lektion2_backup

October 17, 2024

1 Lektion 2

1.1 Kopieren

Wir weisen a den Wert 1 zu

[ ]:
```
a = 1
a
```

Zuweisung von b

[ ]:
```
b = a
b
```

id gibt die Identität aus

[ ]:
```
id(a), id(b)  # a und b haben dieselbe Identität
```

hier wird a ein neuer Wert zugewiesen

[ ]:
```
a = 2
a
```

b bleibt

[ ]:
```
b
```

a ist jetzt neu

[ ]:
```
id(a), id(b)
```

1.2 Kopieren II

Erzeuge Liste al

[ ]:
```
al = [1, 2, [3, 4]]
```

bl wird al zugewiesen

[ ]:
```
bl = al
```

Jetzt verändern wir in bl die 3 in der Unterliste [3, 4] und die 2

[ ]:
```
bl[2][0] = "Hallo"
bl[1] = 22
```

dadurch wurde auch al verändert, weil bl keine Kopie von al ist, sondern auf das gleiche Objekt verweist.

[ ]:
```
al
```

Hier wird eine Kopie von al erstellt

[ ]:
```
cl = al[:]
cl
```

Jetzt ersetzen wir die 1 in al durch 42

[ ]:
```
cl[0] = 42
cl
```

Ist das ok?

[ ]:
```
al
```

Klappt das?

[ ]:
```
cl[2][0] = 2018
cl
```

Warum?

[ ]:
```
al
```

Hier brauchen wir deepcopy, eine tiefe Kopie siehe Übungsaufgabe 5

1.3 Operationen auf Listen

max und min

[ ]:
```
zahlen = [1, 2, 3, 1, 6, 5, 6]
max(zahlen), min(zahlen)
```

append, remove und pop

[ ]:
```
zahlen.append(4)  # anfügen
zahlen
```

[ ]:
```
zahlen.remove(6)  # entfernt den ersten Eintrag mit Wert 6
zahlen
```

[ ]:
```
zahlen.pop(1)    # entfernt das Element mit Index 1 und gibt es zurück
```

```
[ ]: zahlen.pop(-1)   # entfernt das letzte Element und gibt es zurück
```

```
[ ]: zahlen
```

1.4 Boolsche Variablen und Operationen

Ein boolesch Variable kann nur zwei Werte annehmen: True (wahr) oder False (falsch). Boolesche Variablen werden verwendet, um Bedingungen zu überprüfen und innerhalb eines Programms zu entscheiden wie es weiter geht.

```
[ ]: True
```

```
[ ]: False
```

Die drei wichtigsten Operatoren sind und `and`, oder `or` und nicht `not`

```
[ ]: True and True, True and False  # logisches und  'and'
```

und Verknüpfung (bitweise)

```
[ ]: True & True  # bitweises und
```

oder Verknüpfung

```
[ ]: True or False  # logisches oder
```

oder Verknüpfung

```
[ ]: True | False  # bitweises oder
```

Verneinung

```
[ ]: not True  # Negation
```

1.5 Vergleiche

```
[ ]: 2 <= 2, 2 < 2
```

Gleichheit

```
[ ]: 1 == 2, 1+1==2
```

das geht nicht nur zwischen Zahlen

```
[ ]: 'a' == 'a', [1, 2] == [2, 3]
```

ungleich

```
[ ]: 1 != 2
```

enthalten sein

[ ]:
```
2 in zahlen
```

1.6 Verzweigungen, If Anweisungen

Bedingte Anweisungen (if Anweisungen) dienen dazu, den Programmfluss unter bestimmten Bedingungen verzweigen zu lassen. Damit wird es möglich während das Programm ausgeführt wird zu entscheiden, ob bestimmte Programmteile ausgeführt werden sollen oder nicht.

```
if bedinung:
    anweisung
```

bedingung ist hier ein Ausdruck, der wahr oder falsch ist. Optional kann man weitere Bedingungen angeben, die überprüft werden, falls die erste nicht erfüllt ist.

[ ]:
```
from numpy.random import rand
c = -2+4*rand()
```

[ ]:
```
print(f'c = {c} zu Anfang')

if c > 1:
    c = c*2
elif c > 2:  # elif (else if) optional
    c = -1
else:          # else optional
    c = c**3

print(f'c = {c} zu Ende')
```

[ ]:
```
a = []
if not a:
    print(' a ist eine leere Liste ')
```

Eine leere Liste hat den Wahrheitswert `False`

1.7 Funktionen

Mit Funktionen lassen sich Teile eines Programms zusammenfassen. Dadurch wird der Programmcode übersichtlicher und weniger anfällig für Fehler.

Ein wesentliches Merkmal von Funktttionen ist die Rückgabe von Werten an das die Funktion aufrufende Programm. Dazu kann man mit der `return` Anweisung Rückgaben definieren. Erreicht man innerhalb einer Funktion eine `return` Anweisung, so wird die Funktion verlassen und das Objekt zurückgeliefert, das nach der `return` Anweisung erzeugt wird. Funktionen ohne `return` Anweisungen geben `None` zurück. Funktionen haben die folgende allgemeine Struktur:

```
def funktionsname(parameter):
    anweisungen
```

Funktionen beginnen in Python mit dem Schlüsselwort `def` gefolgt vom Funktionsnamen. In einer Klammer folgen ein oder mehrere Parameter. Nach der schließenden Klammer steht ein Doppelpunkt. Eingerückt stehen in den weiteren Zeilen Anweisungen.

[ ]:
```
def mysqr2(x):
    """ Berechnet das Quadrat von x"""
    y = x**2
    return y
```

[ ]:
```
mysqr2(2)
```

[ ]:
```
def mypow(x):
    """ Berechnet x**n """
    y = x**n  # n ist 'global'  NICHT verwenden
    return y
```

[ ]:
```
n = 1/2
mypow(2)
```

besser

[ ]:
```
def mypow(x, n=2):
    """ Berechnet x**m und, falls m nicht gegeben ist, das Quadrat von x"""
    y = x**n
    n = 42
    return y
```

[ ]:
```
y = mypow(2, n=n)
n
```

Achtung !

[ ]:
```
def f(a, L=[]):
    print(f'Liste in der Funktion {L}')
    L.append(a)
    return L
```

[ ]:
```
L = 23
f(1)
```

[ ]:
```
print(f'L ausserhalb {L}')
```

[ ]:
```
f([1, 2])
```

[ ]:
```
f([1, 2, 3])
```

[ ]:
```
def f(a, L=[]):
    L.append(a)
```

```
    return L
```

[ ]:
```
# Der default Wert wird nur einmal ausgewertet und L ist hier veränderlich
print(f(1))
print(f(2))
print(f(3))
print(f(4, [2, 3, 1]))
```

besser mit None statt mit einer veränderlichen Variablen initialisieren

[ ]:
```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

[ ]:
```
print(f(1))
print(f(2))
print(f(3))
print(f(4, [2, 3, 1]))
```

1.8 Rekursionen

1.8.1 erstes Beispiel

Rekursive Funktionen sind Funktionen, die sich selbst aufrufen. Ein einfaches Beispiel ist die Fakultätsfunktion

$$n! = n(n-1)! \quad \text{und} \quad 0! = 1$$

[ ]:
```
def factorial(n):
    if n==0:
        return 1
    return n * factorial(n-1)
```

[ ]:
```
factorial(4)
```

Rekursive Funktionen bestehen aus einer Rekursionsbasis, d.h. einer Beschreibung des kleinsten Falls ($0! = 1$), und einem rekursivem Aufruf

$$n! = n \cdot (n-1)!$$

[ ]:
```
def factorial(n):
    print(n)
    if n==1:
        return 1
    return n * factorial(n-1)
```

[ ]:
```
factorial(4)
```

1.8.2 zweites Beispiel

Berechne rekursiv die Summe der Einträge einer Liste. Das kann `sum` eigentlisch schon

Basisfall $\sum_{j=0}^{-1} a_j = 0$

Für nicht negative ganze Zahlen n ist

$$\sum_{j=0}^{n} a_j = a_n + \sum_{j=0}^{n-1} a_j$$

[ ]:
```
def rek_sum(seq):

    if not seq:
        return 0

    assert isinstance(seq[-1], (int, float, complex)), 'Die Liste darf nur␣
 ↪Zahlen enthalten'

    return seq[-1] + rek_sum(seq[:-1])
```

[ ]:
```
seq1 = [1,2.,1j]
rek_sum(seq1), sum(seq1)
```

Eingabeüberprüfung

```
assert Bedingung, Fehlermeldung
```

wirft einen Fehler, einen sogenannten AssertionError, falls die Bedingung nicht erfüllt ist

[ ]:
```
seq2 = ['a',1,2]
rek_sum(seq2),
```

[ ]:
```
sum(seq2)
```

[ ]:
```

```

[ ]:
```

```