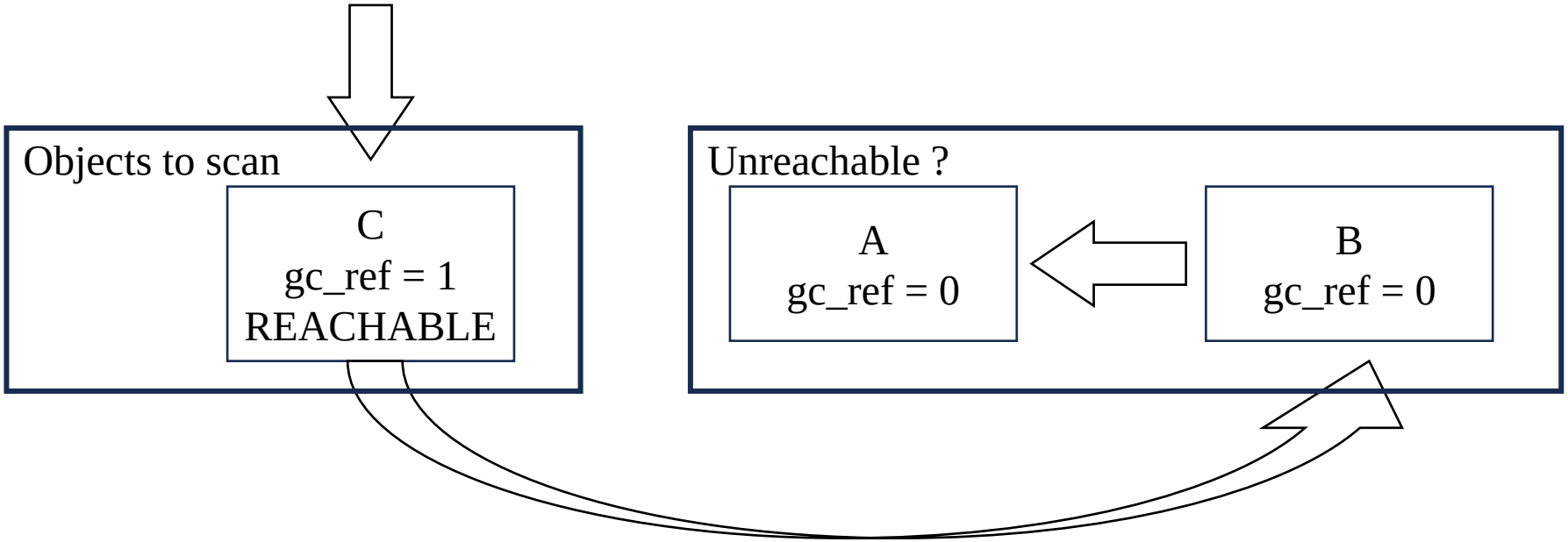


https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

2.2. 循環参照への対処

2.2.4. オブジェクトを移動することが効果的な理由

A, B は”暫定的に到達不能”リストへ移動される

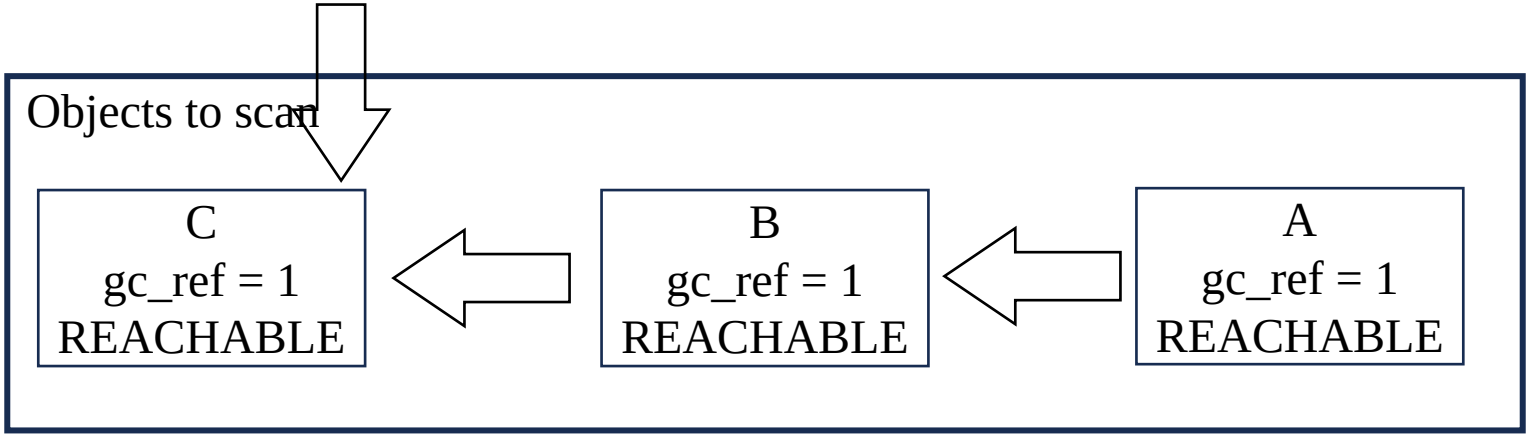


https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

2.2. 循環参照への対処

2.2.4. オブジェクトを移動することが効果的な理由

再度 C を走査をするとオブジェクトは復活し，C, B, A の順に並ぶ



A, B を2度移動することは一見コストの無駄のように見えるが，このC, B, A の順番は，後続の走査で変更されることはなく，今後走査を行うことにより発生しうる無限回の移動を省略できる可能性がある.

https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md#Optimization-incremental-collection

今回の内容

1. Garbage Collection 概論
2. Python における GC
3. 冒頭の疑問への答え
4. 今後の方針

卒論に向けた調査時に発生した疑問


```python
import gc

print("_____GC 0_____")
print(gc.collect())
print(gc.get_stats())

class C:
    pass

x1 = C()
x2 = C()
x1.next = x2
x2.next = x2

del x2

print("_____x2 deleted_____")
print(x1.next)
print(x1.next.next)

print("_____GC 1_____")
print(gc.collect())
print(gc.get_stats())


x1.next = 0
print("_____GC 2_____")
print(gc.collect())
print(gc.get_stats())
```

```
_____GC 0_____
0
[{'collections': 9, 'collected': 0, 'uncollectable': 0},
 {'collections': 0, 'collected': 0, 'uncollectable': 0},
 {'collections': 1, 'collected': 0, 'uncollectable': 0}]
_____x2 deleted_____
<__main__.C object at 0x7f0dc5d16470>
<__main__.C object at 0x7f0dc5d16470>
_____GC 1_____
0
[{'collections': 9, 'collected': 0, 'uncollectable': 0},
 {'collections': 0, 'collected': 0, 'uncollectable': 0},
 {'collections': 2, 'collected': 0, 'uncollectable': 0}]
_____GC 2_____
2
[{'collections': 9, 'collected': 0, 'uncollectable': 0},
 {'collections': 0, 'collected': 0, 'uncollectable': 0},
 {'collections': 3, 'collected': 2, 'uncollectable': 0}]
```

gc.collect() や gc.get_stats() の仕様は?
x1.next = 0 とした後に GC されたのは何?
なぜGCされた?
gc.collect() をしないと GC されないのか?

3. 冒頭の疑問への答え


```python
import gc

print("_____GC 0_____")
print(gc.collect())
print(gc.get_stats())

class C:
    pass

x1 = C()
x2 = C()
x1.next = x2
x2.next = x2

del x2

print("_____x2 deleted_____")
print(x1.next)
print(x1.next.next)

print("_____GC 1_____")
print(gc.collect())
print(gc.get_stats())


x1.next = 0
print("_____GC 2_____")
print(gc.collect())
print(gc.get_stats())
```

Reference Conuting方式であるからGCは随時実行される. しかし，循環参照はいつでも起こり得るため，メモリリークを防ぐためにincremental scanを用いた2世代 GC が実装されている.

gc.collect() は Full-scavenge を即時実行する.
また，gc.get_stats() は 各世代の gc の状況を出力する.

https://docs.python.org/3.13/library/gc.html

3. 冒頭の疑問への答え


```python
import gc

print("_____GC 0_____")
print(gc.collect())
print(gc.get_stats())

class C:
    pass

x1 = C()
x2 = C()
x1.next = x2
x2.next = x2

del x2

print("_____x2 deleted_____")
print(x1.next)
print(x1.next.next)

print("_____GC 1_____")
print(gc.collect())
print(gc.get_stats())


x1.next = 0
print("_____GC 2_____")
print(gc.collect())
print(gc.get_stats())
```

```
_____x2 deleted_____
<__main__.C object at 0x7f0dc5d16470>
<__main__.C object at 0x7f0dc5d16470>
_____GC 1_____
0
[{'collections': 9, 'collected': 0, 'uncollectable': 0},
 {'collections': 0, 'collected': 0, 'uncollectable': 0},
 {'collections': 2, 'collected': 0, 'uncollectable': 0}]
```

x2 がdel されても x1.next や x1.next.next のように x2 自身への参照は残っており，オブジェクトは解放されない．

参照が残っている以上，GCも行われない．

3. 冒頭の疑問への答え


```python
import gc

print("_____GC 0_____")
print(gc.collect())
print(gc.get_stats())

class C:
    pass

x1 = C()
x2 = C()
x1.next = x2
x2.next = x2

del x2

print("_____x2 deleted_____")
print(x1.next)
print(x1.next.next)

print("_____GC 1_____")
print(gc.collect())
print(gc.get_stats())


x1.next = 0
print("_____GC 2_____")
print(gc.collect())
print(gc.get_stats())
```

```
_____GC 2_____
2
[{'collections': 9, 'collected': 0, 'uncollectable': 0},
 {'collections': 0, 'collected': 0, 'uncollectable': 0},
 {'collections': 3, 'collected': 2, 'uncollectable': 0}]
```

x1.next = 0 とすると,

x1→x2

の参照が切れ,

x2 →x2

の参照が残る. かつ x2は外部から到達不可.
これはPythonの GC によりキャッチされる対象であり, gc.collect() を実行することにより GC された.

“collected” が 2 となっているのは, オブジェクトx2と, それ自身の .__dict__ であると考えられる.

```python
print(x1.next.next.__dict__)
```

```
{'next': <__main__.C object at 0x7fe25aa96470>}
```

3. 冒頭の疑問への答え

gc.collect() を実行しなくても x2 は解放されるのか?

```python
import gc

gc.set_debug(gc.DEBUG_STATS)

class C:
    def __init__(self, name):
        self.name = name

    def __del__(self):
        # オブジェクトが削除される際に通知
        print(f"[INFO] {self.name} is being deleted")

x1 = C("x1")
x2 = C("x2")
x1.next = x2
x2.next = x2

del x2

x1.next = 0

print("[INFO] The object x1 is:", x1)
print("[INFO] The object x1.next is:", x1.next)
```

3. 冒頭の疑問への答え

たしかにx2は GC されているが，リアルタイムに GC データを出力するコードが書けず…

```
[INFO] The object x1 is: <__main__.C object at 0x7fb38250ffd0>
[INFO] The object x1.next is: 0
gc: collecting generation 2...
gc: objects in each generation: 35 4746 0
gc: objects in permanent generation: 0
[INFO] x2 is being deleted
gc: done, 2 unreachable, 0 uncollectable, 0.0003s elapsed
gc: collecting generation 2...
gc: objects in each generation: 70 0 4684
gc: objects in permanent generation: 0
[INFO] x1 is being deleted
gc: done, 181 unreachable, 0 uncollectable, 0.0003s elapsed
gc: collecting generation 2...
gc: objects in each generation: 0 0 3889
gc: objects in permanent generation: 0
gc: done, 1076 unreachable, 0 uncollectable, 0.0003s elapsed
gc: collecting generation 2...
gc: objects in each generation: 0 0 2667
gc: objects in permanent generation: 0
gc: done, 187 unreachable, 0 uncollectable, 0.0001s elapsed
```

わかりやすくするため，GC 1回当たりの情報を白枠で囲っています．

どうやら4回 gen2 の GC が実行され，1回目で x2 が削除され，2回目で x1 が削除されたらしい…？　なぜ…？

今回の内容

1. Garbage Collection 概論
2. Python における GC
3. 冒頭の疑問への答え
4. 今後の方針

4. 今後の方針

・Python でメモリリークが発生するような例について,
　　静的メモリ管理を行う Rust で書くとするとどうなるか
　　メモリ効率や実行速度など何か違いは生じるか
などの対照実験を行う.

今後のアイデアの一つとして

・Python の GC の仕組みを Coq 等で定式化する

参考文献

Garbage collection (computer science) - Wikipedia

https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

一般教養としてのGarbage Collection (コンパイラ演習番外編)

東京工業大学 総合研究院 スーパーコンピューティング研究センター　遠藤 敏夫

http://matsu-www.is.titech.ac.jp/~endo/gc/gc.pdf

Garbage collector design (CPython Internals Documentation)

https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md

参考文献

gc — ガベージコレクターインターフェース

https://docs.python.org/ja/3/library/gc.html

gc — Garbage Collector interface (Python 3.13.0)

https://docs.python.org/3.13/library/gc.html

gc — Garbage Collector interface (Python 3.14.0a2)

https://docs.python.org/3.14/library/gc.html