2. Vorlesung 6.12.2024

Inhaltsverzeichnis

Kontrollstrukturen

Die Möglichkeit, den Programmablauf zu steuern, ist für imperative Programmierung unabdinglich.
Wie bereits von Matlab bekannt, gibt es auch in Python (und damit Sage) mehrere solche Konstrukte, jeweils mit einem *Doppelpunkt* abgeschlossen.

In Python ist der bedingte Block jeweils *einzurücken* - diese Einrückung erhöht die Lesbarkeit und macht ein abschließendes "end" unnötig. Der Block endet, wenn die Einrückung endet.

Kontrollstrukturen können beliebig verschachtelt werden, wobei jeweils auf die entsprechende Ein- sowie Ausrückung geachtet werden muss.

Kontrollstruktur: if-Verzweigung

Die grundlegendste Kontrollstruktur ist die einfache **if**-Verzweigung: Nur wenn die *Bedingung* erfüllt ist, wird der eingerückte Block ausgeführt.

In [1]:
```
a = 1
if a < 1:
    print("a ist kleiner als 1!")

if a > 1:
    print("a ist größer als 1!")
```

Falls mehrere mögliche Fälle überprüft werden sollen, kann mittels **elif** eine weitere Bedingung angegeben werden.
Diese Bedingung wird nur überprüft, wenn die zuvorige **if**-Bedingung sowie etwaige vorherige **elif**-Bedingungen nicht erfüllt waren.

Eine **else**-Bedingung funktioniert wie eine **elif**-Bedingung, die immer zutreffend ist. Also: sofern die dazugehörigen **if**- und **elif-**Bedingungen nicht erfüllt waren, wird der eingerückte Block ausgeführt.

In [2]:
```
a = 3
b = 1
if a == b:
    print("a ist gleich b!")
elif a <= b:
    print("a ist kleiner als b!")
elif b <= a:
    print("b ist kleiner als a!")
else:
    print("Ist <= hier etwa keine Totalordnung?")
```

```
b ist kleiner als a!
```

Bedingungen für Fortgeschrittene

Wahrheitswerte ("wahr" oder "falsch") werden in der Informatik für gewöhnlich als *boolean* bezeichnet. In Python heißt die "wahr"-Konstante **True** und die "falsch"-Konstante **False**.
Jede *Bedingung* ist in Wahrheit ein solcher *boolean*-Wert. Falls der angegebene Ausdruck noch kein *boolean*-Wert sein sollte, wird er in einen solchen umgewandelt. Hierbei wird beispielsweise aus der Null der Wert **False** und aus jeder anderen Zahl der Wert **True**.

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

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

In [4]:
```
parent(1<2)
```

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

In [5]:
```
parent(42)
```

Out[5]:
```
Integer Ring
```

In [6]:
```
bool(42)
```

Out[6]:
```
True
```

In [7]:
```
bool(0)
```

Out[7]:
```
False
```

Mehrere *boolean*-Werte können miteinander verknüpft werden, um eine komplexere Bedingung zu bilden.
Hierfür gibt es die üblichen Operatoren **not** (unär) sowie **and** und **or** (binär). Um die Abfolge explizit zu machen, können **Klammern** verwendet werden.

In [8]:
```
not True
```

Out[8]:
```
False
```

In [9]:
```
True and False
```

Out[9]:
```
False
```

In [10]:
```
True or False
```

Out[10]:
```
True
```

In [11]:
```
a = 12
b = 30
c = 35

if not 5.divides(a):
    print("5 ist kein Teiler von a!")

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 ist kein Teiler von a!
6 teilt b oder c!
6 teilt b, aber nicht c!
```

Achtung, Gleichungen zwischen formalen Objekte werden nicht ausgewertet, sondern es entsteht eine formale Gleichung, die man ggf. lösen kann (später).

In [13]:
```
(x+1)^2==x^2+2*x+1
```

Out[13]:
```
(x + 1)^2 == x^2 + 2*x + 1
```

In [14]:
```
bool(_)
```

Out[14]:
```
True
```

In [15]:
```
bool(x<5)
```

Out[15]:
```
False
```

In [16]:
```
bool(x>5)
```

Out[16]:
```
False
```

In [17]:
```
bool(I<0)
```

Out[17]:
```
False
```

In [18]:
```
bool(I>0)
```

Out[18]:
```
True
```

(hier handelt es sich um eine interne Ordnung der Terme, die keine mathematische Bedeutung hat).

Kontrollstruktur: while-Schleife

Wie bereits aus Matlab bekannt, ähnelt die **while**-Schleife in ihrer Natur der **if**-Verzweigung.
Der Unterschied besteht darin, dass nach jedem Durchlauf des eingerückten Blocks wiederum die *Schleifenbedingung* überprüft und gegebenenfalls der Block erneut durchlaufen wird.

In [19]:
```
i = 2
while i < 10000:
    print(i)
    i = i * 2
print("fertig")
```

```
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
fertig
```

wenn man nicht aufpasst, gerät man in eine Endlosschleife, die nur mit Gewalt (Menü: [Kernel]->[Interrupt]) unterbrochen werden kann.

In [21]:
```
i=2
while i>0:
    print(i)
    i=i+1
```

Kontrollstruktur: for-Schleife

Die strukturiertere Alternative hierzu bietet die **for**-Schleife, die den eingerückten Block für jeden Wert (in einer Liste oder ähnlichen *iterierbaren* Objekten) einmal durchläuft.
(Im weiteren Verlauf der Lehrveranstaltung werden andere iterierbare Objekte vorgestellt werden.)

In [22]:
```
L = [4, 8, 15, 16, 23, 42, pi]
for n in L:
    print(n)
```

```
4
8
15
16
23
42
pi
```

Listen für Fortgeschrittene (Teil 1)

Um längere Listen zu erzeugen, gibt es verschiedene Kurzschreibweisen:

Implizite Schrittweite 1

In [23]:
```
range(0,10)
```

Out[23]:
```
range(0, 10)
```

In [24]:
```
parent(_)
```

Out[24]:
```
<class 'range'>
```

In [25]:
```
list(range(10))
```

Out[25]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [26]:
```
[0,..,10]
```

Out[26]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [27]:
```
[0..10]
```

Out[27]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Explizite Schrittweite

In [28]:
```
list(range(0,10,2))
```

Out[28]:
```
[0, 2, 4, 6, 8]
```

In [29]:
```
[0,2,..,10]
```

Out[29]:
```
[0, 2, 4, 6, 8, 10]
```

Auch möglich - negative Schrittweite!

In [30]:
```
list(range(10,0,-2))
```

Out[30]:
```
[10, 8, 6, 4, 2]
```

In [31]:
```
[10,8,..,0]
```

Out[31]:
```
[10, 8, 6, 4, 2, 0]
```

Listenelemente sind nicht auf Zahlen beschränkt - beliebige Datentypen können in derselben Liste gespeichert werden.

Listen selbst sind auch eine Form von Datentyp - so können beispielsweise Listen in Listen enthalten sein.

In [32]:
```
A = [20, "foo", True, [log(15), "bar"]]
print(A)
for el in A:
    print(parent(el))
```

```
[20, 'foo', True, [log(15), 'bar']]
Integer Ring
<class 'str'>
<class 'bool'>
<class 'list'>
```

Funktionen

Wie in fast jeder imperativen Programmiersprache kann man auch in Python kleine "Teilprogramme" - so genannte *Funktionen*, nicht zu verwechseln mit dem mathematischen Begriff - definieren.
Dies erlaubt es, den Programmcode gut lesbar und übersichtlich zu halten, indem idente Operationen nicht mehrmals ausprogrammiert werden müssen.

In Python werden Funktionen mit dem Schlüsselwort **def** definiert.

In [33]:
```
def print_list_types(l):
    print("======")
    for e in l:
        print(parent(e), e)

print_list_types([1,2,"foo",1/2])
print_list_types([["abra","cadabra"], log(e), 15^51])
```

```
======
Integer Ring 1
Integer Ring 2
<class 'str'> foo
Rational Field 1/2
======
<class 'list'> ['abra', 'cadabra']
Symbolic Ring 1
Integer Ring 95643225032107438035511171037228450586553663015365600585937 5
```

Mittels des Schlüsselworts **return** kann die Funktion frühzeitig beendet werden. Optional kann dem **return** ein Wert nachfolgen - dieser ist dann der *Rückgabewert* der Funktion.

In [34]:
```
def absolute(x):
    if x > 0:
        return x
    else:
        return -x

print(absolute(-42), absolute(15))
```

```
42 15
```

In [35]:
```
absolute(-42)
```

Out[35]:
```
42
```

In [36]:
```
absolute(15)
```

Out[36]:
```
15
```

In [37]:
```
absolute(I)
```

Out[37]:
```
I
```

In [38]:
```
absolute(x)
```

Out[38]:
```
-x
```

In [39]:
```
absolute(ZZ)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
Cell In [39], line 1
----> 1 absolute(ZZ)

Cell In [34], line 5, in absolute(x)
      3     return x
      4 else:
----> 5     return -x

TypeError: bad operand type for unary -: 'sage.rings.integer_ring.Integer
ng_class'
```

In [40]:
```
sin(1.5)
```

Out[40]:
```
0.997494986604054
```

In [41]:
```
def sin(x):
    return -x
```

In [43]:
```
sin(1.5)
```

Out[43]:
```
-1.50000000000000
```

In [44]:
```
restore('sin')
```

Für "unübliche" Funktionsparameter kann ein *Standardwert* vergeben werden. Der Parameter wird dadurch *optional*, d.h. muss nicht mehr angegeben werden.

In [45]:
```
def inc(a,b=1):
    return a+b
```

In [46]:
```
inc(1)
```

Out[46]:
```
2
```

In [47]:
```
inc(1,3)
```

Out[47]:
```
4
```

Dabei ist die obige Reihenfolge einzuhalten:

In [48]:
```
def inc(a=1,b):
    return a+b
```

```
  Cell In [48], line 1
    def inc(a=Integer(1),b):
                         ^
SyntaxError: non-default argument follows default argument
```

Es können beliebig viele optionale Argumente angegeben werden.

In [49]:
```
def inc(a,b,c=x,d=x^2):
    return a+b+c+d
```

In [50]:
```
inc(1,2)
```

Out[50]:
```
x^2 + x + 3
```

In [51]:
```
inc(1,2,3)
```

Out[51]:
```
x^2 + 6
```

In [52]:
```
inc(1,2,d=5)
```

Out[52]:
```
x + 8
```

Rekursion

Es ist auch erlaubt, innerhalb einer Funktion dieselbe Funktion nochmals aufzurufen. Dies bezeichnet man als *Rekursion*.

In [53]:
```
def fac(n):
    if n == 0:
        return 1
    return n*fac(n-1)
```

In [54]:
```
fac(4)
```

Out[54]:
```
24
```

Bei falschen Parametern kann einiges schiefgehen.

In [55]:
```
fac(3/2)
```

```
---------------------------------------------------------------------------
RecursionError                                Traceback (most recent call las
Cell In [55], line 1
----> 1 fac(Integer(3)/Integer(2))

Cell In [53], line 4, in fac(n)
      2 if n == Integer(0):
      3     return Integer(1)
----> 4 return n*fac(n-Integer(1))

Cell In [53], line 4, in fac(n)
      2 if n == Integer(0):
      3     return Integer(1)
----> 4 return n*fac(n-Integer(1))

    [... skipping similar frames: fac at line 4 (2969 times)]

Cell In [53], line 4, in fac(n)
      2 if n == Integer(0):
      3     return Integer(1)
----> 4 return n*fac(n-Integer(1))

Cell In [53], line 2, in fac(n)
      1 def fac(n):
----> 2     if n == Integer(0):
      3         return Integer(1)
      4     return n*fac(n-Integer(1))

File /usr/lib/python3/dist-packages/sage/structure/element.pyx:1112, in s
e.structure.element.Element.__richcmp__ (build/cythonized/sage/structure/
ement.c:10569)()
   1110         return (<Element>self)._richcmp_(other, op)
   1111     else:
-> 1112         return coercion_model.richcmp(self, other, op)
   1113
   1114 cpdef _richcmp_(left, right, int op):

File /usr/lib/python3/dist-packages/sage/structure/coerce.pyx:1981, in sa
.structure.coerce.CoercionModel.richcmp (build/cythonized/sage/structure/
erce.c:20759)()
   1979     assert not (isinstance(x, Element) and
   1980         (<Element>x)._parent.get_flag(Parent_richcmp_element_with
t_coercion))
-> 1981     return PyObject_RichCompare(x, y, op)
   1982
   1983 # Comparing with coercion didn't work, try something else.

RecursionError: maximum recursion depth exceeded in comparison
```

Der grösste gemeinsame Teiler mit dem euklidischen Algorithmus.

In [56]:

```
def ggT(a,b):
    if a < 0:
        return ggT(-a,b)
    if b < 0:
        return ggT(a,-b)
    if a < b:
        return ggT(b,a)
    if b == 0:
        return a
    return ggT(b,a-b)
```

In [57]:
```
ggT(135,165)
```

Out[57]:
```
15
```

Die Fibonaccifolge

In [58]:
```
def fib1(n):
    if n < 2:
        return 1
    else:
        return fib1(n-1)+fib1(n-2)
```

Das wird sehr langsam.

In [59]:
```
fib1(33)
```

Out[59]:
```
5702887
```

Wir zählen mit, wie oft die Funktion aufgerufen wird. Wenn innerhalb einer Funktion eine Variable definiert wird, hat sie nichts mit Variablen außerhalb der Funktion zu tun. Wir bezeichnen solche Variablen als *lokal*.
Falls eine Funktion explizit eine *globale* Variable verändern will, kann diese Variable mittels des Schlüsselworts **global** sichtbar gemacht werden.

In [60]:
```
nsa = []

def fib2(n):
    global nsa
    nsa.append(n)

    if n < 2:
        return 1
    else:
        return fib2(n-1)+fib2(n-2)
```

In [61]:
```
fib2(13)
len(nsa)
```

Out[61]:
```
753
```

Für rekursiv definierte mathematische Funktionen mit hoher Rekursionstiefe (oder einer hohen Anzahl an redundanten Aufrufen mit denselben Parametern während der Berechnung) kann eine Definition als klassische rekursive Funktion oft äußerst langsam werden.
Aus diesem Grund gibt es die Möglichkeit, die Funktion für jeden möglichen Parameterwert nur einmal auszuführen und die Rückgabewerte zu speichern. Dies erreicht man durch **@cached_function**, einen sogenannten *Dekorator*.

In [78]:
```
nsa3 = []
@cached_function
def fib3(n):
    global nsa3
    nsa3.append(n)

    if n < 2:
        return 1
    else:
        return fib3(n-1) + fib3(n-2)
```

In [79]:
```
fib3(10)
```

Out[79]:
```
89
```

In [80]:
```
nsa3
```

Out[80]:
```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Weiters zu beachten ist die von Python vorgegebene *maximale Rekursionstiefe*.

In [83]:
```
fib3(2000)
```

Sie kann schrittweise umgangen werden:

In [72]:
```
fib3(500)
```

Out[72]:
```
225591516161936330872512695036072072046011324913758190588638866418474627738868834050159870527969684986 26
```

In [73]:
```
fib3(1000)
```

Out[73]:
```
703303677114228158218352548771835497701812698363587327426049050871545371181969335797422494945626117334877504492417659910881863632654502236471060120533741212738673391111981393731255987676900919022452453234035 01
```

oder bei Bedarf mittels *sys.setrecursionlimit(depth)* erhöht werden.

In [74]:
```
sys.getrecursionlimit()
```

Out[74]:
```
3000
```

In [75]:
```
sys.setrecursionlimit(5000)
```

Auch die bekannten **Operatoren**, also z.B. "+" und "-", sind nichts anderes als Funktionen, die zwei Parameter (<-> zwei Seiten des Operators) erwarten.

Falls man diese Funktionen direkt aufrufen möchte, findet man sie unter *operator*, also z.B. *operator.add* und *operator.sub*.

In [76]:
```
operator.add(operator.sub(5,2),operator.sub(9,4))     # dieser Ausdruck i
```

Out[76]:
```
8
```

In [ ]:
```

```