# 

# Abschnitt 3

## Variables are Memory References

|  |  |
| --- | --- |
|  | Memory Address:   * id(var) * hex(id(var)) |

## Reference Counting

|  |  |
| --- | --- |
|  | Count of References to Memory Addresses:   * sys.getrefcount(var) Note: always sub 1 from Result * ctypes.c\_long.from\_address(address).value |

## Garbage Collection

|  |
| --- |
| If Reference Count goes down to 0, Python releases the corresponding memory.  But in some cases, Reference Count is not enough to decide whether to release Memory, for example in the case circular References. In these cases, the garbage collector detects this and releases the memory. |

## Dynamic vs Static Typing

|  |  |
| --- | --- |
| Static:  myVar can only reference to String Objects in memory, because the variable itself is declared as String. |  |
| Dynamic:  The reference to an Object in memory changes and the Object can be everything. |  |
| Determine the type of the currently referenced Object: type(var)   * Python looks up the object var is referencing (pointing to), and returns the type of the object at that memory location. | |

## Variable Re-Assignment

|  |  |
| --- | --- |
| The value inside the int objects, can never be changed!  🡺 Immutable |  |

## Object Mutability

|  |  |
| --- | --- |
| An Object whose internal state CAN be changed, is called MUTUBLA: | * Lists * Sets * Dictionarys * User-Defined Classes |
| An Object whose internal state CANNOT be changed is called IMMUTABLE:  (Instead, new Object in Memory with “changed State” will be created in this case) | * Numbers (int, float, bool, …) * Strings * Tuples * Frozen Sets * User-Defined Classes |
| ATTENTION: Immutable Objects can contain MUTUBLA Objects such as Tuples that contain Lists. The Tuple cannot be changed but the referenced Objects – Lists – can be changed, so the Tuple itself appears to have been changed, if for example the containing List gets more Elements.  (The Reference in the Tuple to the List is NOT changed – TUPLE IMMUTABLE – but the internal state of the referenced Obejct in Memory changed – LIST MUTABLE!) | |

## Function Arguments and Mutability

|  |  |
| --- | --- |
|  | Immutable objects are safe from unintended side-effetcts.  ! But watch out for immutable collection objects that contain mutable objects! |
|  | Mutable objects are NOT safe from unintended side-effects |
|  | Immutable collection objects that contain mutable objects are NOT safe from unintended side-effects |

## Shared References Mutability

|  |  |
| --- | --- |
|  | Immutable Objects:  This is not critical, because if state of the Object is changed by one of the references, a new object is automatically created and the original object with the other references is not touched! |
|  | Mutable Objects:  Attention: If the state of the object is changed by a reference, this also affects the other references.  If two identical mutable objects are created, they will not be share references, because of the mutability! |

## Variable Equality

|  |  |
| --- | --- |
| By Memory Address:  a is b | a is not b | not(a is b) | By Object State: a == b | a != b | not(a == b) |
| The None object:   * None is a real Object managed by the Python Memory Manager * All Variables that are None, has the same Memory Address of None (shared reference) | |

## Everything is an Object

|  |  |
| --- | --- |
| Everything is an Object, including:   * Functions * Classes * Types   This means they all have a memory address! | my\_func now contains the reference to the memory address where the Function is saved in Memory.  Possible: a = my\_func   * a is my\_func 🡪 True * Calling Method through a(arg) |
| As a consequence:   * Any object can be assigned to a variable; including functions… * Any object can be passed to a function; including functions… * Any object can be returned from a function; including functions… | |
| a = 10 is the same as a = int(10) | |
| Der springende Punkt ist:  Funktionen sind Objekte welche in Memory gespeichert werden. Die Namen der Funktionen enthalten wieder nichts anderes als die Referenzen auf diese Memory Adresse in welchem die Funktion liegt. Ich kann diese Namen beliebig assignen und speichern sowie shared-references betreiben. Und genauso kann ich die Funktionen dann auch wieder aufrufen, ich muss nur die Referenz zur Funktione im Speicher kennen. | |

## Python Optimizations: Int Interning (using objects on-demand)

There are many implementations of Python, we are using the standard/reference implementation CPython! (Makes a difference in terms of how Python dos optimizations)

* Python pre-loads (caches) the objects for the int range [-5;256]
* For example, if we assign 10 to a variable, it simply points to the cached/existing object in memory.
* Thtats why shared references come up, for integers in the range [-5;256]

## Python Optimizations: String Interning

Beim kompilieren des Programms werden viele identifiers wie

* Variable names
* Function names
* Class names
* …

sowie einige String literale Interniert. Im Grunde heißt das, das sie vorgeladen werden und im Memory abgelegt werden; dabei werden gleiche String Inhalte wieder als shared references angelegt.

=> Das ist die eigentliche Optimierung. Denn jetzt kann Python intern bei jeglichen vergleichen anstatt Zeichen für Zeichen zu vergleichen (==), einfach die Referenzen vergleichen (is). Wenn diese auf die gleiche Memory Address zeigen sind die Strings gleich!

Man kann auch selber Interning machen (import sys):

* a = sys.intern(“This is text“)
* b = sys.intern(“This is text“)
* c = “This is text“
* a is b 🡪 True
* Aber Achtung: a is c 🡪 False

## Python Optimizations: Peehole

|  |  |
| --- | --- |
|  | In my\_func.\_\_code\_\_.co\_consts werden vorberechnete Werte gespeichert die zur Laufzeit dann schon zur Verfügung stehen und nicht erneut berechnet werden müssen. |

and more…

# Abschnitt 4

## A

## B

## C