Exercise 4

Table of Contents

1. Garbage collection 2
2. Sys & Garbage Collector Module 2
3. Memory Profiling Monte Carlo Walk Analysis 3
    3.1. Quellcode 4

1. Garbage collection

Python besitzt eine automatische Speicherfreigabe. Nicht mehr referenzierte Objekte werden vom Speicher freigegeben. Jedes Objekt besitzt einen Typ, Wert und einen Referenz-Zähler. Beim Referenz-Zähler wird dabei angegeben, wie oft das Objekt von Namen referenziert wird.

```
a = 1 # refs: 1
b = a # refs: 2

a = None # refs: 1
b = None # refs: 0
```

Wenn der Referenz-Zähler 0 erreicht, würde das Objekt gelöscht werden. In diesem Fall wäre dies, nachdem b auf None gesetzt wurde. Diese Variante mit dem Mitzählen von Referenzen funktioniert zwar ganz gut, es kann aber zu zyklischen Referenzen kommen. Wenn das Objekt eine Referenz auf sich selbst bezieht, würde diese Variante nicht funktionieren. Ein einfaches Beispiel wäre eine Referenz auf sich selbst, indem eine Liste in sich selbst eingefügt wird.

```
a = []
a.append(a)
print(a)
```

Das Objekt wird dabei nie gelöscht, da der Referenz-Zähler nie 0 erreicht. Für dieses Problem gibt es eine zweite Art der Garbage Collection, die sogenannte Generational Garbage Collection.

Dabei werden bei einem Programmlauf drei Listen erstellt: Generation 0, 1 and 2. Neu erstellte Objekte werden in die Generation 0 Liste eingefügt. Zyklische Abhängigkeiten werden erkannt, wenn ein Objekt keine Referenzen von außerhalb besitzt wird es entfernt. Objekte, welche überbleiben, landen in der Generation 1 Liste. Die gleichen Schritte wie auch bei der Generation 0 Liste werden ausgeführt. Überlebende Objekte dieser Liste landen danach in der Generation 2 Liste. Diese Objekte bleiben für den ganzen Programmlauf bestehen.

2. Sys & Garbage Collector Module

Mit sys.getrefcount kann die Anzahl der Referenzen zurückgegeben werden. Der kleinstmögliche Wert beträgt dabei drei, da zwei Referenzen von der Funktion erstellt

werden und einer durch die Objekterstellung vorhanden ist.

```
import sys

print(sys.getrefcount(1000)) # refs: 3
a = 1000
print(sys.getrefcount(a)) # refs: 4
a = None
print(sys.getrefcount(1000)) # refs: 3
```

Mit dem gc-Modul kann das Verhalten des Garbage-Collectors verändert werden.

```
import gc

gc.set_threshold(10, 10, 1)
gc.set_threshold(0, 0, 0)
collected_count = gc.collect()
collected_count = gc.collect(0)
collected_count = gc.collect(1)
collected_count = gc.collect(2)
```

Mittels set_treshold kann festgelegt werden, ab wann die Garbage-Collection für die einzelnen Listen erfolgen soll. Wenn set_treshold auf 0 gesetzt wird, wird die Garbage-Collection deaktiviert. Mit gc.collect() kann diese manuell für alle oder für einzelne Listen angestoßen werden.

3. Memory Profiling Monte Carlo Walk Analysis

Die Variante mit dem deaktivierten gen_walk Flag weißt dabei einen wesentlich geringeren Memory Peak auf als die originale Variante ohne Flag.

```
Memory peak (monte_carlo_walk_analysis): 46826264
Memory peak (monte_carlo_walk): 12914112
Allocations:
c:\Users\Andi\Documents\GitHub\fh-se-python\exercise\Ue04\src\Memory.py:89: size=110 KiB (+110
KiB), count=1996 (+1996), average=56 B
c:\Users\Andi\Documents\GitHub\fh-se-python\exercise\Ue04\src\Memory.py:87: size=0 B (-110 KiB),
count=0 (-1996)
c:\Users\Andi\Documents\GitHub\fh-se-python\exercise\Ue04\src\Memory.py:132: size=1728 B (+1728
B), count=22 (+22), average=79 B
c:\Users\Andi\Documents\GitHub\fh-se-python\exercise\Ue04\src\Memory.py:81: size=3192 B (-1176
B), count=57 (-21), average=56 B
C:\Users\Andi\AppData\Local\Programs\Python\Python310\lib\tracemalloc.py:558: size=952 B (+896
B), count=17 (+16), average=56 B
C:\Users\Andi\AppData\Local\Programs\Python\Python310\lib\tracemalloc.py:423: size=568 B (+568
B), count=4 (+4), average=142 B
C:\Users\Andi\AppData\Local\Programs\Python\Python310\lib\tracemalloc.py:560: size=464 B (+464
B), count=2 (+2), average=232 B
c:\Users\Andi\Documents\GitHub\fh-se-python\exercise\Ue04\src\Memory.py:144: size=440 B (+440
B), count=1 (+1), average=440 B
c:\Users\Andi\Documents\GitHub\fh-se-python\exercise\Ue04\src\Memory.py:147: size=416 B (+416
B), count=1 (+1), average=416 B
c:\Users\Andi\Documents\GitHub\fh-se-python\exercise\Ue04\src\Memory.py:141: size=0 B (-416 B),
count=0 (-1)
c:\Users\Andi\Documents\GitHub\fh-se-python\exercise\Ue04\src\Memory.py:66: size=710 B (+168 B),
count=7 (+3), average=101 B
c:\Users\Andi\Documents\GitHub\fh-se-python\exercise\Ue04\src\Memory.py:145: size=112 B (+112
B), count=3 (+3), average=37 B
c:\Users\Andi\Documents\GitHub\fh-se-python\exercise\Ue04\src\Memory.py:51: size=56 B (+56 B),
count=1 (+1), average=56 B
C:\Users\Andi\AppData\Local\Programs\Python\Python310\lib\tracemalloc.py:313: size=48 B (+48 B),
count=1 (+1), average=48 B
C:\Users\Andi\AppData\Local\Programs\Python\Python310\lib\tracemalloc.py:315: size=40 B (+40 B),
count=1 (+1), average=40 B
c:\Users\Andi\Documents\GitHub\fh-se-python\exercise\Ue04\src\Memory.py:83: size=432 B (+0 B),
count=1 (+0), average=432 B
C:\Users\Andi\AppData\Local\Programs\Python\Python310\lib\random.py:378: size=432 B (+0 B),
count=1 (+0), average=432 B
c:\Users\Andi\Documents\GitHub\fh-se-python\exercise\Ue04\src\Memory.py:25: size=152 B (+0 B),
count=2 (+0), average=76 B
c:\Users\Andi\Documents\GitHub\fh-se-python\exercise\Ue04\src\Memory.py:68: size=144 B (+0 B),
count=2 (+0), average=72 B
c:\Users\Andi\Documents\GitHub\fh-se-python\exercise\Ue04\src\Memory.py:27: size=84 B (+0 B),
count=2 (+0), average=42 B
c:\Users\Andi\Documents\GitHub\fh-se-python\exercise\Ue04\src\Memory.py:53: size=76 B (+0 B),
count=2 (+0), average=38 B
C:\Users\Andi\AppData\Local\Programs\Python\Python310\lib\random.py:239: size=61 B (+0 B),
count=2 (+0), average=30 B
C:\Users\Andi\AppData\Local\Programs\Python\Python310\lib\random.py:375: size=41 B (+0 B),
count=2 (+0), average=20 B
```

3.1. Quellcode

Listing 1. Ausgabe

```
# -*- coding: utf-8 -*-
"""
```

```
@author: Andreas Wenzelhuemer
"""

import random
import tracemalloc

directions = ('N', 'E', 'S', 'W')

def generate_walk(blocks: int = 1):
    """
    Generates walk with block count

    Parameters:
        block: Count for walk generation

    Returns:
        Iterable of random directions
    """
    if blocks < 0:
        raise ValueError("Blocks must be positive")

    for _ in range(blocks):
        yield random.choice(directions)

def decode_walk(walk):
    """
    Calculates end position of given walk

    Parameters:
        walk: List with directions

    Returns:
        Calculated final position after walk
    """
    x = 0
    y = 0

    for direction in walk:
        if direction == 'N':
            y += 1
        elif direction == 'S':
            y -= 1
        elif direction == 'E':
            x += 1
        elif direction == 'W':
            x -= 1
        else:
            raise ValueError("Walk contains an invalid direction")
    return (x, y)

def distance_manhattan(start, end):

    """
    Calculates manhattan distance from start and end point

    Parameters:
        start: Start point tuple with x and y coordinates
        end: End point tuple with x and y coordinates

    Returns:
```

```
        Manhattan distance as an integer value
    """

    return sum([abs(s - e) for s, e in zip(start, end)])

def do_walk(blocks, dist = distance_manhattan, gen_walk=True):
    """
    Generates walk and calculates distance

    Parameters:
        blocks: Count for walk generation
        dist: Distance calculation method, default is manhattan distance

    Returns:
        Tuple with generated walk and manhattan distance
    """
    walk = generate_walk(blocks)
    if gen_walk:
        walk = list(walk)
    start = (0,0)
    change = decode_walk(walk)
    end = (start[0] + change[0], start[1] + change[1])

    if gen_walk:
        return walk, dist(start, end)
    else:
        return None, dist(start, end)

def monte_carlo_walk_analysis(max_blocks, repetitions = 10000):
    """
    Generates walks from length 1 to block count with n repetitions

    Parameters:
        max_blocks: Max block count which should be generated
        repetitions: Count how often should each block count generation repeated

    Returns:
        Dictionary with max length as key and generated walks and distances as tuple
    """

    if max_blocks < 0:
        raise ValueError("Max blocks have to be greater zero")
    if repetitions < 0:
          raise ValueError("Repetitions have to be greater zero")

    walks = {}
    for blocks in range(1, max_blocks + 1):
        walks[blocks] = [do_walk(blocks) for _ in range(repetitions)]
    return walks

def monte_carlo_walk(max_blocks, repetitions = 10000):
    """
    Generates walks from length 1 to block count with n repetitions

    Parameters:
        max_blocks: Max block count which should be generated
        repetitions: Count how often should each block count generation repeated

    Returns:
        Dictionary with max length as key and generated walks and distances as tuple
```

```
    """

    if max_blocks < 0:
        raise ValueError("Max blocks have to be greater zero")
    if repetitions < 0:
          raise ValueError("Repetitions have to be greater zero")

    walks = {}
    for blocks in range(1, max_blocks + 1):
        walks[blocks] = [do_walk(blocks, gen_walk=False) for _ in range(repetitions)]
    return walks


if __name__ == "__main__":
    tracemalloc.start()
    monte_carlo_walk_analysis(20)
    current_size, peak_size = tracemalloc.get_traced_memory()
    print(f'Memory peak (monte_carlo_walk_analysis): {peak_size}')
    original = tracemalloc.take_snapshot()
    tracemalloc.reset_peak()

    monte_carlo_walk(20)
    current_size, peak_size = tracemalloc.get_traced_memory()
    print(f'Memory peak (monte_carlo_walk): {peak_size}')
    improved = tracemalloc.take_snapshot()

    print("Allocations:")
    for stat in improved.compare_to(original, 'traceback'):
        print(stat)
```