**Display address of 1D list and 2D list**

Here is the Python program updated to display each address and size one by one in a clear readable format using simple for loops:

import sys  
  
list\_1d = [10, 20, 30, 40, 50]  
list\_2d = [[1, 2, 3], [4, 5, 6]]  
  
def format\_address(addr):  
 return str(addr)  
  
print("1D list element addresses and sizes:")  
for item in list\_1d:  
 addr = format\_address(id(item))  
 size = sys.getsizeof(item)  
 print(f"Address: {addr}, Size: {size} bytes")  
  
print("\n2D list inner list addresses and sizes:")  
for inner\_list in list\_2d:  
 addr = format\_address(id(inner\_list))  
 size = sys.getsizeof(inner\_list)  
 print(f"Inner list address: {addr}, Size: {size} bytes")  
  
print("\n2D list element addresses and sizes inside inner lists:")  
for inner\_list in list\_2d:  
 for item in inner\_list:  
 addr = format\_address(id(item))  
 size = sys.getsizeof(item)  
 print(f"Address: {addr}, Size: {size} bytes")  
 print("--- End of inner list ---")

This version prints each memory address and size on its own line, grouping elements clearly and showing when each inner list ends.

**Explanation:**

* The Size: 28 bytes is the size of the Python **integer object** in memory — which includes the object header plus integer value storage.
* However, the **memory addresses** are aligned *not* simply by the object size but also by memory allocator alignment rules, which typically align objects to multiples of 8 or more bytes for performance reasons.

Here the object size is **28 bytes**, but the addresses increment by **32 bytes** because Python's memory allocator rounds up to the nearest alignment boundary, leaving some padding bytes between objects.

**Summary:**

* The size output (28 bytes) is the allocated size of the object itself.
* The address increments reflect actual memory layout with padding/alignment.
* So addresses are spaced by **32 bytes**, not exactly by the visible size (28), due to memory alignment and allocator behavior in CPython.

This is normal and expected in Python's memory management. The spacing between objects is often larger than the precise object size for alignment and bookkeeping.

**why : 2D list inner list addresses and sizes:**

**for example**

Inner list address: 2602467535744, Size: 88 bytes  
Inner list address: 2602467529088, Size: 88 bytes

The output:

2D list inner list addresses and sizes:  
Inner list address: 2602467535744, Size: 88 bytes  
Inner list address: 2602467529088, Size: 88 bytes

means the two inner lists (the sublists inside the 2D list) have distinct memory addresses and each occupies 88 bytes of memory.

**Explanation for the size and addresses:**

* Each **inner list** is a separate Python object.
* The Size: 88 bytes represents the memory allocated for the inner list object itself. This includes:
  + The list object overhead (metadata, reference counts, pointers to elements, etc.)
  + Space for the array of pointers to the elements (here each inner list holds 3 elements, so at least space for 3 pointers).
* The two inner lists having different addresses (2602467535744 and 2602467529088) confirms they are **independent objects** in memory (no contiguous single block).
* The size (88 bytes here) can vary depending on Python implementation and architecture but typically reflects the overhead plus space for references to elements.

**Summary**

* Inner lists in a Python 2D list are separate list objects with their own memory allocations.
* Their sizes (~88 bytes) include Python list overhead and pointers to elements, not the elements themselves.
* Address difference shows they are stored in distinct memory locations, not contiguous as a single 2D array block.

This matches Python’s design of lists as dynamic arrays of pointers, and 2D lists are lists of lists (not contiguous multidimensional arrays).

**so id is the function for address display?**

Yes, in Python, the built-in function id() returns a unique integer that represents the **identity** of an object during its lifetime. In CPython (the standard Python implementation), this integer corresponds to the **memory address** where the object is stored (though this is a virtual address managed by the OS).

You can view it as the object's address in memory for most practical purposes.

For example:

a = [1, 2, 3]  
print(id(a)) # Prints a unique integer (memory address)  
print(hex(id(a))) # Prints the same value in hexadecimal (memory address style)

**Note:**

* The address returned by id() is unique and constant only during the object's lifetime.
* It is a *virtual memory address* and cannot be directly used to access memory as in lower-level languages.
* Other Python implementations (like PyPy, Jython) might not use the actual memory address for id().

If you want to see addresses as memory-style hex strings for clarity, you can wrap id(obj) like hex(id(obj)).

In short:  
**Yes, id() is the standard Python function to get the memory address (identity) of an object.**