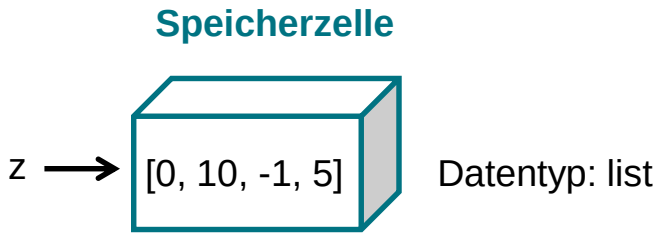
Wiederholung: Variablen und Parameter (1)

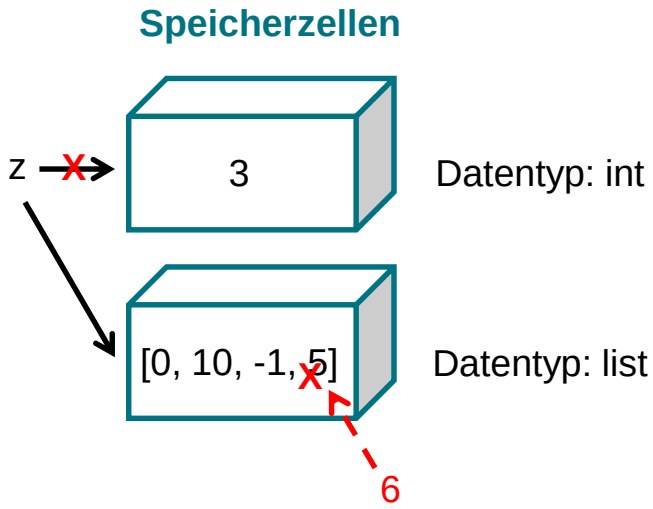
- Eines der wichtigsten Konzepte in Programmiersprachen sind Variablen. Deshalb wird dieses Thema hier nochmal wiederholt und vertieft.
- Eine **Variable** in Python ist eine **Referenz** auf eine Speicherzelle. Die Speicherzelle kann einen Wert eines beliebigen Datentyps (int, float, bool, str, tuple, list) enthalten.
- Durch eine **Zuweisung** wird der Wert in eine Speicherzelle gespeichert und wird eine Referenz darauf erstellt. Auf der rechten Seite einer Zuweisung stehen immer eine oder mehrere Variablen und auf der rechten Seite stehen die Werte.
- Mit Hilfe der Referenz kann lesend auf den gespeicherten Wert zugegriffen werden.
- Beispiel:

```
z = [0, 10, -1, 5]
print(z)
```



Wiederholung: Variablen und Parameter (2)

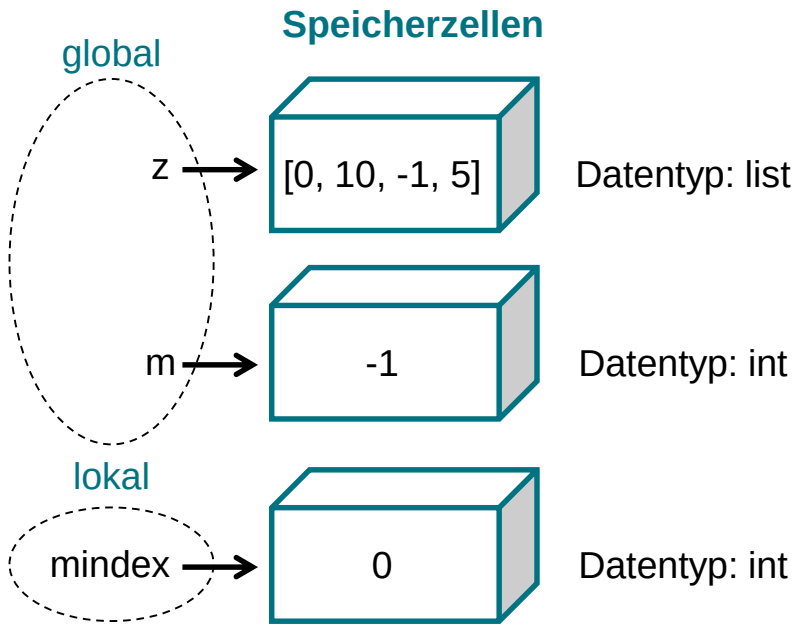
- Daten sind in Python **unveränderlich**. Das heißt, dass bei einem schreibenden Zugriff der Inhalt der Speicherzelle nicht geändert wird, sondern dass die neuen Daten in eine andere Speicherzelle gespeichert wird.
- Bei einem schreibenden Zugriff kann sich der Datentyp ändern (**dynamische Typisierung**).
- Eine Ausnahme der Unveränderbarkeit bilden Listenelemente. Diese können geändert, gelöscht oder hinzugefügt werden.
- Beispiel:
  ```
  z = 3
  z = [0, 10, -1, 5]
  z[3] = 6
  ```



Wiederholung: Variablen und Parameter (3)

- Variablen, auf die außerhalb von Funktionen schreibend zugegriffen werden, heißen **globale Variablen** und sind auch innerhalb von Funktionen verwendbar.
- Variablen, auf die innerhalb einer Funktion schreibend zugegriffen werden, heißen **lokale Variablen** und nur sind innerhalb der jeweiligen Funktion verwendbar.
- Beispiel:

```
def min_index(lst):
        mindex = 0
        # …
z = [0, 10, -1, 5]
m = min_index(z)
```

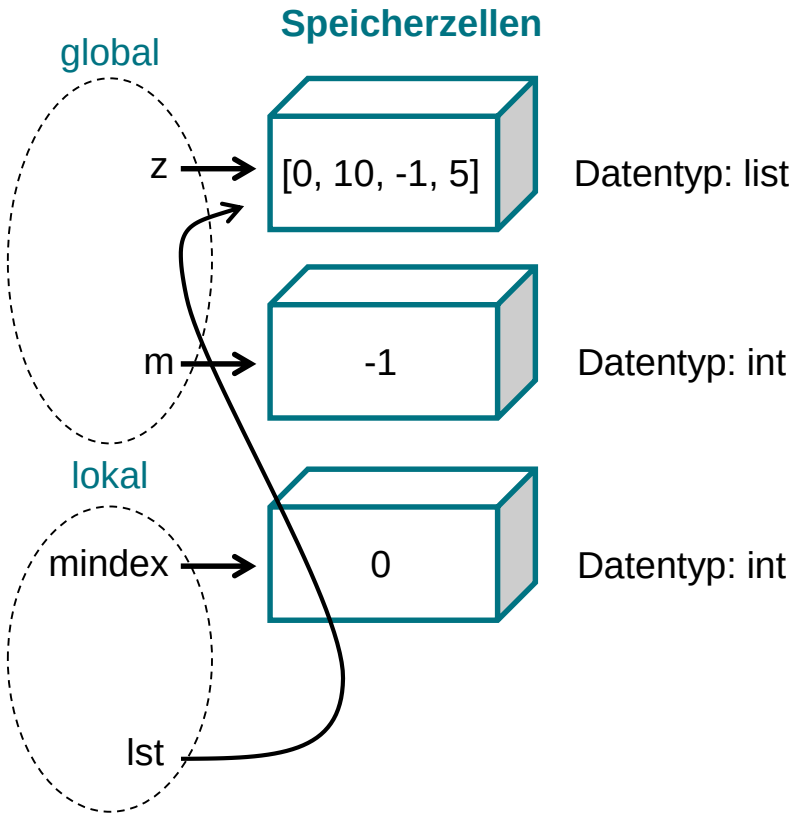


Wiederholung: Variablen und Parameter (4)

- In der Kopfzeile einer Funktion steht eine Liste von **Parametern**. Die Funktion erwartet beim Aufruf für jeden Parameter ein **Argument**. Die Argumente sind Referenzen auf Werte. Parameter werden wie lokale Variablen verwendet.

- Beispiel:

```
def min_index(lst):
    mindex = 0
    # …
z = [0, 10, -1, 5]
m = min_index(z)
```



- Die Werte werden nicht kopiert! Im Beispiel sind z und lst Referenzen auf die selbe Speicherzelle. Die Unveränderbarkeit und die Ausnahme bei Listen gilt auch beim Zugriff über Parameter.

- Beispiel:

```
lst = [0, 1]      # Ändert z nicht
lst[3] = 6        # Ändert auch z
```

Wiederholung: Variablen und Parameter (5)

- Das komplette Beispiel:

```
def min_index(lst):             # Die Funktion gibt den Index des kleinsten Elements des Parameters zurück.
    # lst = [0, 1]                                  # Ändert z nicht
    # lst[3] = 6                                    # Ändert auch z
    if not isinstance(lst, list) or len(lst) == 0:  # Der Rückgabewert ist -1 falls der Parameter keine Liste ist
        return -1                                   # oder falls die Parameterliste leer ist.
    else:
        mindex = 0
        for i in range(0, len(lst)):
            if lst[i] < lst[mindex]:
                mindex = i
        return mindex

z = [0, 10, -1, 5]
m = min_index(z)
print(z, m)
```

Wiederholung: Datentypen (1)

- Python ist eine **dynamisch typisierte** Programmiersprache. Das heißt, dass jede Variable zu jedem Zeitpunkt einen eindeutigen Datentyp hat. Der Datentyp kann sich aber im Laufe der Zeit ändern.
- Tipp: VS Code zeigt den aktuellen Datentyp einer Variablen an, wenn man mit dem Mauszeiger über den Variablennamen fährt.
- Durch implizite Typumwandlungen akzeptieren Operatoren und Funktionen oft verschiedene Datentypen.
- Beispiele:

```
z1 = 5 + 2              # int + int = int
z2 = 5.5 + 2            # float + int = float
z3 = 5 + 2.5            # int + float = float
z4 = 5.5 + 2.5          # float + float = float
print(z1, type(z1), z2, type(z2), z3, type(z3), z4, type(z4))
```

```
z1 = 5 / 2              # int / int = float
z2 = 5.5 / 2            # float / int = float
z3 = 5 / 2.5            # int / float = float
z4 = 5.5 / 2.5          # float / float = float
print(z1, type(z1), z2, type(z2), z3, type(z3), z4, type(z4))
```

```
z1 = 5 // 2             # int // int = int
z2 = 5.5 // 2           # float // int = float
z3 = 5 // 2.5           # int // float = float
z4 = 5.5 // 2.5         # float // float = float
print(z1, type(z1), z2, type(z2), z3, type(z3), z4, type(z4))
```

Wiederholung: Datentypen (2)

```
z1 = 5 % 2                  # int % int = int
z2 = 5.5 % 2                # float % int = float
z3 = 5 % 2.5                # int % float = float
z4 = 5.5 % 2.5              # float % float = float
print(z1, type(z1), z2, type(z2), z3, type(z3), z4, type(z4))

z1 = 5 > 2                  # int > int = bool
z2 = 5.5 > 2                # float > int = bool
z3 = 5 > 2.5                # int > float = bool
z4 = 5.5 > 2.5              # float > float = bool
z5 = 5 > 2 and 1 <= -1      # bool and bool = bool
print(z1, type(z1), z2, type(z2), z3, type(z3), z4, type(z4), z5, type(z5))

s = "Python"
s1 = s[2]                   # str[int] = str
s2 = s + "!"                # str + str = str
s3 = len(s)                 # len(str) = int
print(s1, type(s1), s2, type(s2), s3, type(s3))
```

```
q = ["a", "c", "e"]         # list<str> (Liste von str)
q1 = q[2]                   # list<str>[int] = str
q2 = q.pop(1)               # list<str>.pop(int) = str
q3 = "c" in q               # str in list<str> = bool
q4 = q + ["g"]              # list<str> + list<str> = list<str>
print(q1, type(q1), q2, type(q2), q3, type(q3), q4, type(q4))

q = ("a", "c", "e")         # tuple<str> (Tupel von str)
q1 = q[2]                   # tuple<str>[int] = str
q2 = "c" in q               # str in tuple<str> = bool
print(q1, type(q1), q2, type(q2))

q = [(1, 2), (3, 4)]        # list<tuple<int>>
q1 = q[1]                   # list<tuple<int>>[int] = tuple<int>
q2 = (3, 4) in q            # tuple<int> in list<tuple<int>> = bool
q3 = q[1][0]                # list<tuple<int>>[int][int] = int
print(q1, type(q1), q2, type(q2), q3, type(q3))
```

Wiederholung: For-Schleifen (1)

- Sequentielle Daten str, tuple, list können mit for-Schleifen durchlaufen werden.
- Es gibt zwei Möglichkeiten:
  - Durchlauf mit einem int-Index. Beispiel:

```
s = "Python"
for ind in range(0, len(s)):
    print(s[ind])
```

    Der Index ind ist eine ganze Zahl.

  - Durchlauf mit einem Iterator. Beispiel:

```
t = [("P", 12), ("y", 23), ("t", 14), ("h", 9)]
for iter in t:
    print(iter)
```

    Der Iterator iter hat den selben Datentyp wie die Listeninhalte, im Beispiel tuple.