Informatik

inf12 08.10.2020

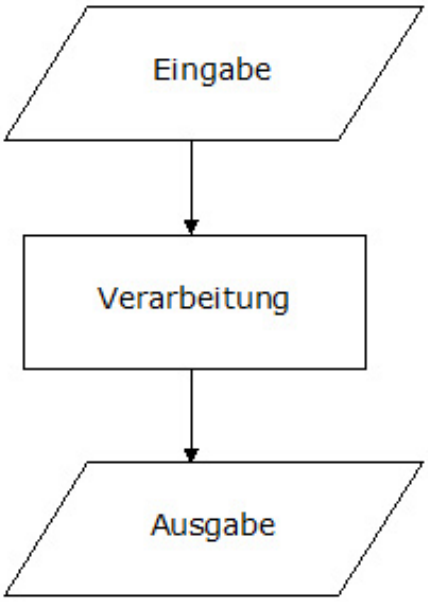
Definition:
Unter einem **Algorithmus** verstehen wir ein aus endlich vielen Anweisungen bestehendes allgemeines Verfahren, welches eine Klasse von Problemen in endlich vielen Schritten löst.

Wir beschreiben einen Algorithmus, unabhängig von der Programmiersprache, in der er codiert wird, durch ein Flußdiagramm oder ein Struktogramm („Nassi-Shneiderman"-Diagramm).

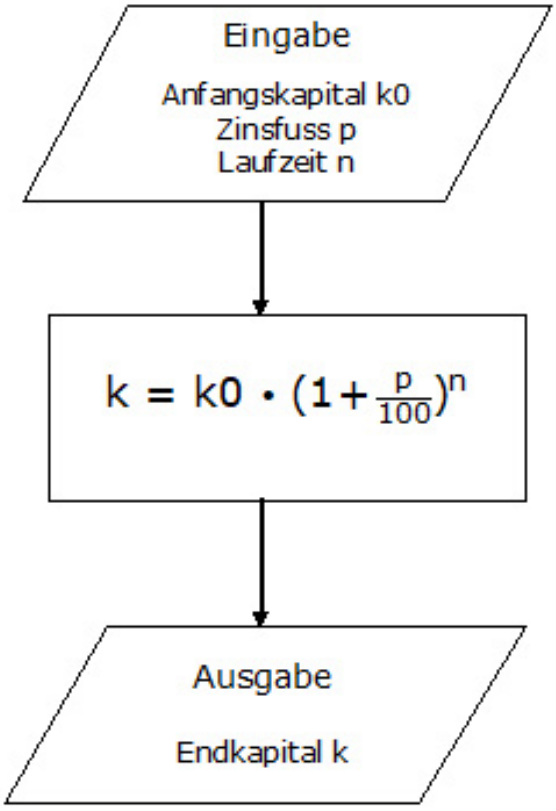
1. Lineare Algorithmen

Wenn bei der Abarbeitung eines Algorithmus die einzelnen Anweisungen sich längs eines einzigen Pfades aneinanderreihen, sprechen wir von einem linearen Algorithmus; insbesondere gibt es hier keine Verzweigungen. Beispiel: Zinseszinsberechnung.

Allgemeines Flußdiagramm eines Algorithmus:



Flußdiagramm des Algorithmus „Zinseszins":

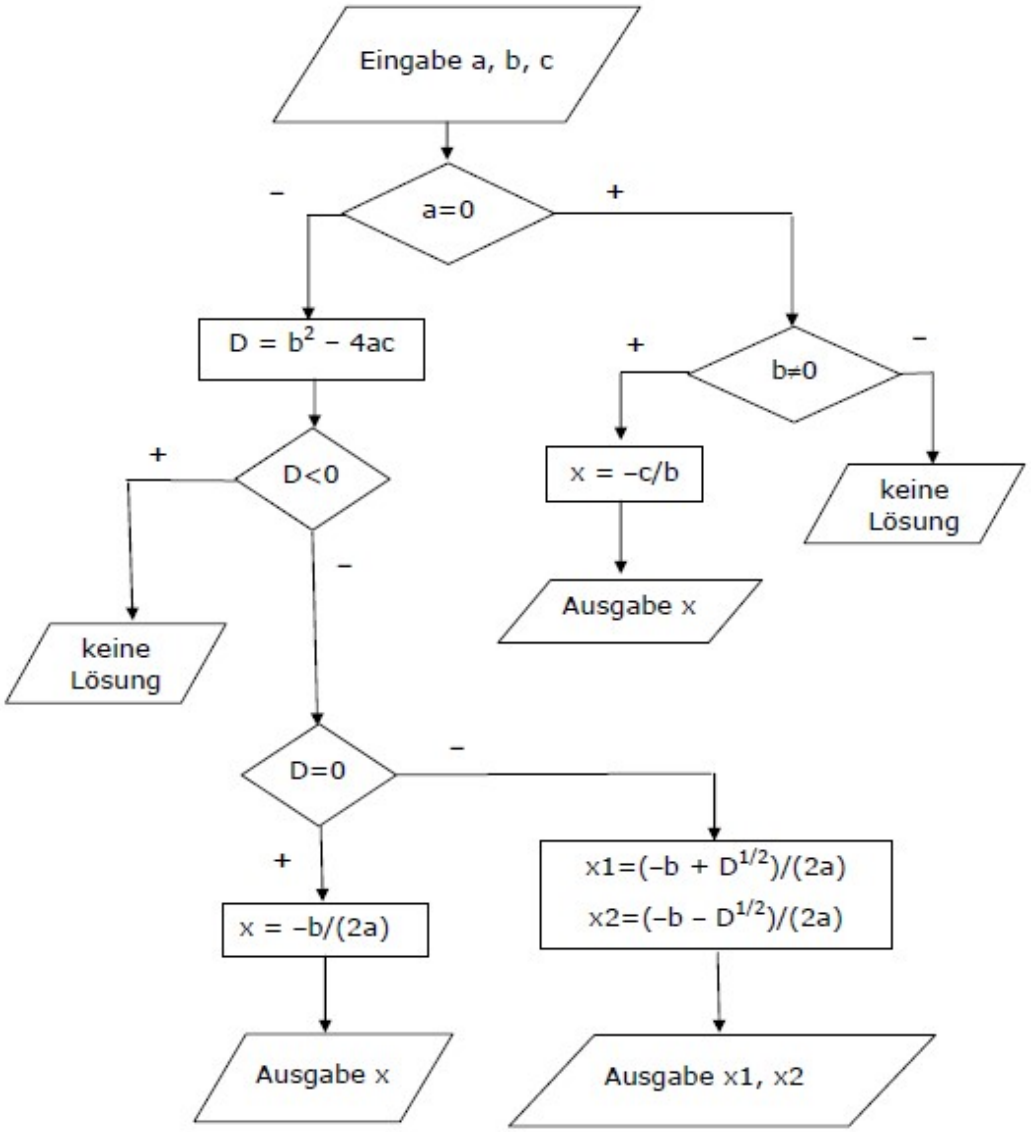


2. Verzweigte Algorithmen

Ein Algorithmus, bei dessen Abarbeitung unterschiedliche Anweisungsblöcke durchlaufen werden können, heißt verzweigter Algorithmus; dabei entscheidet die Abfrage einer Bedingung (in Form einer booleschen Variablen oder eines booleschen Ausdrucks) darüber, welcher Zweig durchlaufen wird.

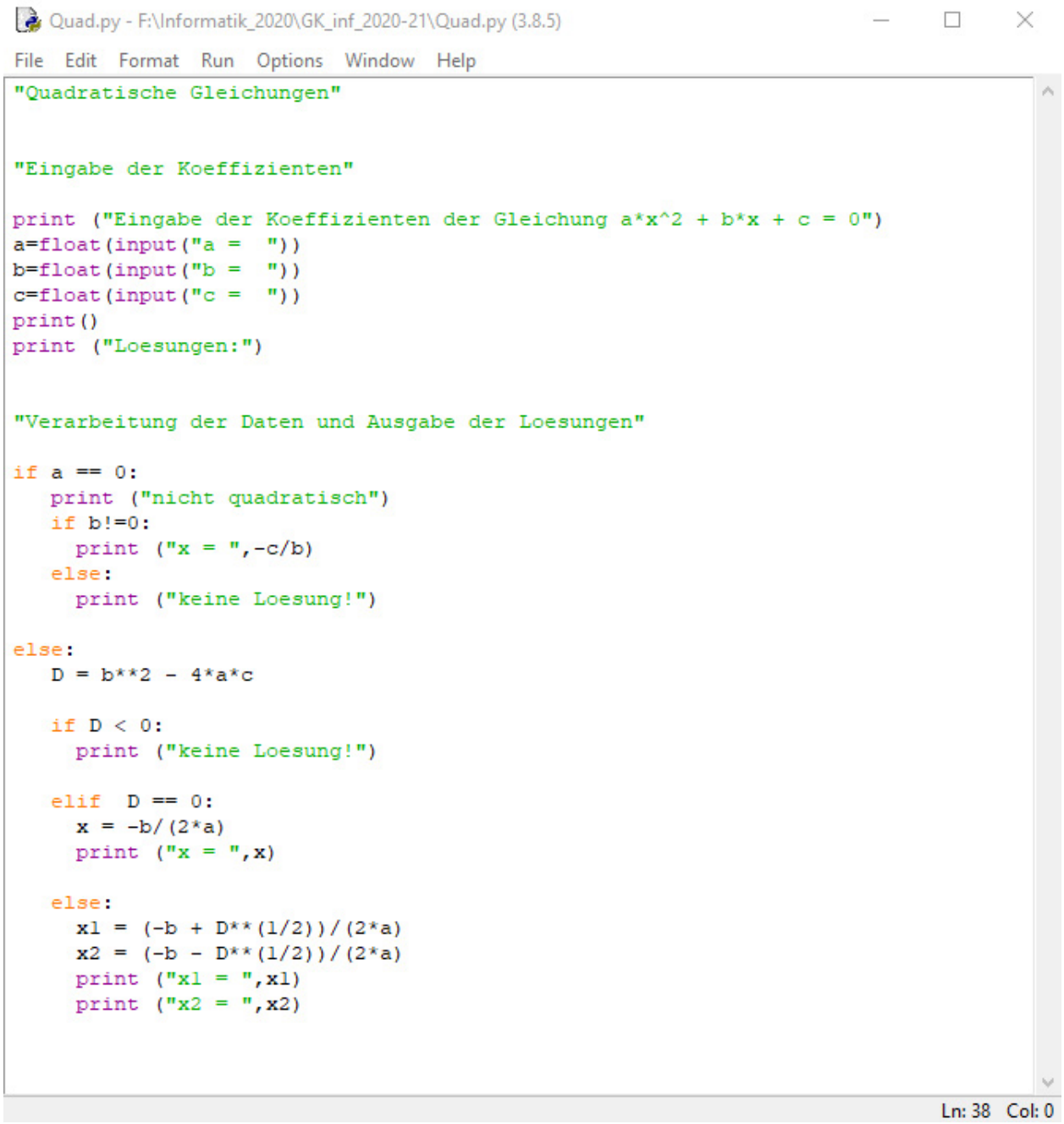
Beispiel:
Der Algorithmus „**QuadEquation**", der die Lösungsmenge einer quadratischen Gleichung bestimmt.

Der Ablauf ergibt sich aus einem Flußdiagramm oder einem Struktogramm (bei letzterem fehlt die Abfrage b≠0):



<table>
<tr><td colspan="4">Eingabe a, b, c</td></tr>
<tr><td>+</td><td colspan="3">a=0 –</td></tr>
<tr><td>Ausgabe
"nicht quadratisch"</td><td colspan="3">D:=b*b - 4*a*c</td></tr>
<tr><td></td><td colspan="3">+ D<0 –</td></tr>
<tr><td></td><td>Ausgabe
"keine Lösung"</td><td colspan="2">+ D=0 –</td></tr>
<tr><td></td><td></td><td>x:= -b/(2*a)</td><td>x1:=(-b + sqrt(D))/(2*a)</td></tr>
<tr><td></td><td></td><td>Ausgabe
x</td><td>x2:=(-b - sqrt(D))/(2*a)
Ausgabe
x1; x2</td></tr>
</table>

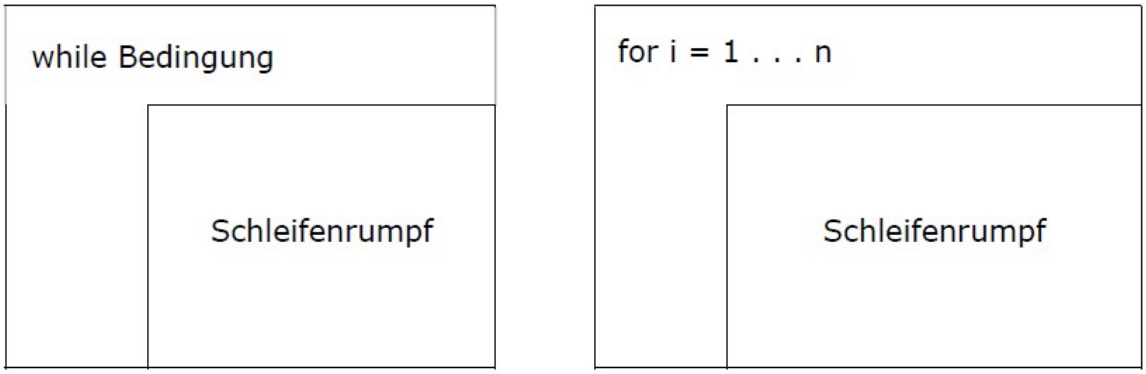
Quelltext des Algorithmus **QuadEquation** in Python codiert:



3. Schleifen

Soll ein Anweisungsblock innerhalb eines Algorithmus mehrmals durchlaufen werden, sprechen wir von einer Schleife; der wiederholt durchlaufene Anweisungsblock heißt auch Schleifenrumpf. Wenn die Anzahl der Durchläufe einer Schleife a priori (von vorneherein) feststeht, läßt sich eine **for**- oder **while**-**Schleife** verwenden; hat z. B. die Abfrage einer Bedingung (in Form eines Booleschen Ausdrucks) innerhalb des Schleifenrumpfs Einfluß auf die Anzahl der Durchläufe, kommt nur die **while-Schleife** in Frage.

Struktogramme:



Der Algorithmus „**Quadratzahlen**" gibt die Quadrate der ganzen Zahlen aus dem Intervall [a, b] aus:

Quellcode in Python, realisiert mit einer while-Schleife:

*Quadratzahlen_while-Schleife.py - F:\Informatik_2020\GK_inf_202...

File Edit Format Run Options Window Help

```
# Tabelle Quadratzahlen (while-Schleife)
# i = Schleifenindex
# a = kleinste ganze Zahl, deren Quadrat berechnet wird
# b = grösste ganze Zahl, deren Quadrat berechnet wird

print('In welchem Bereich soll das Quadrat berechnet werden?')
a=int(input('untere Grenze = '))
b=int(input('obere Grenze  = '))
print()

# Initialisierung des Schleifenindicis i
i=a

# while-loop
while i<=b:
    print(i,'^2 = ',i**2)
    i=i+1
    # alternativ: i += i
```

Ln: 19 Col: 0

Quellcode in Python, realisiert mit einer for-Schleife:

*Quadratzahlen_for-Schleife.py - F:/Informatik_2020/GK_inf_2020-...

File Edit Format Run Options Window Help

```
# Tabelle Quadratzahlen (for-Schleife)
# i = Schleifenindex
# a = kleinste ganze Zahl, deren Quadrat berechnet wird
# b = grösste ganze Zahl, deren Quadrat berechnet wird

print('In welchem Bereich soll das Quadrat berechnet werden?')
a=int(input('untere Grenze = '))
b=int(input('obere Grenze  = '))
print()


# for-loop
for i in range(a,b+1):
    print(i,'^2 = ',i**2)
```

Ln: 11 Col: 0

Ausgabe der Quadratzahlen:

Python 3.8.6 Shell

File Edit Shell Debug Options Window Help

```
Python 3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53
) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for m
ore information.
>>>
= RESTART: F:\Informatik_2020\GK_inf_2020-21\MSS12\Quadr
atzahlen_while-Schleife.py
In welchem Bereich soll das Quadrat berechnet werden?
untere Grenze = 11
obere Grenze  = 19

11 ^2 =  121
12 ^2 =  144
13 ^2 =  169
14 ^2 =  196
15 ^2 =  225
16 ^2 =  256
17 ^2 =  289
18 ^2 =  324
19 ^2 =  361
>>>
```

Ln: 18 Col: 4

Quadratwurzel aus einer positiven reellen Zahl

Der Algorithmus „**Wurzelberechnung**" approximiert **√a** für eine positive reelle Zahl **a**. Die Iteration bricht ab, sobald der Abstand zweier aufeinanderfolgender Folgenglieder kleiner als eine einzugebende Fehlerschranke **d** wird.

Der Abbruch erfolgt, sobald die Boolesche Variable **condition** innerhalb des Schleifenrumpfs den Wert **False** erhält, was eintritt, wenn **abs(y - z)** kleiner als **d** wird.

Anmerkung:
Der hier vorgestellte Algorithmus stützt sich darauf, daß die Folge $\{x_i\}$ *mit* $x_{i+1} = \frac{1}{2}(x_i + \frac{a}{x_i})$, $x_0 = a$, *gegen* $\sqrt{a}$ *konvergiert; dies sei hier ohne Beweis und ohne nähere Begründung mitgeteilt.*

*sqrt_while-loop_mit_abbruchbedingung.py - F:/Informatik_2020/...

File Edit Format Run Options Window Help

```
" Wurzelberechnung "

# Eingabe
print('Eingabe des Radikanden:')
a = float (input('a = '))
# Die Iteration bricht ab, sobald der Abstand
# zweier aufeinanderfolgender Iterationen kleiner als d wird
print('Eingabe Fehlerschranke d:')
d = float (input('d = '))
print()


# Verarbeitung und Ausgabe
condition = True    # Initialisierung der Bedingung
x = a               # Startwert; alternativ: x = 1
while condition:
    z = x
    x = 0.5*(x + a/x)
    y = x
    print(x)
    condition = (abs(y - z) >= d)
```

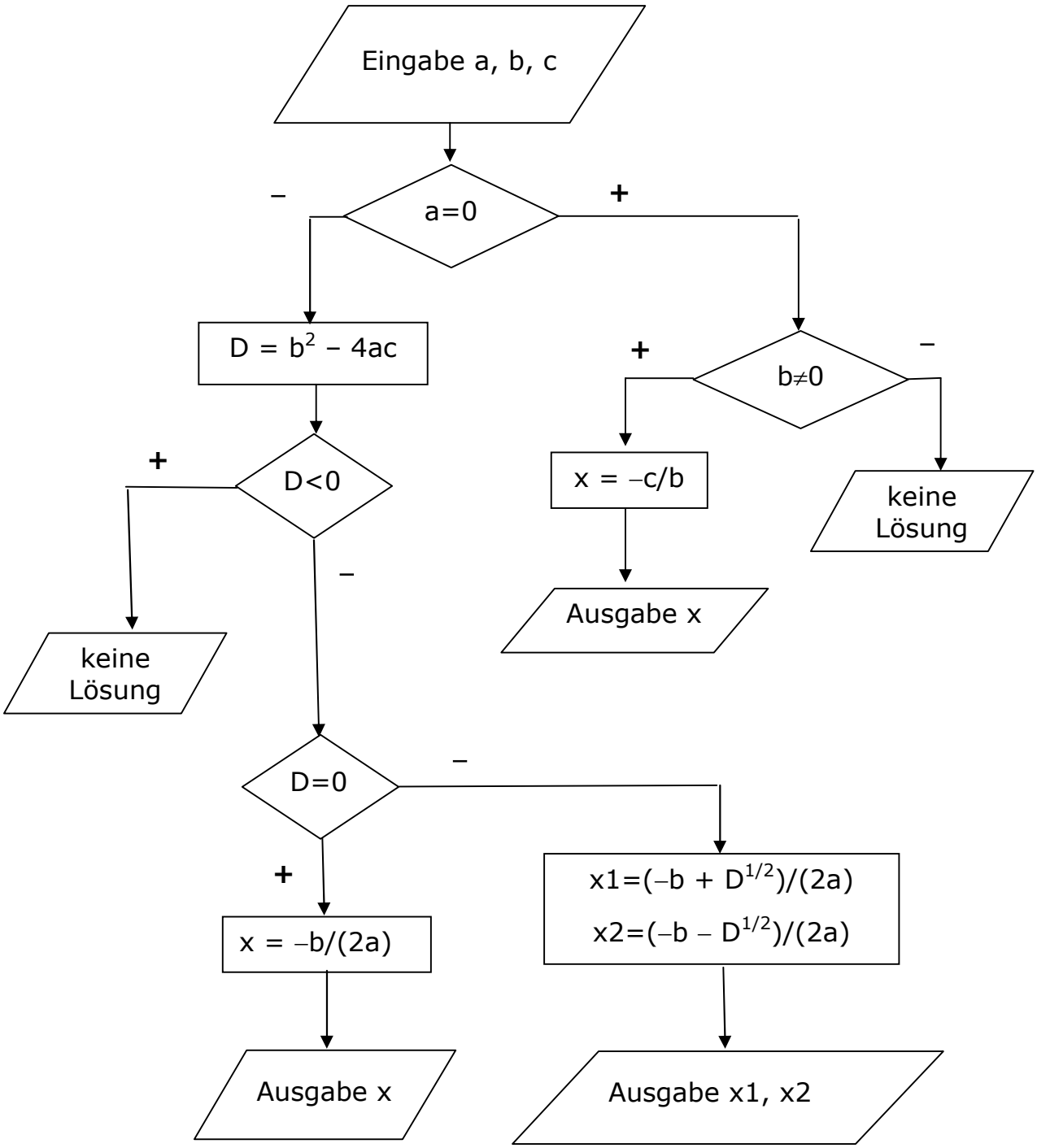
Ln: 19 Col: 8


Python 3.8.6 Shell
File Edit Shell Debug Options Window Help
Python 3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: F:/Informatik_2020/GK_inf_2020-21/MSS12/sqrt_while-loop_mit_abbruchbedingung.py
Eingabe des Radikanden:
a = 85
Eingabe Fehlerschranke d:
d = 0.0001
43.0
22.488372093023255
13.134051610110387
9.802889441169137
9.236901144433745
9.219560764417094
9.21954445730731
>>>
= RESTART: F:/Informatik_2020/GK_inf_2020-21/MSS12/sqrt_while-loop_mit_abbruchbedingung.py
Eingabe des Radikanden:
a = 85
Eingabe Fehlerschranke d:
d = 0.0000000000001
43.0
22.488372093023255
13.134051610110387
9.802889441169137
9.236901144433745
9.219560764417094
9.21954445730731
9.219544457292887
9.219544457292887
>>>
= RESTART: F:/Informatik_2020/GK_inf_2020-21/MSS12/sqrt_while-loop_mit_abbruchbedingung.py
Eingabe des Radikanden:
a = 25
Eingabe Fehlerschranke d:
d = 0.00000000000001
13.0
7.461538461538462
5.406026962727994
5.015247601944898
5.000023178253949
5.000000000053722
5.0
5.0
>>>
Ln: 48 Col: 4

Algorithmus **QUADRATISCHE GLEICHUNGEN**

Spezifikation:
Nach Eingabe der Koeffizienten a, b, c der allgemeinen quadratischen Gleichung **$ax^2 + bx + c = 0$** ermittelt der Algorithmus die Lösungsmenge und gibt diese aus.

Flußdiagramm:



05.10.2020

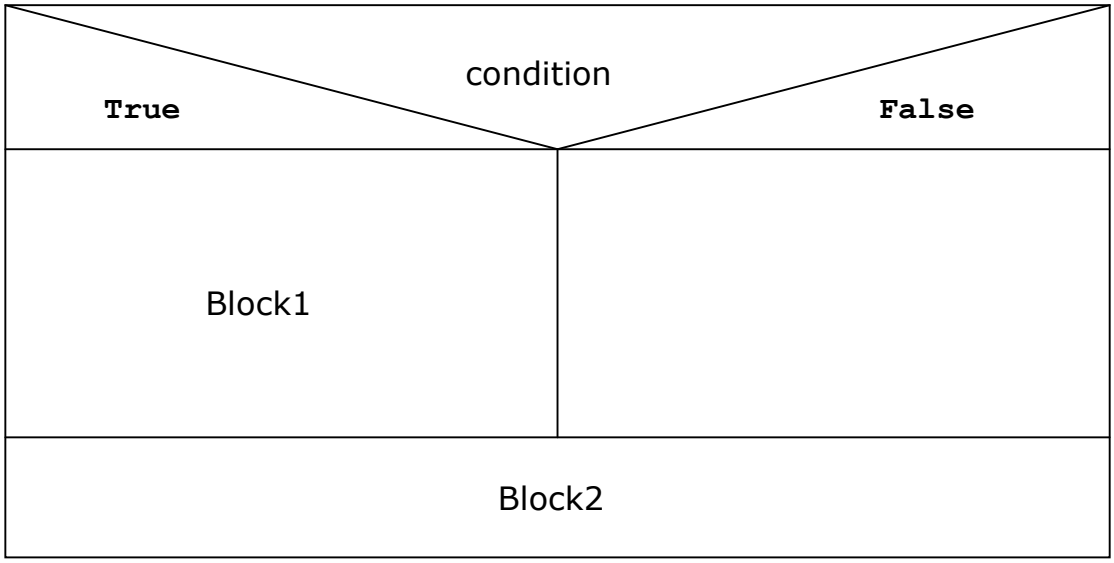
Verzweigte Algorithmen

Definition: Ein **Anweisungsblock** besteht aus einer Folge zusammengehörender Anweisungen, die nacheinander ausgeführt werden.
Ein Anweisungsblock, der innerhalb einer Schleife wiederholt wird, heißt **Schleifenrumpf**.
Den zu einer Funktion gehörenden Anweisungsblock nennen wir auch **Funktionsrumpf**.

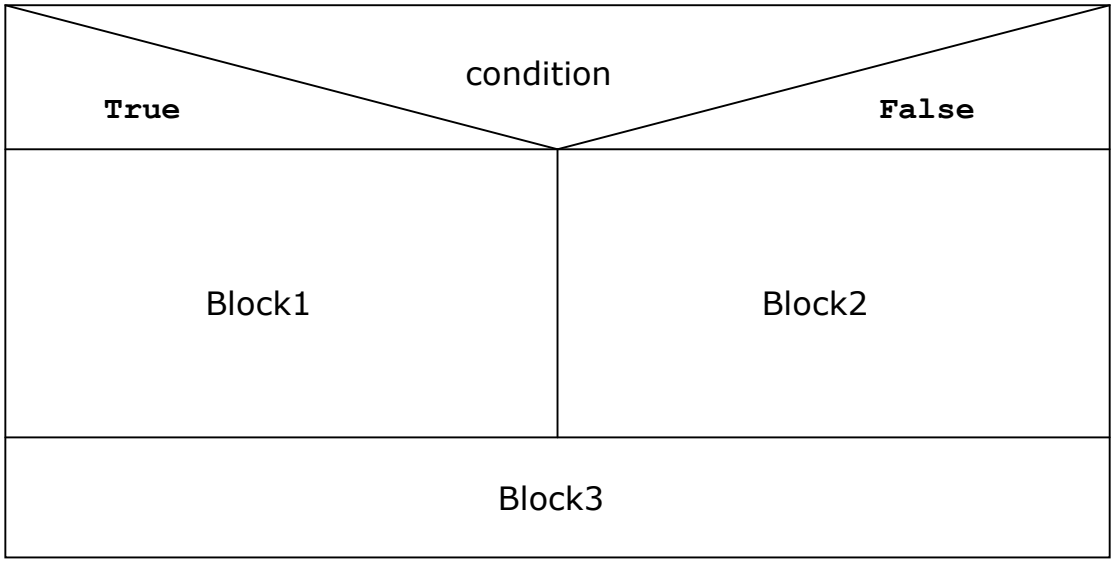
Bemerkungen:
- Anweisungsblöcke können auch ineinander verschachtelt sein.
- In Python wird ein Anweisungsblock durch Einrücken des Programmtextes gekennzeichnet.

Im folgenden verstehen wir unter ***condition*** einen Booleschen Term (der auch nur aus einer Booleschen Variablen bestehen kann), der die Werte **`True`** oder **`False`** annimmt. In Struktogrammen kennzeichnen wir **`True`** auch durch , **+** ', **`False`** durch , – '.

Einseitige Auswahl



Zweiseitige Auswahl

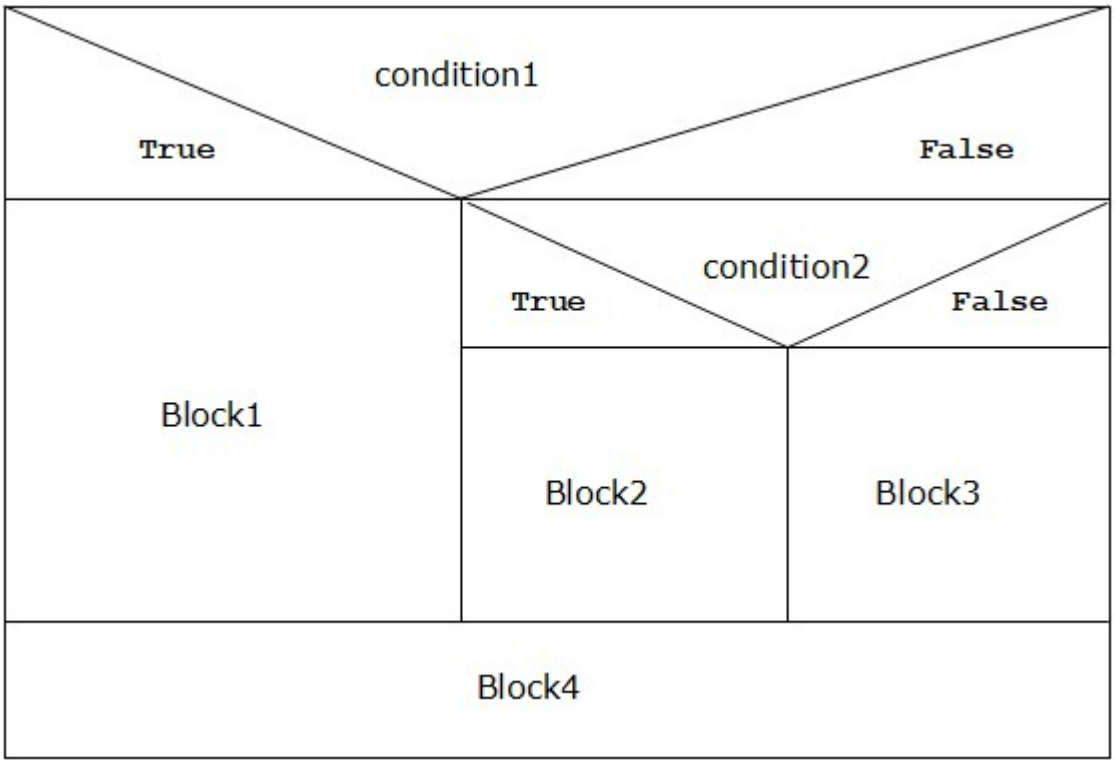


Formulierung in Python:

```
if condition:
    Block1
Block2
```

```
if condition:
    Block1
else:
    Block2
Block3
```

Mehrstufige Auswahl



Formulierung in Python:

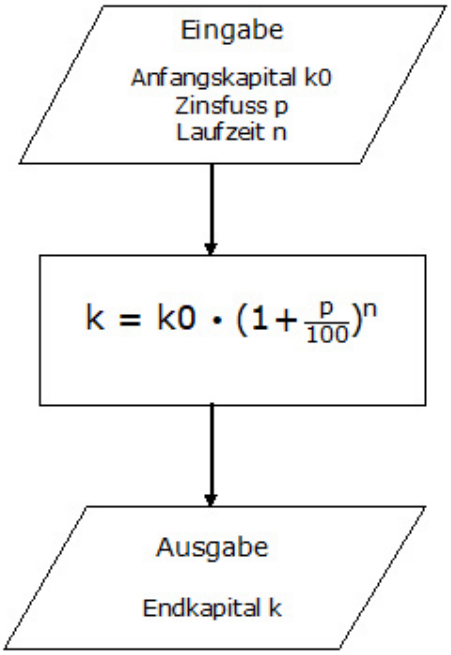
```
if condition1:
    Block1
else:
   if condition2:
       Block2
   else:
       Block3
Block4
```

```
if condition1:
    Block1
elif condition2:
    Block2
else:
    Block3
Block4
```

Aufgabenblatt Nr. 1 inf12 27.10.2020

1. **Zinseszins** (linearer Algorithmus, ohne Verzweigungen)

   Flußdiagramm:



   Wenn ein Anfangskapital **k0** zu einem jährlichen Zinssatz **p** % über einen Zeitraum von **n** Jahren mit Zinseszins angelegt wird (der Zinsbetrag wird also am Ende jeden Jahres dem zu verzinsenden Kapital zugeschlagen), ermittelt der Algorithmus „Zinseszins" das Endkapital **k** nach **n** Jahren.

   Schreibe den durch nebenstehendes Flußdiagramm gegebenen Algorithmus als Python-Programm und teste es.

2. **Mobilfunkrechung**
   (Verzweigter Algorithmus)

   Der Betreiber eines Mobilfunknetzes hat folgende Tarifgestaltung:

   Monatliche Grundgebühr (einschließlich 100 Gesprächsminuten): 20 €;
   für die nächsten, über 100 Minuten hinausgehenden 200 Minuten sind 5 ct je Minute zu entrichten; jede weitere Minute kostet 4 ct.

   Schreibe einen Algorithmus als

   a) Struktogramm,
   b) Pythonprogramm,

   um nach Eingabe der Anzahl **x** der monatlichen Gesprächsminuten den Rechnungsbetrag **b** zu bestimmen.

3. **n-te Wurzel aus einer positiven reellen Zahl a**
   (Algorithmus mit Wiederholung, also mit Iteration (lat. „iterare", wiederholen))

   Die Folge ***{x_i}*** *mit*

   $x_{i+1} = \frac{1}{n}\left((n-1) \cdot x_i + \frac{a}{x_i^{n-1}}\right)$, $x_0 = a$, *konvergiert gegen* $\sqrt[n]{a}$;

   dies sei hier ohne Beweis und ohne nähere Begründung mitgeteilt.

Schreibe und teste ein Python-Programm, welches nach Eingabe des Radikanden **a**, der Ordnung **n** und der Fehlerschranke **d** (größter Abstand zwischen dem letzten und dem vorletzten Folgenglied) die **n**-te Wurzel aus **a** bestimmt.

Hinweis: Man orientiere sich am Algorithmus „Quadratwurzel" auf S. 6 der Zusammenfassung vom 08.10.2020.

4. **Summe der Zahlen 1, 2, 3, ... , n**
(Algorithmus mit Wiederholung)

Der Algorithmus **Sum** ermittelt nach Eingabe der natürlichen Zahl **n** die Summe der Zahlen 1, 2, ... , n.

Schreibe und teste jeweils ein Python-Programm unter Verwendung einer
a) for-Schleife,
b) while-Schleife.

Prinzipien zur Erstellung eines Programms

Imperativer Ansatz

Der Quellcode (formuliert in einer Programmiersprache, z. B. Pascal oder Python) besteht aus einer Folge von ausführbaren Anweisungen, die in der vorgegebenen Reihenfolge abgearbeitet werden.
In Maschinensprache (Assembler) geschriebene Programme verfolgen stets den imperativen Ansatz, die elementaren (Maschinen-)Befehle werden nacheinander ausgeführt.

Wesentliche Kontrollstruktur: **Iterationen** (for-, while-Schleife)

Funktionaler Ansatz

Der Quellcode bedient sich mathematischer Funktionen, durch die ein Algorithmus beschrieben wird.

Wesentliche Kontrollstruktur: **Rekursion**

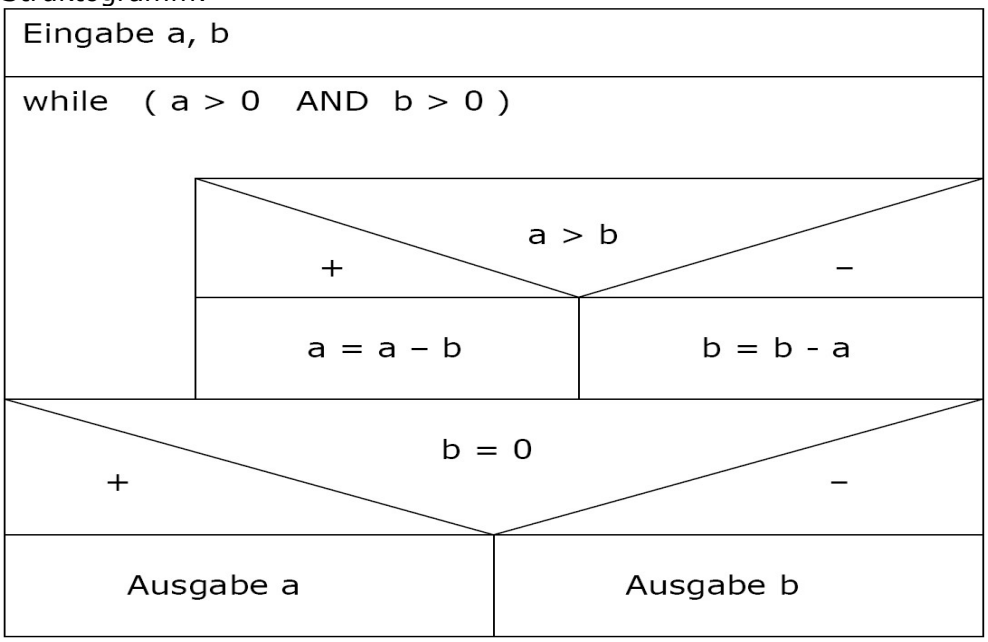
Definition:
*Eine Prozedur (Teilprogramm) oder eine Funktion heißt **rekursiv**, wenn ihr Anweisungsteil mindestens einen Aufruf von sich selbst enthält.*

Bei beiden Ansätzen ist durch eine Abbruchbedingung sicherzustellen, daß der Algorithmus terminiert, also nach endlich vielen Schritten beendet wird und zu einem Ergebnis führt.

Beispiel 1
Der Algorithmus **ggT** (**g**rößter **g**emeinsamer **T**eiler)

Nach Eingabe zweier natürlicher Zahlen a und b bestimmt ggT die größte ganze Zahl, durch die sich a und b jeweils ohne Rest teilen lassen.

a) **Imperativer Ansatz**, formuliert als **iterativer Algorithmus**
(„Euklidischer Algorithmus")
Struktogramm:



b) **Funktionaler Ansatz**, formuliert als **rekursiv definierte Funktion**

Die Funktion $(a, b) \rightarrow ggT(a,b)$ läßt sich rekursiv definieren:

Rekursionsanfang: $ggT(a,a) = a$

Rekursionsvorschrift: $ggT(a,b) = ggT(a-b, b)$, falls $a > b$

$ggT(a,b) = ggT(a, b-a)$, falls $b > a$

Aufgabe:
Realisiere den Algorithmus ggT als iteratives und als rekursives Python-Programm; vergleiche die Laufzeiten.

Beispiel 2
Die Funktion **„Fakultät"** (englisch: Factorial)

Die Funktion **fact** ordent jeder natürlichen Zahl **n** das Produkt
n! = 1 · 2 · . . . . . · n zu; definitionsgemäß gilt: 0! = 1.

a) **Imperativer Ansatz**

Formuliere den Algorithmus iterativ (for- oder while-Schleife) als Struktogramm und als Python-Programm

b) **Funktionaler Ansatz**

Die Funktion $n \rightarrow fact(n)$ läßt sich rekursiv definieren:

Rekursionsanfang: $fact(0) = 1$

Rekursionsvorschrift: $fact(n) = n \cdot fact(n-1)$, falls $n > 0$

Formuliere die Funktion **fact** als rekursives Python-Programm!

Beispiel 3
Die **Hofstadter-Funktion**

Die Funktion **hof** ist rekursiv definiert, $n \in \{1, 2, 3, ... \}$:

Rekursionsanfang: $hof(1) = 1$
$hof(2) = 1$

Rekursionsvorschrift: $hof(n) = hof(n - hof(n - 1)) + hof(n - hof(n - 2))$, $n>2$

Aufgabe:
Codiere den Algorithmus hofstadter
a) rekursiv,
b) iterativ
jeweils in Python; vergleiche insbesondere die Laufzeiten!

Hinweis zu b): Definiere in geeigneter Weise ein array (Feld), in dem bereits berechnete Funktionswerte gespeichert werden.

Arbeitsauftrag GK inf für 17.11.2020

1.) Zu **Aufgabe 4**, Aufgabenblatt Nr. 1 vom 27.10.2020
(Summe der ganzen Zahlen 1, ... , n)

Struktogramm (10.11.2020):

Eingabe n
i = 1
summe = 0
while i <= n
summe = summe + i
i = i +1
Ausgabe summe

← Initialisierung des Schleifen-index i

a) Vervollständige folgende **Trace**-Tabelle für n=6 (SD = Schleifendurchlauf):

	n	i	summe	i<=n
vor dem 1. SD	6	1	0	True

b) Schreibe und teste ein **Python-Programm,** welches nach Eingabe einer natürlichen Zahl n die Summe 1 + ... + n ermittelt!

2.) Zu **Beispiel 2** des am 27.10.2020 ausgeteilten Papers
„Funktionaler_und_Imperativer Ansatz.pdf"
Die **n!-Funktion**; lies: „n-Fakultät"

Definition: n! = Produkt der ganzen Zahlen 1, 2, ... , n
= 1 · 2 · 3 · . . . . . · n

Beachte: 0! = 1 (daß diese Definition Sinn macht, werdet ihr noch im Mathematikunterricht kennenlernen.)

a) Erstelle ein **Struktogramm** (mit while-Schleife) für den Algorithmus, der nach Eingabe einer ganzen Zahl n, $n \geq 0$, die Fakultät von n bestimmt.
b) Fertige eine **Trace**-Tabelle an für n = 5 (entsprechend obiger Tabelle).
c) Schreibe und teste ein **Python-Programm**, welches nach Eingabe einer ganzen, nicht negativen Zahl n die Fakultät von n berechnet!

3.) Freiwillige **Zusatzaufgabe**: Formuliere die Programme aus 2.) und 3.) jeweils mit einer for-Schleife statt einer while-Schleife!

5. **Summe ungerader Zahlen**
   Sei n eine ungerade ganze Zahl; gesucht ist die Summe der ungeraden Zahlen 1, ... , n.
   Konzipiere diesen Algorithmus als Struktogramm und codiere ihn in Python; teste das Programm. Was fällt auf?

6. Die **Ackermann-Funktion**
   Für m, n $\in \mathbb{N}_0$ ist die Ackermann-Funktion $f : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$ wie folgt definiert:

   1. Rekursionsanfang:

      (1) f(0,n) = n+1

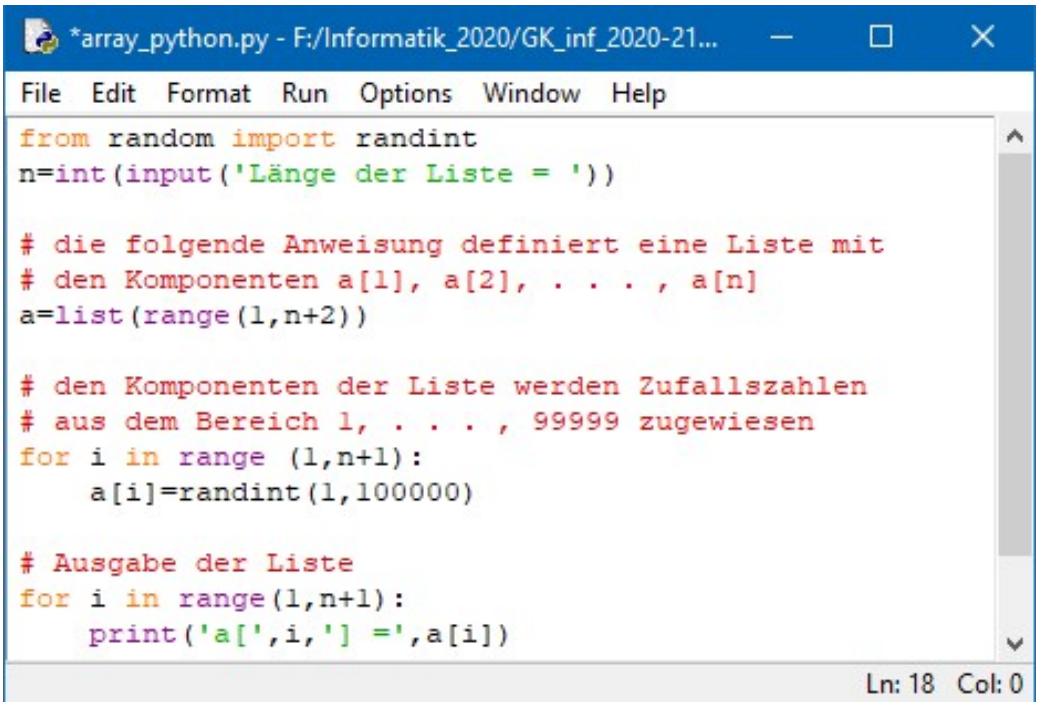
   2. Rekursionsvorschrift:

      (2) f(m,0) = f(m-1,1)
      (3) f(m,n) = f(m-1, f(m,n-1))

   a) Man erhält:
   f(0,0) = 1
   f(0,1) = 2
   f(0,2) = 3
   f(1,0) = f(0,1) = 2
   Berechne f(2,0); f(1,1); f(1,2); f(3,0) .

   b) Schreibe den Algorithmus zur Berechnung der Ackermann-Funktion als Python-Programm mit rekursivem Funktionsaufruf.
   Berechne f(3,7); f(3,8); f(4,1) ; f(3,15) ; f(4,3)
   Bemerkung: Die Ackermann-Funktion ist eine berechenbare Funktion, allerdings übersteigt deren ungeheure Rekursionstiefe sehr schnell die Möglichkeiten jedes auch noch so leistungsfähigen Computers!

7. Die Datenstruktur **„array"** läßt sich in Python als **Liste** mit den Komponenten a[1], a[2], ... , a[n] z. B. wie folgt realisieren:



   Formuliere ein Struktogramm und erweitere oben stehendes Python-Programm so, daß das größte (kleinste) Element der Liste in der ersten Komponente a[1] abgespeichert ist und der vorherige Inhalt von a[1] an derjenigen Stelle steht, von der das größte Element genommen wurde.

Arbeitsauftrag GK inf12 für 24.11.2020

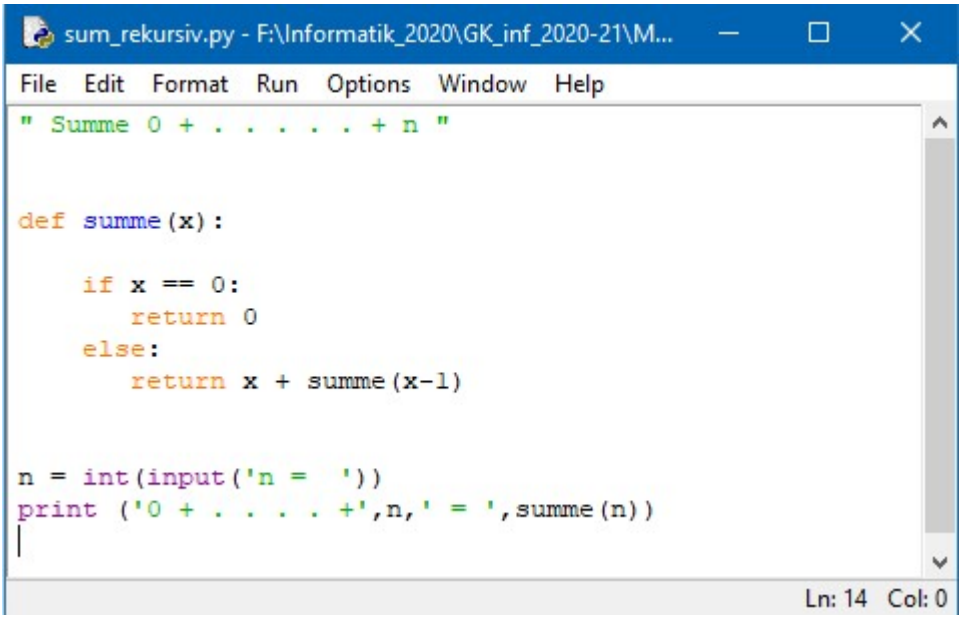
In höheren Programmiersprachen (wie Python) ist die Möglichkeit implementiert, Funktionen, auch rekursiv formulierte Funktionen, zu definieren. Dabei verstehen wir unter einer Funktion ein Unterprogramm (Prozedur), welches nach der Übergabe von Daten einen Funktionswert an das aufrufende Programm zurückgibt. Der zur Funktion gehörende Anweisungsblock heißt auch Funktionsrumpf (in Python wird der Funktionsrumpf durch Einrücken des Programmtextes kenntlich gemacht). Eine Funktion, deren Funktionsrumpf mindestens einen Aufruf ihrer selbst enthält, heißt rekursiv (lat. recurrere, zurücklaufen).

Die Funktion **summe** (siehe Arbeitsauftrag für 17.11.2020), die einer natürlichen Zahl **n** mit n ∈ {0, 1, 2, . . . } die Summe
0 + ... + n zuordnet, läßt sich rekursiv wie folgt definieren:

Rekursionsanfang: **summe(0) = 0**

Rekursionsvorschrift: **summe(n) = n + summe(n – 1)** falls n ≥ 1

Realisierung von **summe** in Python:



Erläuterungen:

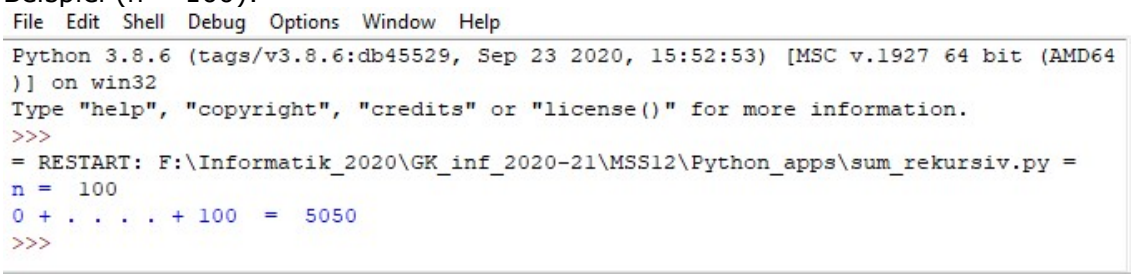
def summe(x): Funktionskopf;
summe = Name der Funktion
x = lokale (nur innerhalb der Funktion verfügbare) Variable
Nach dem Doppelpunkt folgt der durch Einrücken kenntlich gemachte Funktionsrumpf.

return mit **return** wird der berechnete Funktionswert an das aufrufende Programm übergeben

Der Aufruf **summe(n)** (hier: innerhalb der **print**-Anweisung) bewirkt:

- Der aktuelle Wert der Variablen **n** wird der lokalen, nur innerhalb der Funktion verfügbaren Variablen **x** zugewiesen
- Nach der (hier rekursiv erfolgenden) Berechnung wird der Funktionswert mit **return** zurückgegeben.

Beispiel (n = 100):



```
File  Edit  Shell  Debug  Options  Window  Help
Python 3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53) [MSC v.1927 64 bit (AMD64
)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: F:\Informatik_2020\GK_inf_2020-21\MSS12\Python_apps\sum_rekursiv.py =
n =  100
0 + . . . . + 100  =  5050
>>>
```

Die rekursive Berechnung von **summe(6) = s(6)** läßt sich wie folgt verdeutlichen:

```
s(6) = 6 + s(5)
     = 6 + (5 + s(4))
     = 6 + (5 + (4 + s(3)))
     = 6 + (5 + (4 + (3 + s(2))))
     = 6 + (5 + (4 + (3 + (2 + s(1)))))
     = 6 + (5 + (4 + (3 + (2 + (1 + s(0))))))
```

mit **s(0)** ist der Rekursionsanfang erreicht, die Rekursion bricht ab.

Arbeitsaufträge für 24.11.2020:

1.) Erstelle den Programmtext für die rekursive Berechnung von **summe(n)**; man orientiere sich an dem obenstehenden screenshot.

2.) Teste das Programm sowohl als iterativ (gemäß Ziffer 1 aus Arbeitsauftrag für 17.11.2020) als auch als rekursiv definierten Algorithmus für unterschiedliche Werte von n; wähle auch n = 1000, 10000, 100000, 1000000. Was fällt auf?

3.) Die Fakultätsfunktion (eng.: factorial) läßt sich rekursiv definieren (vgl. das Paper „Funktionaler_und_Imperativer_Ansatz" vom 27.10.2020):

Rekursionsanfang: **fact(0) = 1**

Rekursionsvorschrift: **fact(n) = n · fact(n – 1)** , falls n > 0

Schreibe und teste ein Python-Programm, um die Fakultätsfunktion rekursiv zu berechnen; vergleiche mit dem iterativ formulierten Algorithmus gemäß Ziffer 2 des Arbeitsauftrags für 17.11.2020.

4.) fakultativ: Aufgabe Nr. 5 aus Aufgabenblatt Nr. 2 vom 10.11.2020