Q:1 How memory is manged in python?

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the Python memory manager. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if they regularly manipulate object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

To avoid memory corruption, extension writers should never try to operate on Python objects with the functions exported by the C

library: malloc(), calloc(), realloc() and free(). This will result in mixed calls between the C allocator and the Python memory manager with fatal consequences, because they implement different algorithms and operate on different heaps. However, one may safely allocate and release memory blocks with the C library allocator for individual purposes, as shown in the following example:

```
PyObject *res;
char buf = (char *) malloc(BUFSIZ); / for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyBytes_FromString(buf);
free(buf); /* malloc'ed */
return res;
```

In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the bytes object returned as a result.

Q: 2 what is purpose continue statement in python?

Python Continue Statement

Python Continue statement is a loop control statement that forces to execute the next iteration of the loop

while skipping the rest of the code inside the loop for the current iteration only, i.e. when the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped for the current iteration and the next iteration of the loop will begin.

Python continue Statement Syntax

```
while True:
    ...
    if x == 10:
        continue
    print(x)
```

Application of the Continue Statement

In Python, loops repeat processes on their own in an efficient way. However, there might be occasions when we wish to leave the current loop entirely, skip iteration, or dismiss the condition controlling the loop. We use Loop control statements in such cases. The continue keyword is a loop control statement that allows us to change the loop's control.

The continue Keyword

In Python, the continue keyword return control of the iteration to the beginning of the Python for loop or Python while loop. All remaining lines in the prevailing iteration of the loop are skipped by the continue keyword, which returns execution to the beginning of the next iteration of the loop.

Both Python while and Python for loops can leverage the continue statements.

Q: 3 what are negative indexes and why are they used ?

As we know, indexes are used in arrays in all the programming languages. We can access the elements of an array by going through their indexes. But no programming language allows us to use a negative index value such as -4. Python programming language supports negative indexing of arrays, something which is not available in arrays in most other programming languages. This means that the index value of -1 gives the last element, and -2 gives the second last element of an array.

The negative indexing starts from where the array ends. This means that the last element of the array is the first element in the negative indexing which is -1.

Example:

```
arr = [10, 20, 30, 40, 50]
print (arr[-1])
print (arr[-2])
```

Output:

```
50
40
```

If you would like to dedicate more of your time to studying Python and all of its nuances, consider getting some help with your other assignments from a rewrite my essay , essaydoc -cheap essay writer and Research paper writing service