04 - Memory usage

Garbage collection in Python

The Python garbage collector consists of 2 aspects:

- Reference counting
- Generational garbage collection

Reference counting

We are using the CPython implementation which means, that for every python object there is an underlying C object. That C object saves the python type and a reference count. This reference count is incremented and decremented as the references are adjusted. Once the count reaches zero, the object is freed from memory. This is ususally already very effective, but has its limits, especially when it comes to circular references.

```
import sys

custom_test_var = []
a = sys.getrefcount(custom_test_var)
custom_list = [custom_test_var]
b = sys.getrefcount(custom_list)
c = sys.getrefcount(custom_test_var)

print(a) # 2
print(b) # 2
print(c) # 3
```

Generational garbage collection

This garbage collector keeps track of all objects in memory. Each object starts in the first generation of the collector. Whenever a garbage collection is executed and the object survives, it is moved up one generation. The garbage collector has in total three generations and each generation has a threshold of how many objects can be in it. When that threshold is exceeded in any generation, the garbage collection progress is triggered for all generations. Unlike reference counting, the behaviour of the generational garbage collector can be changed via the 'gc' module.

```
import gc

# Gets thresholds for the three generations
print(gc.get_threshold()) # (700, 10, 10)
# Gets current count in each generation
print(gc.get_count())     # (612, 4, 1)
# Manually runs the collection and returns the amount of objects freed
print(gc.collect())       # 0
# Manually sets the thresholds
gc.set_threshold(100, 5, 5)
```

```
# Disables automatic garbage collection
gc.disable()
```

Profiling BaseLibrary

The BaseLibrary was adapted to the feedback from last class and the requirements of this excercise. Not saving the walks reduces the peak memory usage drasticly:

```
Peak memory used with walk saving: 46_826_457
Peak memory used without walk saving: 12_909_956
```

With the new monte_carlo_walk function each walk can be freed from memory right after processing which saves large amounts of memory.

```
import BaseLibraryNew as lib
import tracemalloc as tracer

tracer.start()

walk_dict = lib.monte_carlo_walk_analysis(20, 10_000)
walk_snapshot = tracer.take_snapshot()
_, walk_peak = tracer.get_traced_memory()
tracer.reset_peak()
tracer.stop()

tracer.start()
dist_dict = lib.monte_carlo_walk(20, 10_000)
dist_snapshot = tracer.take_snapshot()
_, dist_peak = tracer.get_traced_memory()

diff = dist_snapshot.compare_to(walk_snapshot, 'lineno')

print(f"Peak memory used with walk saving: {walk_peak}")
print(f"Peak memory used without walk saving: {dist_peak}")

print("[ Difference in the 10 most allocation heavy lines ]")
for entry in diff[:10]:
    print(entry)
```