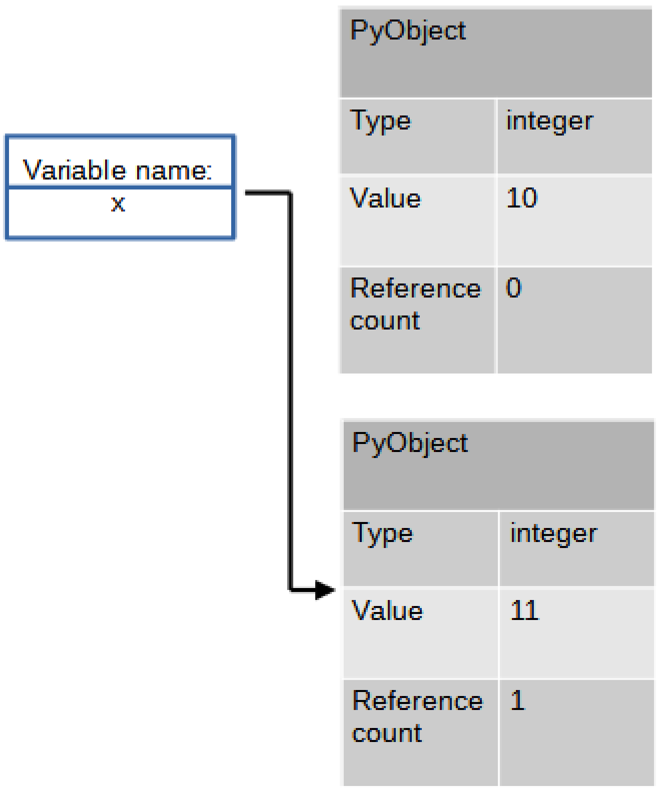
Memory management in Python

When the Python interpreter is started, the host **allocates** an amount of the hardware memory available to the interpreter, so that it can be used by the interpreted scripts. This memory is totally free of any content. Because memory is a scarce resource within a computer, other applications may attempt to use the memory allocated to the Python interpreter. A mechanism exists to prevent this. It is called **Global Interpreter Lock** and ensures that memory allocated to Python will not be accessed by other applications.

Everything in Python is an object. When an operation is applied on a variable, a new object is generated and the name of the variable is made to point to the new object.



After the change is completed, the disused object takes a reference count of 0. However, the space in memory is still held by it. This is considered to be garbage left behind in memory. If new objects are to be generated during the programme run, the memory must be **freed** to leave space for the new objects to be generated. Memory used by the Python interpreter is also managed by it.

Memory management

Python assumes that the OS allocates to it areas of memory that are aligned to memory page boundaries. These areas are 256 kB long and from the Python perspective, they are called **Arenas**. Each arena is further divided into **pools** which are 4 kB long. Each pool is linked to its neighbours through a double linked list, which allows the memory manager to traverse them. A list of pointers to all **free pools within an Arena** is also maintained. A third list of **allocated** pools complements the previous one. Under this scheme, a pool can be in either of three states: **untouched** (never used by any Python object), **free** (used in the past, but freed), and **allocated** (currently used by an object).

Arenas themselves are also linked between themselves through double linked lists. The interpreter selects for object allocation pools from the arena that bears the list amount of free pools, thus maximising the efficiency of the memory resources used.

When a new object is generated, it requests from the interpreter to allocate a number of bytes for its use. A number of pools are allocated to the object to satisfy the request. These pools are now **allocated**.

Freeing memory

As objects are disused, there reference count is zeroed. The Python interpreter periodically goes through the list of generated objects and checks whether their reference count is zero. If it is, the object is destroyed and the memory pool originally allocated to it is returned to the list of free pools. It is now available to be allocated to another object that will need it.

At the end of the interpreter session and just before exit to the host OS, the interpreter goes through all tables of allocated pools and frees them. Then, it frees all allocated arenas, before finally returning them to the OS for use by another application.