Python

Memory Management

RAHUL,

SOLUTION ARCHITECT, ORACLE

How is memory managed in python?

- Memory management in Python involves a **private heap** containing all **Python objects** and **data structures**.
- **Interpreter** takes care of **Python heap** and that the **programmer** has **no access** to it.
- The **allocation** of heap space for Python objects is done by **Python memory manager**.
- The core API of Python provides some tools for the programmer to code reliable and more robust program.

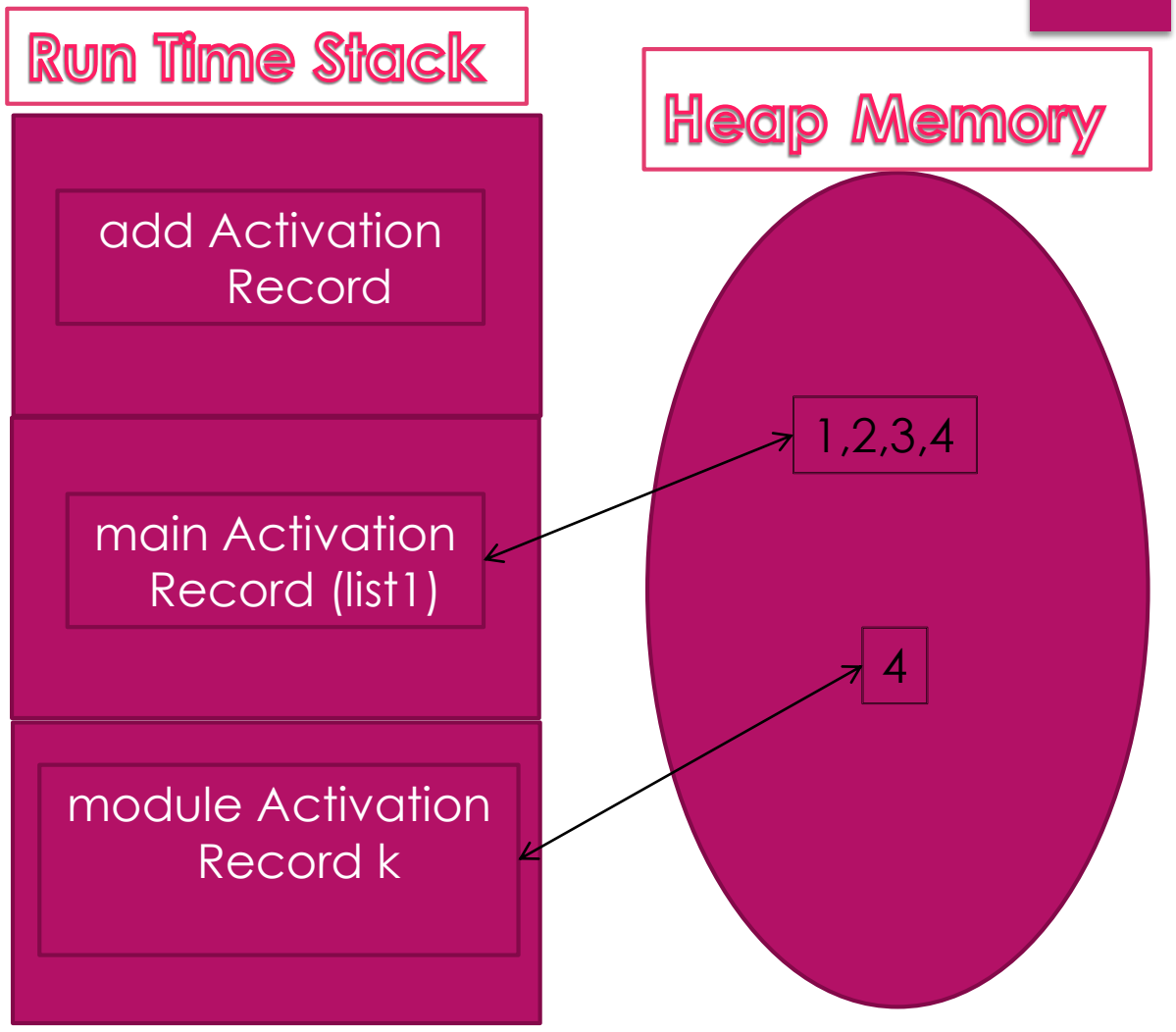
Stack and Heap Memory

- Python memory management is been divided into two parts.
  - Stack memory
  - Heap memory
- Methods and variables are created in Stack memory.
- Objects and instance variables values are created in Heap memory.
- In stack memory - a stack frame is created whenever methods and variables are created.
- These stacks frames are destroyed automaticaly whenever functions/methods returns.
- Python has mechanism of Garbage collector, as soon as variables and functions returns, Garbage collector clear the dead objects.

Python Code

```
k = 4
def main():
list1 = []
    def add():
        for x in xrange(k):
            list1.append(x)
            print list1
     add()
main()
```

Python runtime Execution



Activation Records

- Local scope of function, contained parameters and variables, must be stored someplace in the **RAM** of Computer. Python divided the RAM into two parts called **Run-time stack** and the **Heap**.
- The **activation** record is a "**chunk of memory**" (a bunch of buckets) which contains all the information necessary to keep track of a "**function call**". The run-time stack is a stack of **Activation Records**.
- When a function is called the **Python interpreter pushes the activation record** of function onto the **run-time stack**.
- When a function **returns**, its corresponding **activation record is popped** from the run-time stack. The **Heap** is area of **RAM where all values (objects) are stored**.
- The run-time stack never contains the object. The **run-time stack** store **only references pointed** to corresponding objects in the heap

Memory Management Example

- x =10 print (type(x))
  - memory manager (MM): *x points to 10*
- y = x if(id(x) == id(y)): print('x and y refer to the same object')
  - (MM): *y points to same 10 object*
- x=x+1 if(id(x) != id(y)): print('x and y refer to different objects')
  - (MM): *x points to another object is 11, previously pointed object was destroyed*
- z=10 if(id(y) != id(z)): print('y and z refer to same object') else: print('y and z refer different objects')

Garbage Collection in Python

- Python also has a **build-in garbage collector** which recycles all the unused memory.
- When an object is no longer referenced by the program, the **heap space** it occupies can be freed.
- The **garbage collector** determines objects which are no longer referenced by the **program** frees the occupied **memory** and make it available **to the heap space**.
- The gc module defines functions to enable /disable garbage collector:
  - **gc.enable()** -Enables automatic garbage collection.
  - **gc.disable()** - Disables automatic garbage collection.

Garbage collector in CPython

- Maintain reference count.
- For every object, there is a count of the total number of references to that object.
- If that count ever falls to 0,
- Then you can immediately deallocate that object because it is no longer live.

Garbage Collection Usecase

- Periodically detect reference cycles.
- Deallocating when the reference count falls to 0 doesn't work for all cases.
- Consider two objects A and B, where A holds a reference to B and B holds a reference to A. This is called a reference cycle.
- It could be the case that these are no longer live and so that both A and B should be garbage collected.
- However, the reference count on both objects are not zero, so they remain alive.
- To get around this, CPython uses an algorithm for detecting reference cycles and deallocating objects in the cycle.

Cpython Algorithm

- Performance is enhanced with heuristics.
- Objects that have been created recently are more likely to need to be garbage collected.
- CPython introduces the concept of a generation to account for the relative age of an object.
- Younger generations have objects that have more recently been created and older generations hold objects that are less recent.
- Each object belongs to exactly one generation.
- When garbage collection is performed, CPython tries to garbage collect younger generations.
- Periodically, CPython will perform garbage collection on older generations (the rate at which this happens is determined by a heuristic).

Garbage Collection Lifecycle

- Let's create a new object and see what happens with the garbage collector:
- Python wants to allocate a new object.
- To do this, it makes a call to **`_PyObject_GC_Malloc`**.
- This method assigns the object some memory locations and adds the object to the garbage collector's first generation (we'll call it generation 0).
- The method then checks to see if the number of objects in **generation 0** is greater than some **threshold**.
- If it is and the garbage collector is not currently running, then a call to **`collect_generations`** is made to begin garbage collection.
- Otherwise, the object is just allocated normally.

Garbage Collection Lifecycle

- Python starts to do garbage collection when `**collect_generations**` gets called.
- This method figures out what generation to do garbage collection on. By default CPython has 3 **generations** but this can be modified with the GC module.
- In addition, younger generations have **lower indices**, so **generation 0** is the youngest generation.
- Python will loop over all generations (**from oldest to youngest**) and detect whether a particular generation's object count is **greater than** some **threshold**.
- If it is, then it will merge all younger generations with the current generation and perform garbage collection on that generation by calling `**collect**`.
- Python wants to do garbage collection on **generation 0** for **better performance**, because this has the newest objects and also will have the **fewest objects** to iterate over.
- Doing **garbage collection** on the oldest generation is equivalent to collecting over all objects because doing garbage collection on **generation i** will use all objects in **generations 0 through i**.

Garbage Collection Lifecycle

- The `**collect**` method will run garbage collection on a specified generation.
- What this amounts to is running the **reference cycle detection algorithm** (explained later) and **finding a set of reachable and unreachable objects in a particular generation**.
- **The reachable objects will be merged into the next higher generation (i.e. if `collect` was run on generation `i`, then the reachable objects from generation `i` would be merged into generation `i+1`).**
- **For the unreachable objects, CPython will make all necessary finalizer callbacks, make weak ref callbacks, and finally deallocate the objects.**
- Finally, the internal state of the garbage collection module will be updated as `**collect**` finishes performing its duties.

CPython's Algorithm for Detecting Reference Cycles

- Python attempts to find reference cycles within a generation.
- Confining the search for reference cycles to a single generation decreases the amount of work that has to be done in a single collection (if it is one of the generations that holds younger objects).
- To find reference cycles, Python uses `**young**`, the pointer to the **head of the list of objects** for the generation that's being garbage collected, and runs the following:
  - **update_refs**(young)
  - **subtract_refs**(young)
  - **gc_init_list**(&unreachable)
  - **move_unreachable**(young, &unreachable)
- The `**update_refs**` method makes a copy of the reference count for every object in the generation so that the garbage collector can mutate its own version of the reference count without messing with the **real reference count**.

Algorithm Cont.

- Then `**subtract_refs**` goes through each of the objects `i` in the generation being garbage collected, and **decrements** the reference counts on any objects `j` in the generation list that are referenced by object `i`.
- After this method has run, the reference count on an object in the generation should equal the number of references to that object from objects which do not belong to that generation (since all references from objects within the same generation have been removed).
- Now comes the fun part.
- The `**move_unreachable**` method scans through the `**young**` list and moves objects with a reference count of 0 into the `unreachable` list and changes their reference count to `**GC_TENTATIVELY_UNREACHABLE**`.
- Objects with a **non-zero** reference count are marked as `**GC_REACHABLE**` and the objects they reference are traversed and marked as `**GC_REACHABLE**` then moved to the end of the `**young**` list so they too can be traversed later.

Algorithm Cont.

- The reason that objects with a reference count of 0 are tentatively unreachable is as follows. Suppose object A has been marked as **tentatively** unreachable and is referenced by some object B.
- Suppose that B is in the same generation as A and is actually reachable from outside the generation, but that B comes later in the `young` list than A.
- Then A would be sent to the `**unreachable**` list when `**move_unreachable**` scans over it. However, when `**move_unreachable**` scans over B, it will notice that it's reference count is non-zero, mark it as `**GC_REACHABLE**`, and traverse B's references and mark them as reachable.
- Now, A has become `**GC_REACHABLE**` as well and has been moved to the end of the `young` list so that it's references can also be marked as `**GC_REACHABLE**`.
- Thus, we only know that an object is unreachable after `**move_unreachable**` has scanned over the entire `**young**` list.
- Once the entire `**young**` list has been traversed, then all the items left in the `**unreachable**` list are definitely unreachable and so they can be **deallocated**. The items in the `**young**` list are then merged into older generations.

Algorithm Cont. Performance Notes:

- CPython's garbage collector still stops the world, but does so more infrequently than other implementations.
- Reference count deallocation increases the time between collections because the number of objects in a generation decreases whenever an object's reference count falls to 0 and gets deallocated.
- Since collections are triggered when the number of objects in a generation are above a threshold, reference count deallocation decreases the number of collections as long as there aren't too many reference cycles.
- If the hypothesis that younger objects are the objects more likely to need to be garbage collected is true, then running the garbage collector on younger generations can significantly reduce total runtime.

Algorithm Cont.

- Memory fragmentation occurs.
- Reference counting will naturally cause memory to get fragmented since the deallocation of an object means there will be small chunks of memory that get added back onto the heap.
- Some garbage collectors deal with fragmentation by copying all live objects into a different section of memory and freeing up an entire section of memory, but CPython doesn't.
- Normal execution is slower.
- There is an extra check whenever an object gets allocated, referenced, or dereferenced.
- This means that instead of running normally, then performing a garbage collection all at once like some tracing garbage collectors do, CPython's reference count garbage collector will spread the work across normal activities and perform collection less frequently.
- Unfortunately, this means the total amount of work that goes into garbage collection is higher (because of the added reference count checks).

Thank You

SUBSCRIBE OUR CHANNEL FOR NEXT VIDEO ON VIRTUALIZATION