Python Wert/Objekt Lebensdauer (Lifetime) (C) 2017-2025 T.Birnthaler OSTC GmbH
==========

--> python-scope.txt Scope/Sichtbarkeitsbereich
--> python-namespace.txt Namespace/Namensraum
--> python-lifetime.txt Lebensdauer/Lifetime/Existenz/Gültigkeitsbereich

Der Begriff "Lebensdauer/Lifetime/Existenz/Gültigkeitsbereich" (wie lange ein Python-Objekt = Wert/Objekt existiert und benutzbar ist) hat mit den Begriffen "Scope/Sichtbarkeitsbereich" und "Namespace/Namensraum" nichts zu tun!

In Python ist ein WERT/OBJEKT existent und benutzbar ("am Leben"), solange MINDESTENS EINE REFERENZ darauf zeigt und diese REFERENZ von Python aus direkt per NAME oder indirekt über andere Wert/Objekte erreichbar ist.

Sobald die LETZTE REFERENZ auf ein Wert/Objekt verschwindet, ist es nicht mehr erreichbar und sein Speicherplatz kann von Python für andere Zwecke verwendet werden. Python zählt daher die ANZAHL der Referenzen auf ein Wert/Objekt ständig in einem "Reference Counter" mit, der Teil des Wert/Objektes ist (abfragbar per sys.getrefcount(OBJ)).

Jede zusätzliche Referenz erhöht den Referenz-Zähler um 1, jede aufgelöste Referenz verringert den Referenz-Zähler um 1. Erreicht der Referenz-Zähler den Wert 0, ist sichergestellt, dass ein Wert/Objekt nicht mehr erreichbar ("referenzierbar") und damit benutzbar ist und sein Speicherplatz wieder für andere Aufgaben zur Verfügung steht.

```
  text = "hallo"           # 1. Referenz auf str-Objekt "hallo"
  ref1 = text              # 2. Referenz auf str-Objekt "hallo"
  ref2 = [ 1, 2, text ]    # 3. Referenz auf str-Objekt "hallo"

  assert id(text) == id(ref1) == id(ref2[2])   # OK da ID immer die gleiche

  ref2[2] = 3              # 3. Referenz auf str-Objekt zerstört
  ref1 = 123               # 2. Referenz auf str-Objekt zerstört
  del text                 # 1. Referenz auf str-Objekt zerstört (Name gelöscht)
```

Sich gegenseitig ZYKLISCH referenzierende aber insgesamt nicht mehr erreichbare Wert/Objekte werden von einer periodisch stattfindenden GARBAGE COLLECTION aufgeräumt (Modul "gc"):

```
  import gc                # Modul "gc" importieren
  l1 = []                  # --> l1 ist leere Liste
  l2 = [l1]                # --> Liste l2 referenziert Liste l1
  l1.append(l2)            # --> Liste l1 referenziert Liste l2 --> Zyklus!
  print(l1)                # --> [[[...]]] --> Zyklus!
  print(l2)                # --> [[[...]]] --> Zyklus!
  gc.collect()             # --> ? (0 oder beliebiger anderer Wert)
  gc.collect()             # --> 0 (d.h. Speicher für 0 Wert/Objekte freigegeben)
  del l1                   # --> Name l1 löschen --> Referenz-Zähler sinkt um 1
  del l2                   # --> Name l2 löschen --> Referenz-Zähler sinkt um 1
  # --> Wert/Objekte im Zyklus l1 <-> l2 bleiben erhalten obwohl nicht erreichbar!
  gc.collect()             # --> 2 (d.h. Speicher für 2 Wert/Objekte freigegeben)
  gc.collect()             # --> 0 (d.h. Speicher für 0 Wert/Objekte freigegeben)
```

Am Programmende löscht der Python-Interpreter immer ALLE Referenzen und daher werden auch ALLE Wert/Objekte AUFGERÄUMT (d.h. ihr Speicherplatz wird freigegeben). Dies kann durchaus zu einer WARTEZEIT nach der Ausführung der letzten Anweisung bis zum endgültigen Verlassen des Programms führen.

ACHTUNG: Das Verhalten im Rahmen einer IDE (Integrated Development Environment) ist anders, da der Python-Interpreter am Programmende von der IDE üblicherweise NICHT automatisch beendet wird (damit man noch alle im Programm definierten Wert/Objekte nutzen kann). In der Regel wird der Python-Interpreter in einer IDE erst DIREKT VOR dem nächsten Programmlauf beendet.

Datenübergabe (d.h. Übergabe von Wert/Objekten) an Funktionen und Datenrückgabe aus Funktionen erfolgt in Python grundsätzlich in Form von REFERENZEN (CALL BY/RETURN BY REFERENCE). Diese Art der Datenübergabe ist sogar sehr EFFIZIENT:

```
  def f(a, b, c):          # Referenzen auf Wert/Objekte in a, b, c übergeben;
      t = (a + b, b + c)   # Neues Tupel-Objekt mit Name t erzeugt
      return t             # Referenz auf Tupel-Objekt zurückgeben

  erg = f(1, 2, 3)         # Referenzen auf Wert/Objekte 1, 2 und 3 übergeben;
                           # erg enthält zurückgegebene Referenz auf
                           # in Funktion erzeugtes Tupel-Objekt zugewiesen
```

PARAMETER-Variablen zeigen also auf Wert/Objekte, die VON AUSSEN stammen (d.h. sie sind LOKALE NAMEN, die von außen mit Wert/Objekten initialisiert werden) und stellen eine weitere Referenz auf diese da. Sofern diese Wert/Objekte IM-MUTABLE sind, ist das problemlos. Falls ein übergebenes Wert/Objekt MUTABLE ist, kann durch Manipulation seines INNEREN in der Funktion ein SEITENEFFEKT (side-effect) nach außen entstehen (den man dem Funktionaufruf nicht ansieht).

```
  def f(a):              # --> Lokale Variable a von außen initialisieren
      a.append("c")      # --> Inneres von lokaler Variable a manipulieren
      return len(a)      # --> Länge von lokaler Variable a zurückgeben
                         #
  lst = ["a", "b"]       # --> Globales list-Objekt erzeugen
  print(len(lst), lst)   # --> 2 ["a", "b"]
  print(f(lst))          # --> 3 (Fkt. aufrufen, globales Wert/Objekt übergeben)
  print(len(lst), lst)   # --> 3 ["a", "b", "c"] (globales Wert/Objekt verändert)
```

In einer Funktion NEU ERZEUGTE WERT/OBJEKTE können problemlos als RÜCKGABEWERT per "return" zurückgegeben werden (auch wenn der lokale Variablenname außerhalb der Funktion nicht mehr existiert), da schon eine einzige Referenz auf sie außerhalb der Funktion dafür sorgt, dass sie nicht freigegeben werden können.

```
  def f(a, b):          #
     tpl = (a+b, a-b)   # --> Tupel-Objekt erzeugen, Name tpl ist lokal
     return tpl         # --> Tupel-Objekt zurückgeben, Name tpl verschwindet
                        #
  erg = f(1, 2)         # --> Funktion aufrufen + Rückgabe-Wert speichern
                        # --> Name "erg" zeigt auf in Fkt. erzeugtes Wert/Objekt
  print(erg)            # --> (3, -1)
```

ACHTUNG: Per Schlüsselwort "del" (delete) wird ein NAME gelöscht und die darin abgelegte REFERENZ auf ein Wert/Objekt entfernt. D.h. der Referenz-Zähler eines Wert/Objekts sinkt dadurch um 1. Weder wird dadurch der Destruktor "__del__" des Wert/Objekts aufgerufen noch wird sein Speicherplatz freigegeben.

Sobald der Referenz-Zähler eines Wert/Objekts auf 0 sinkt, ruft Python den zugehörigen DESTRUKTOR "__del__" (delete) auf (falls er definiert ist). Dieser dient dazu, evtl. von dem Wert/Objekt belegte RESSOURCEN (z.B. Dateien, Netzwerk-Verbindungen, Datenbank-Anmeldung, Login, Speicher, ...) freizugeben.

Der SPEICHERPLATZ des Wert/Objekts wird weder durch "del" noch durch das Sinken des Referenz-Zählers auf 0 freigegeben und wiederverwendet. Dies entscheidet und erledigt Python selbständig zum einem beliebigen späteren Zeitpunkt.

Zwei Wert/Objekte mit NICHT ÜBERLAPPENDER Lebensdauer können während der Laufzeit eines Programms sogar die gleiche ID bekommen, da der einem Wert/Objekt zugeordnete Speicher nur dann für andere Zwecke wiederverwendet wird, falls keine einzige Referenz mehr darauf existiert, d.h. der Speicher hinter dem Wert/Objekt nicht mehr erreichbar ist und im "Nirwana schwebt". An der Ausgabe der folgenden Schleife kann man dieses Verhalten beobachten, da 2 verschiedene IDs alternierend zum Speichern des Wert/Objekts 123456 ... 123465 genutzt werden.

```
  for i in range(123456, 123456+10):
      print("I", i, id(i))
```

HINWEIS: Aus Effizienzgründen werden einige häufig benötigte Wert/Objekte wie die Ganzzahlen -5 .. 256, die Booleschen Werte True und False, der Wert None, der leere String "" sowie die Fließkommawerte 0.0 und 1.0 bereits beim Start des Python-Interpreters erzeugt und während dem Programmlauf nie freigegeben (IMMORTAL Wert/Objekte).

Erreicht wird dies durch Setzen eines speziellen Referenz-Zählerwerts 4294967295 = 2^32 - 1 in diesen Wert/Objekten, der das Hoch/Runterzählen des Referenz-Zählers unterbindet.

Bei obigen Wert/Objekten ist somit auch garantiert, dass sie nur 1x erzeugt und immer wiederverwendet werden. Andere IM-MUTABLE Wert/Objekte können MEHRFACH mit dem gleichen Inhalt erzeugt werden, d.h. für Sie kann zwar gelten OBJ1 == OBJ2 (wertemäßig gleich), aber daraus folgt nicht unbedingt OBJ1 is OBJ2 (Wert/Objekt identisch, d.h. "geteilt")

```
  text1 = "welt"                              #
  text2 = "we"                                #
  text2 = text2 + "lt"                        #
  print("TEXT1", text1, id(text1))            # --> welt 432946942
  print("TEXT2", text2, id(text2))            # --> welt 432946308
  print("TEXT1 == TEXT2:", text1 == text2)    # --> True
  print("TEXT1 is TEXT2:", text1 is text2)    # --> False
```