2025_11_28_sage_2

November 28, 2025

Kontrollstrukturen

[1]:
```
a=3
b=5

if a==b:
    print("a ist gleich b")
elif a<=b:                          # eine elif-Anweisung ("else if") wird nur␣
 ↪ausgeführt, wenn die vorher kommenden Bedingungen nicht erfüllt waren
    print("a ist kleiner als b")
elif b<=a:
    print("b ist kleiner als a")
else:                               # der else-Block wird genau dann ausgeführt,␣
 ↪wenn keine der vorher kommenden Bedingungen erfüllt war.
    print("nicht vergleichbar")
```

```
a ist kleiner als b
```

if versucht, die angegebene Bedingung zu einem boole'schen Wert (True, False) auszuwerten. Bei True wird der eingerückte Block ausgeführt, bei False nicht.

Vergleiche von ganzen Zahlen ergeben automatisch boole'sche Werte:

[2]:
```
parent(a==b)
```

[2]:
```
<class 'bool'>
```

[3]:
```
parent(True)
```

[3]:
```
<class 'bool'>
```

[4]:
```
parent(False)
```

[4]:
```
<class 'bool'>
```

[5]:
```
1<7
```

[5]:
```
True
```

Vergleiche zwischen symbolischen Ausdrücken ergeben neue symbolische Ausdrücke und werden nicht automatisch ausgewertet.

[6]:
```
(x+1)^2==x^2+2*x+1
```

[6]:
```
(x + 1)^2 == x^2 + 2*x + 1
```

[7]:
```
parent(_)
```

[7]:
```
Symbolic Ring
```

[8]:
```
parent(x)
```

[8]:
```
Symbolic Ring
```

Mit bool() kann man Ausdrücke in boole'sche Werte umwandeln.

[9]:
```
bool((x+1)^2==x^2+2*x+1)
```

[9]:
```
True
```

[10]:
```
bool(x<5)
```

[10]:
```
False
```

Zahlen werden als True interpretiert, wenn sie ungleich 0 sind. Die Zahl 0 wird als False interpretiert.

[11]:
```
bool(1)
```

[11]:
```
True
```

[12]:
```
bool(0)
```

[12]:
```
False
```

[13]:
```
bool(-720)    # jede Zahl außer 0 wird als True interpretiert
```

[13]:
```
True
```

[14]:
```
bool(0+I)
```

[14]:
```
True
```

[15]:
```
bool(x)
```

[15]:
```
True
```

Logische Verknüpfungen von Wahrheitswerten funktioniert wie üblich.

[16]:
```
a=12
b=30
c=35
```

```
if not 5.divides(a):
    print("5 teilt a nicht")
if 6.divides(b) or 6.divides(c):
    print("6 teilt b oder c")
if 6.divides(b) and (not 6.divides(c)):
    print("6 teilt b aber nicht c")
```

```
5 teilt a nicht
6 teilt b oder c
6 teilt b aber nicht c
```

[17]:
```
not True
```

[17]: False

[18]:
```
not False
```

[18]: True

[19]:
```
False or False
```

[19]: False

[20]:
```
True or False
```

[20]: True

[21]:
```
True or True
```

[21]: True

while-Schleife

while-Schleifen ähnlich wie if-Anweisungen. Zulätzlich wird bei erfüllter Bedingung am Ende des eingerückten Blocks wieder an den Start gesprungen und die Bedingung erneut überprüft. Der eingerückte Block wird also so lange immer wieder ausgeführt, wie die Bedingung erfüllt ist.

[22]:
```
i=1
while i<10000:
    print(i)
    i=i*2
```

```
1
2
4
8
16
32
64
128
```

```
256
512
1024
2048
4096
8192
```

Mit while kann man leicht in einer Endlosschleife landen, die nie abbricht.

Die Ausführung kann händisch mittels Kernel->Interrupt Kernel bzw. mittels der Stopptaste oben unterbrochen werden.

[23]:
```
while True:   # Endlosschleife!
    None
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[23], line 1
----> 1 while True:   # Endlosschleife!
      2     None

File signals.pyx:355, in cysignals.signals.python_check_interrupt()

KeyboardInterrupt:
```

for-Schleife

Der eingerückte Block wird für jeden Wert von n in der gegebenen Liste L ausgeführt.

[24]:
```
L=[4,8,16,16,pi]
for n in L:
    print(n)
```

```
4
8
16
16
pi
```

In for-Schleifen kann man nicht nur Listen verwenden, sondern allgemein iterables. Das sind Objekte, die schrittweise durchlaufen werden können (mehr dazu später). Ein iterable ist range(a,b), das alle ganzen Zahlen von a bis b-1 durchläuft.

[25]:
```
range(0,10)
```

[25]:
```
range(0, 10)
```

[26]:
```
parent(_)
```

[26]:
```
<class 'range'>
```

[27]:
```
for i in range(0,10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

[28]:
```
list(range(0,10))
```

[28]: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`

Listen können auch auf andere Art intuitiv erzeugt werden. (Achtung: die Zahlen in dieser und der letzten Liste sehen zwar gleich aus, sind aber nicht genau dieselben. Siehe Übung 26b.)

[29]:
```
[0..9]
```

[29]: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`

[30]:
```
[1,..,10]
```

[30]: `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

Größere Schrittweite

[31]:
```
[1,2,4,..,10]
```

[31]: `[1, 2, 4, 6, 8, 10]`

[32]:
```
range(0,11,2)
```

[32]: `range(0, 11, 2)`

[33]:
```
list(_)
```

[33]: `[0, 2, 4, 6, 8, 10]`

Negative Schrittweite

[34]:
```
range(10,0,-2)
```

[34]: `range(10, 0, -2)`

[35]:
```
list(_)
```

[35]:
```
[10, 8, 6, 4, 2]
```

[36]:
```
[10,8,..,0]
```

[36]:
```
[10, 8, 6, 4, 2, 0]
```

Listen können nicht nur Zahlen enthalten, sondern beliebige Objekte. Mit for-Schleifen kann man also über beliebige Objekte iterieren.

[37]:
```
A=[20,'hallo',pi,True,[e,IntegerRing]]
```

[38]:
```
for i in A:
    print(parent(i))
```

```
Integer Ring
<class 'str'>
Symbolic Ring
<class 'bool'>
<class 'list'>
```

[39]:
```
list(range(19))
```

[39]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
```

Funktionen

Funktionen (im Sinne der Informatik) sind kleine "Teilprogramme". Mit ihnen kann man längere Programme in übersichtliche Stücke aufteilen und muß öfter verwendete Codeteile nur einmal schreiben.

In Python werden Funktionen mit dem Schlüsselwort def definiert. Danach kommt der Name der Funktion (=die Variable, in der die Funktion gespeichert wird), in Klammern die Namen von Parametern, die der Funktion übergeben werden, und ein Doppelpunkt.

[40]:
```
def print_types(l):
    for e in l:
        print(parent(e),e)
```

Bei Aufruf der Funktion werden Werte der Parameter (hier für l) übergeben, mit denen der eingerückte Block ausgeführt wird.

[41]:
```
print_types(A)
```

```
Integer Ring 20
<class 'str'> hallo
Symbolic Ring pi
<class 'bool'> True
<class 'list'> [e, <cyfunction IntegerRing at 0x7186981172a0>]
```

[42]:
```
print_types([1,2,3,e,pi,x])
```

```
Integer Ring 1
Integer Ring 2
```

```
Integer Ring 3
Symbolic Ring e
Symbolic Ring pi
Symbolic Ring x
```

Wir schreiben eine Funktion, die den Absolutbetrag einer Zahl berechnet und zurückgibt. Das Schlüsselwort return weist Python an, die Ausführung der Funktion an dieser Stelle abzubrechen und den danach kommenden Wert (hier x) als Output zurückzuliefern.

[43]:
```
def absolutevalue(x):
    if x>0:
        return x
    else:
        return -x
```

[44]:
```
absolutevalue(5)
```

[44]: 5

[45]:
```
absolutevalue(pi)
```

[45]: pi

[46]:
```
absolutevalue(-2)
```

[46]: 2

[47]:
```
absolutevalue(0)
```

[47]: 0

Die nächsten Ergebnisse sind unerwünscht. Sie kommen daher, dass die Funktion mit Parametern aufgerufen wird, für die sie nicht gedacht ist. Wie mit solchen Fällen umzugehen ist, werden wir später noch sehen (Fehlerbehandlung, Exceptions).

[48]:
```
absolutevalue(x)
```

[48]: -x

[49]:
```
absolutevalue(IntegerRing)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[49], line 1
----> 1 absolutevalue(IntegerRing)

Cell In[43], line 2, in absolutevalue(x)
      1 def absolutevalue(x):
----> 2     if x>Integer(0):
```

```
      3         return x
      4     else:

File /opt/sagemath/sage-10.7/src/sage/rings/integer.pyx:925, in sage.rings.integer.
 ↪Integer.__richcmp__()
    923     c = mpz_cmp_d((<Integer>left).value, d)
    924 else:
--> 925     return coercion_model.richcmp(left, right, op)
    926
    927 return rich_to_bool_sgn(op, c)

File /opt/sagemath/sage-10.7/src/sage/structure/coerce.pyx:2060, in sage.structure.
 ↪coerce.CoercionModel.richcmp()
   2058 # so we raise an exception.
   2059 if op == Py_LT:
-> 2060     raise bin_op_exception('<', x, y)
   2061 elif op == Py_LE:
   2062     raise bin_op_exception('<=', x, y)

TypeError: unsupported operand parent(s) for <: 'Integer Ring' and '<class␣
 ↪'_cython_3_0_11.cython_function_or_method'>'
```

Man kann auch bereits definierte Funktionen überschreiben

[50]:
```
gcd(3,4)
```

[50]: 1

[51]:
```
def gcd(x,y):
    return x+y
```

[52]:
```
gcd(3,4)
```

[52]: 7

Wiederherstellung der ursprünglichen Funktion

[53]:
```
restore('gcd')
```

[54]:
```
gcd(3,4)
```

[54]: 1

Man kann bei der Definition diner Funktion Standardwerte für einige Parameter festlegen. Diese werden dann verwendet, wenn weniger Parameter übergeben werden.

[55]:
```
def increment(a,b=1):
    return a+b
```

[56]:
```
increment(6)
```

[56]: 7

[57]:
```
increment(6,2)
```

[57]: 8

Parameter ohne Standardwerte müssen zuerst kommen.

[58]:
```
def increment(a=1,b):
    return a+b
```

```
  Cell In[58], line 1
    def increment(a=Integer(1),b):
                                  ^
SyntaxError: non-default argument follows default argument
```

[59]:
```
def addiere(a,b,c,d=1,e=1,f=1):
    return a+b+c+d+e+f
```

[60]:
```
addiere(1,2,3)
```

[60]: 9

Explizite Angabe, welcher der optionalen Parameter geändert werden soll. Das wird oft verwendet, um Funktionen mit zusätzlichen Optionen auszustatten, die alle einen Standardwert haben, aber gezielt geändert werden können.

[61]:
```
addiere(1,2,3,f=7)
```

[61]: 15

Rekursion

Wenn eine Funktion sich selbst aufruft, spricht man von Rekursion. Eas klassische Beispiel einer rekursiv definierten Funktion ist die Faktorielle n! = n*(n-1)! für n>=1, zusammen mit dem Startwert 0!=1.

[62]:
```
def fac(n):
    if n==0:
        return 1
    return n*fac(n-1)
```

[63]:
```
fac(6)
```

[63]: 720

Achtung: bei unerwarteten Parametern kann es leicht geschehen, dass die Rekursion niemals endet (in diesem Beispiel, weil niemals 0 erreicht wird.) Python hat eine maximale Rekursionstiefe, bei deren überschreiten eine Fehlermeldung generiert wird.

[ ]:
```
fac(3/2)
```

[65]:
```
sys.getrecursionlimit()
```

[65]:
```
3000
```

Rekursive Berechnung des ggT (Euklidischer Algorithmus)

[66]:
```
def ggT(m,n):
    if m<0:
        return ggT(-m,n)    # ggT(-m,n)=ggT(m,n)
    if n<0:
        return ggT(m,-n)
    # ab jetzt: m,n >= 0
    if m<n:
        return ggT(n,m)     # ggT(n,m)=ggT(m,n)
    # ab jetzt: m>=n>=0
    if n==0:
        return m    # ggT(m,0)=m
    # ab jetzt: m>=n>0
    return ggT(m-n,n)   # ggT(m-n,n)=ggT(m,n)
```

[67]:
```
ggT(6,9)
```

[67]:
```
3
```

[68]:
```
ggT(135,45)
```

[68]:
```
45
```

[69]:
```
ggT(-45,81)
```

[69]:
```
9
```

Fibonacci-Zahlen: f_1 = f_2 = 1; f_n = f_{n-1} + f_{n-2}

[70]:
```
def fib(n):
    if n<3:
        return 1
    return fib(n-1)+fib(n-2)
```

[71]:
```
fib(1)
```

[71]:
```
1
```

[72]:
```
fib(2)
```

[72]: 1

[73]:
```
fib(3)
```

[73]: 2

[74]:
```
fib(4)
```

[74]: 3

[75]:
```
fib(5)
```

[75]: 5

[76]:
```
fib(6)
```

[76]: 8

[77]:
```
fib(33)
```

[77]: 3524578

Die letzte Berechnung hat etwas gedauert. Das liegt daran, dass jeder Aufruf von fib zwei weitere Aufrufe generiert -> Exponentielles Wachstum der Zahl der Aufrufe.

Wir wollen nun diese Anzahl tracken und speichern die in einer variable calls. Diese ist eine globale Variable. Innerhalb des Blocks einer Funktion hat man normalerweise nur Zugriff auf lokale Variablen, also solche, die in diesem Block definiert wurden. Mit dem Schlüsselwort global können wir globale Variablen zugreifbar machen.

[88]:
```
calls=0    #wie oft die Funktion fib aufgerufen wurde
def fib(n):
    global calls # Zugriff auf globale Variable
#    print(calls)
    calls+=1     # kurz für: calls=calls+1
    if n<3:
        return 1
    return fib(n-1)+fib(n-2)
```

[79]:
```
fib(6)
```

[79]: 8

[80]:
```
calls
```

[80]: 15

Um fib(6) zu berechnen, wurde also 15 Mal die Funktion fib() aufgerufen -> Rekursionsbaum zeichnen.

[81]:
```
calls=0
fib(33)
```

[81]: 3524578

[82]:
```
calls
```

[82]: 7049155

Um das exponentielle Wachstum der Funktionsaufrufe zu verhindern, müssen wir die bereits bekannten Werte von fib() aufheben. Dann wird bei jedem Aufruf nicht zweimal der volle Rekursionsbaum durchlaufen, sondern nur einmal, für fib(n-1). Für fib(n-2) wird dann einfach der bereits aus der Berechnung von fib(n-1) bekannte Wert genommen.

Das könnte man händisch implementieren (den am besten geeigneten Datentyp, das dictionary, lernen wir erst kennen). Zum Glück kann uns Python diese Arbeit aber abnehmen, wir müssen ihm nur sagen, dass es die Werte der Funktion speichern soll. Das geht mit dem Dekorator @cached_function, direkt vor die Funktionsdefinition geschrieben.

[92]:
```
calls=0    #wie oft die Funktion fib aufgerufen wurde
@cached_function
def fib(n):
    global calls # Zugriff auf globale Variable
    calls+=1     # kurz für: calls=calls+1
    if n<3:
        return 1
    return fib(n-1)+fib(n-2)
```

[93]:
```
fib(33)
```

[93]: 3524578

[94]:
```
calls
```

[94]: 33

Das recursion limit von Python kann schnell schlagend werden:

[ ]:
```
@cached_function
def fib(n):
    if n<3:
        return 1
    return fib(n-1)+fib(n-2)

fib(1000)
```

Die unterste (=erste) Fehlermeldung zeigt die Quelle der Probleme: maximale Rekursionstiefe überschritten.

Als Workaround kann man schrittweise vorgehen, sodass kein individueller Aufruf die Rekursionstiefe überschreitet (Erinnerung: Werte werden mitgespeichert, der Wert von fib(500) ist bei der Berechnung von fib(1000) also bereits bekannt.

[96]:
```
@cached_function
def fib(n):
    if n<3:
        return 1
    return fib(n-1)+fib(n-2)

fib(500)
fib(1000)
```

[96]:
```
434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798400792551692959225930803226347752096896232398733224711616429964409065331879382989696499285160037044761377951668492288 75
```

Alternativ kann man das Rekursionslimit händisch erhöhen.

[97]:
```
sys.getrecursionlimit()
```

[97]:
```
3000
```

[98]:
```
sys.setrecursionlimit(5000)
```

[99]:
```
@cached_function
def fib(n):
    if n<3:
        return 1
    return fib(n-1)+fib(n-2)

fib(1000)
```

[99]:
```
434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798400792551692959225930803226347752096896232398733224711616429964409065331879382989696499285160037044761377951668492288 75
```

[ ]: