Einführung in das Programm *Mathematica* Teil 3

(leicht bearbeitete Vorlage von T. Unseld und ergänzt mit Anwendungen zur Rekursion & Iteration)

If, Do und **While** sind die zentralen Begriffe, welche im 3.*Teil der Einführung in Mathematica* besprochen und an Beispielen zur Anwendung gebracht werden. Anwendungen, auf welche wir in ANALYSIS - Kapitel 5 *Folgen & Reihen* im Zusammenhang mit **Rekursionen & Iterationen** zurückgreifen werden. Weiter wird noch die Idee der **Module** und deren Anwendungsmöglichkeiten besprochen.

Zur Bearbeitung gilt auch hier wieder:
Dieses Skript steht dir als pdf zur Verfügung. Wie immer gilt, dass das einfache Durchlesen der Unterlagen (mit sehr grosser Wahrscheinlichkeit) nicht ausreicht, *Mathematica* und seine Anwendungen in dem Masse kennenzulernen, das ein sicherer Umgang mit diesem Programm erlernt werden kann.
Ich empfehle daher *dringend*, die Aufgaben nachzurechnen. Vorteilhaft in einem begleitenden nb-file, welches du im Textstyle mit persönlichen Bemerkungen und Erkenntnisse ergänzen kannst. Auf diese Weise wirst du am Schluss ein weiteres auf *dich persönlich* zugeschnittenes Dokument mit den *für dich wichtigen* Erklärungen haben, welches *du* nach Bedarf laufend ergänzen und neuen Bedürnissen anpassen kannst.

Ergänzen werden wir diesen Teil mit klassischen Anwendungen aus dem Bereich der **Näherungsverfahren** und schliessen mit einem (freiwilligen) Exkurs in die **Rechengenauigkeit** ab.

Wir verwenden als Einstieg die Repetition zur **Definition eigener Funktionen:**

Ich möchte zum Beispiel einen Befehl kreieren, der jeden Wert x quadriert. Ich nenne diesen Befehl "quadrieren":

```
quadrieren[x_] := x^2
```

Obwohl die obige Zelle an den Kernel geschickt wurde, erscheint keine Outputzelle, weil eine sogenannte **verzögerte Zuordnung ": ="** benutzt wurde. Das macht auch Sinn, da x nicht x^2 zugeordnet werden soll, sondern der Befehl "quadrieren" soll erst dann zum Zug kommen, wenn **später** für x_ Werte eingegeben werden:

```
quadrieren[3]
{quadrieren[-2], quadrieren[Pi], quadrieren[2 / 3]}
```

9

$\left\{4, \pi^2, \frac{4}{9}\right\}$

"x_" nennt man Muster oder **Pattern** und hat nichts mit der Variablen x zu tun:
Es wird **irgendein Ausdruck x_** , der als Argument in den Befehl eingegeben wird, quadriert:

```
{quadrieren[a], quadrieren[2 a], quadrieren[3 a^3 + 1], quadrieren["Velo"]}
```

$\left\{a^2, 4\,a^2, \left(1 + 3\,a^3\right)^2, \text{Velo}^2\right\}$

Wie man oben sieht, wurde sogar der Text "Velo" quadriert. "Velo" ist also kein Variablenname, sondern wird wegen den Anführungs- und Schlusszeichen von *Mathematica* nur als Text behandelt. Eine **Zeichenkette** wie "Hallo0815Velo" nennt man in der Informatik einen **String**.

Zur Veranschaulichung des obigen neuen Vorgehens möchte ich nun einen zweiten eigenen Befehl (eigene Funktion) definieren, der bei Eingabe des Radius des Kreises den Flächeninhalt berechnet. Mein Befehl, meine Funktion soll den Namen “flaeche” haben:

```
flaeche[r_] := r^2 * Pi
```

Mit dem obigen Befehl kann man für den Pattern r_ irgendeinen Radius eingeben, und es wird die entsprechende Fläche mit der Variablen r berechnet:

```
flaeche[3]
flaeche[2.3]
```

$9\,\pi$

16.619

Nun möchte ich meinen Befehl so verbessern, dass er immer eine numerisch gerundete Fläche berechnet, weil ich mir oben unter $9\,\pi$ als Flächeinhalt nichts vostellen kann. Dieser neue Befehl, diese neue Funktion soll nur noch den Namen "f" haben:

```
f[r_] := N[r^2 Pi, 4]
```

Die oben definierte Funktion "f" sollte die "Fläche auf 4 wesentliche Ziffern gerundet" liefern:

```
f[3]
```

28.27

Eine **eigene Funktion** (einen eigenen Befehl) mit dem **Namen "mike"**, die jedem t zum Beispiel den Wert $2 \cdot t - 3$ zuordnet, definiert man in *Mathematica* nach folgender Struktur:

```
mike[t_] := 2 * t - 3
```

Benutze bei Funktionsdefinitionen die **verzögerte Zuordnung ":="** und den **Pattern "t_"** links vom ":="!
Denke daran, dass rechts vom ":=" kein Pattern "t_" mehr steht, sondern nur noch "t"!

Benutze **für deine eigenen Funktionsnamen nur Kleinbuchstaben**, damit sie sich von den *Mathematica*-Befehlen, die immer mit einem Grossbuchstaben beginnen, unterscheiden!

Die vor dem Merksatz definierte Funktion "f" berechnet ja die Fläche eines Kreises mit Radius r. Leider meldet sie bei Eingabe von sinnlosen Radien auch Resultate:

```
f[-3]
f["0815"]
```

28.27

$3.142\,0815^2$

Wenn man nun will, dass man sinnlose negative Seiten nicht eingeben darf, kann man "f" z. B. mit dem **If-Befehl** verbessern. Er hat die Struktur **If[Bedingung , Befehle bei wahrer Bedingung, Befehle sonst]**:

```
z = 4;
If[z < 10, "kleiner als 10", "grösser als 10"]
```

```
kleiner als 10
```

Oben wurde nur der String "kleiner als 10" ausgegeben, weil die "Bedingung z<10" wahr war, da z = 4 ist.
Unten sollte der zweite String ausgegeben werden, weil die "Bedingung z<10" falsch ist, da jetzt z = 13 ist:

```
z = 13;
If[z < 10, "kleiner als 10", "grösser als 10"]
```

```
grösser als 10
```

Damit kann man nun die Funktion f verbessern:

```
flaeche2[r_] := If[r > 0, r^2 Pi, "Nur positive Radien eingeben!"]
```

Die so definierte Funktion sollte die Fläche bei positiven Radien berechnen und reklamieren, falls dem nicht so ist:

```
flaeche2[2.5]
flaeche2[-3]
```

```
19.635
```

```
Nur positive Radien eingeben!
```

Man kann auch **Befehle, Funktionen mit mehreren Argumenten** definieren:

```
produkt[a_, b_, c_] := a * b * c
produkt[2, 3, 4]
produkt[x^2, 2 x^3, 4 c]
```

```
24
```

$$8\,c\,x^5$$

Aufgabe 12:

a) Die Note einer Prüfung berechnet sich gemäss der Formel

$$\text{Note} = \frac{\text{erreichte Punktzahl}}{\text{maximale Punktzahl}} \cdot 5 + 1$$

Schreibe eine Funktion note, die für eine Prüfung, bei der es 14 Punkte gab, die Noten berechnet. Dabei soll die

berechnete Note auf zwei wesentliche Stellen gerundet werden. Teste deine Funktion mit ein paar Punkten.

b) Erweitere Deine Funktion so, dass nur genügende Noten berechnet werden und ungenügende Leistungen nicht

ausgerechnet, sondern durch eine passende Bemerkung kommentiert wird.

c) Bei den Exen ist die Maximalpunktzahl nicht immer gleich. Schreibe eine zweite Funktion note,

bei der als zweites Argument auch die Maximalpunktzahl übergeben werden kann. Teste deine Funktion mit

ein paar konkreten Punktzahlen.

Natürlich kann *Mathematica* viel mehr als die Algebra in diesem Kapitel. Wie im ersten Kapitel wiederum ein kleiner Vorgeschmack als letzte Aufgabe:

Aufgabe 14:

a) Schicke den unteren Befehl an den Kernel, und klicke auf den Pfeil in der Output-Zelle:

```
OpenerView[{(x + y)^4, (x + y)^4 // Expand}]
```

b) Schicke den unteren Befehl an den Kernel, und klicke ein paar Mal auf das Resultat in der Output-Zelle:

```
Toggler[(x + y)^6, {(x + y)^6, (x + y)^6 // Expand,
  "Ich hasse das Pascal-Dreieck!", "Na und!?", "Weitermachen!"}]
```

c) Schicke den unteren Befehl an den Kernel, und klicke ein paar Mal in die Graphik:

```
FlipView[Table[Plot[Sin[n x], {x, 0, 7},
   Ticks → {{Pi/2, Pi, 3 Pi/2, 2 Pi}, {-1, 1}}, PlotLabel → Sin[n x]], {n, 5}]]
```

und noch eine klasssische Aufgabe zum Abschluss der Repetition:

Aufgabe :

Wir betrachten die folgenden Funktionen:

f(x) = $e^{0.5x}$ und g(x) = cos(2x-0.5)

a) Stelle die Situation graphisch dar, mit einem vernünftig gewählten Defiitions- & Wertebereich und einer Legende.

b) Berechne die Schnittpunkte über [0, 2π]

Schleifen mit *Mathematica*

Schleifen sind ein wichtiges Instrument jeder Programmiersprache.
Als Beispiel wird i^2 für i =1, 2, 3, 4, 5 mit dem Schleifen-Befehl **Do[ ]** berechnet:

```
Do[i^2, {i, 1, 5}]
```

i^2 für i =1, 2, 3, 4, 5 wird nicht angezeigt, obwohl die obere Zelle an den Kernel geschickt wurde:

Innerhalb von Schleifen werden die berechneten Werte auch ohne ";" nicht angezeigt!

Wenn man Werte innerhalb der Schleife ausdrucken will, benötigt man den **Print-Befehl**:

```
Do[Print[i^2], {i, 1, 5}]
```

```
1
4
9
16
25
```

{i, 1, 5} heisst, dass die Zählvariable i zuerst gleich 1 ist und beim 2. Durchgang durch die Schleife 1 grösser, also gleich 2 ist. Und so weiter bis zum letzten Durchgang durch die Schleife, bei der i gleich 5 ist.
Man muss nicht unbedingt bei i = 1 starten:

```
Do[Print[i^2], {i, 4, 6}]
```

```
16
25
36
```

Wenn man will, dass die Zählvariable 2er-Schritte macht, gibt man das als 4. Element der Liste ein: **{i, 3, 7, 2}**

```
Do[Print[i^2], {i, 3, 7, 2}]
```

```
9
25
49
```

Schleifen durchlaufen kann *Mathematica* ziemlich schnell. Man kann das mit dem Befehl **Timing[ ]** nachprüfen:

```
Timing[Do[i^2, {i, 1, 1 000 000}]]
```

```
{0.613864, Null}
```

Innerhalb von ca. 3.5 Sekunden wurde oben die Schleife 1 Million Mal durchlaufen!
Zum Glück werden die Quadratzahlen ohne den Print-Befehl nicht angezeigt....

Als 2. Beispiel sollen die 5 Zahlen 7, 11, 15, 19, 23 programmiert werden.
Das kann man mit der **Zuordnung (Es ist keine Gleichung!)** `a = a + 4` machen:
Sie bedeutet, dass die **Variable a neu um 4 grösser ist als das alte a**!
Damit kann man nun die Zahlen mit dem Startwert a = 7 programmieren. In jedem Durchgang wird a zuerst gedruckt und dann um 4 erhöht:

```
a = 7;
Do[Print[a]; a = a + 4, {i, 1, 5}]
```

```
7
11
15
19
23
```

Die 2 Befehle `Print[a]; a = a + 4` werden in jedem der 5 Durchgänge der Schleife ausgeführt.

> Man kann also auch **mehrere Befehle** in einem Schleifendurchgang ausführen. Man muss sie einfach durch
> **Strichpunkte** trennen!

Die Variable i wird oben nur dazu benötigt, dass die Schleife 5 Mal durchlaufen wird. Man könnte es also auch so programmieren:

```
a = 7;
Do[Print[a]; a = a + 4, {i, 200, 600, 100}]
```

```
7
11
15
19
23
```

Für spätere schwierigere Beispiele benötigt man die Zuordnung `a = a + 4`. Das obige Beispiel geht natürlich viel
einfacher, wenn man weiss, wie gross die letzte Zahl ist:

```
Do[Print[a], {a, 7, 23, 4}]
```

```
7
11
15
19
23
```

Den Do-Befehl benötigt man, wenn man weiss, wie oft man eine Schleife durchlaufen will. Wenn ich jedoch die obigen Zahlen weiterdrucken will, bis zur letzten kleiner als 50, macht die Sache ein **Schleifen-Befehl mit Abbruch-Bedingung**, der **While-Befehl**, um einiges einfacher:
Er hat die Struktur **While["Bedingung", "Befehle, solange die Bedingung erfüllt ist"]**:

```
a = 7;
While[a < 50, Print[a]; a = a + 4]
```

```
7
11
15
19
23
27
31
35
39
43
47
```

Solange a < 50 ist, wird die Schleife mit den 2 Befehlen `Print[a]; a = a + 4` durchlaufen.
Wenn man es genau nimmt, hat die Variable a am Schluss des letzten Durchgangs den Wert 47+4, aber sie wird nicht mehr ausgedruckt:

```
a
```

```
51
```

Aufgabe 15:
Drucke die ersten 100 Zahlen der Form:
a) $\sqrt{1}$, $\sqrt{2}$, $\sqrt{3}$, $\sqrt{4}$, ...
b) die numerischen Werte der Zahlen von a).
c) 1, 3, 5, 7, 9, ...
d) 24, 29, 34, 39, ...
e) 3, 6, 12, 24, 48, ...
f) 27, 9, 3, 1, $\frac{1}{3}$, ...

Aufgabe 16:
Drucke die Zahlen solange sie kleiner als 1000 sind:
a) 2, 4, 6, 8, ...
b) 10, 13, 16, 19, ...
c) 3, 6, 12, 24, 48, ...

Aufgabe 17:
Überlege dir im Kopf, was folgende Schleifen / Algorithmen machen:
(Was wird genau ausgedruckt?)

a)
```
Do[Print[2 i + 3], {i, 1, 10}]
```

b)
```
Do[b = i^2; Print[b], {i, 1, 10}]
```

c)
```
Do[3 i^2 // Print, {i, 2, 10, 4}]
```

d)
```
Do[Print[5 i], {i, 50, 10, -5}]
```

e)
```
a = 100;
  Do[Print[a // N]; a = a / 2, {i, 1, 6}]
```

f)
```
a = 30;
 While[a > 0, Print[a]; a = a - 4]
```

Aufgabe 18:
Drucke die ersten 100 Zahlen der Form 1, 1, 2, 3, 5, 8, 13, 21, ...

Interessant und später (bei uns jetzt schon) auch Stoff im Mathematik-Unterricht sind Zahlen, die sich immer mehr einer bestimmten Zahl, dem sogenannten **Grenzwert der Zahlenfolge**, nähern. Als Beispiel die Zahlen, die mit dem Start a =3 und der Zuordnung $a = \frac{1}{a+4}$ entstehen :
"Die nächste Zahl ist der Kehrwert der um 4 grösseren Zahl."
Die ersten 4 Zahlen sehen wie folgt aus:

```
a = 3.0;
Do[Print[a]; a = 1 / (a + 4), {i, 1, 4}]
```

```
3.
```

```
0.142857
```

```
0.241379
```

```
0.235772
```

Wenn es um den gesuchten Grenzwert geht, interessieren aber nicht die ersten, sondern zum Beispiel die 100. Zahl:

```
a = 3;
Do[a = 1 / (a + 4), {i, 1, 99}]
N[a]
N[a, 50]
```

```
0.236068
```

```
0.23606797749978969640917366873127623544061835961153
```

Die 101. Zahl unterscheidet sich schon nicht mehr von der 100. Zahl bei 50 Stellen Genauigkeit:

```
b = 3;
Do[b = 1 / (b + 4), {i, 1, 100}]
N[b]
N[b, 50]
```

```
0.236068
```

```
0.23606797749978969640917366873127623544061835961153
```

Der Unterschied zwischen der 100. und 101. Zahl ist schon fast Null:

```
b - a
N[b - a]
```

```
20 /
 152 614 750 049 804 174 727 886 047 490 349 151 578 154 867 855 388 774 436 529 837 711 746 ×
  421 573 431 147 866 885 441 638 497 455 484 977 842 756 945 375 714 498 003
```

$$1.31049 \times 10^{-124}$$

Auch die 1000. Zahl "bewegt sich nicht mehr weg" von der Zahl 0.2360679... :

```
a = 3;
Do[a = 1 / (a + 4), {i, 1, 999}]
N[a]
N[a, 50]
```

```
0.236068
```

```
0.23606797749978969640917366873127623544061835961153
```

Die Zahlen nähern sich also immer mehr dem Grenzwert s:

s = `0.2360679774997896964091736687312762354406183596115 3.....`

Weil bei der Zuordnung `a = 1 / (a + 4)` die nächste Zahl immer etwas anders als die vorhergehende ist, muss s hier eine unendliche lange Dezimalzahl sein. Diese Änderung wird einfach immer mehr zu fast nichts, also immer mehr zu 0.

Wenn man den Grenzwert s algebraisch bestimmen will, benutzt man, dass für ihn bei der Zuordnung `a = 1 / (a + 4)` eben nichts mehr ändert : Die nächste Zahl s muss gleich der vorhergehenden s sein!

Also löst man folgende Gleichung:

```
Solve[s == 1 / (s + 4), s]
```

$\{\{s \to -2 - \sqrt{5}\}, \{s \to -2 + \sqrt{5}\}\}$

Der Grenzwert s der Zahlen ist die 2. Lösung der obigen Gleichung:

```
N[-2 + √5, 50]
```

`0.23606797749978969640917366873127623544061835961153`

s ist wegen der Wurzel irrational.

Wenn man nun wissen will, welche der Zahlen zum ersten Mal weniger als ein Milliardstel vom Grenzwert s entfernt ist, benutzt man die While-Schleife:

```
a = 3;
While[Abs[-2 + √5 - a] ≥ 1 / 10^9, a = 1 / (a + 4)]
Abs[-2 + √5 - a]
Abs[-2 + √5 - a] // N
```

$\frac{375\,125}{167\,761} - \sqrt{5}$

1.58903×10^{-10}

Der Unterschied ist wie gewünscht kleiner als 10^{-9}, nur weiss man nicht, die wievielte Zahl es war. Also baut man noch eine Zählvariable z ein, die bei jedem Durchgang durch die Schleife um 1 grösser wird:

```
a = 3;
z = 1;
While[Abs[-2 + √5 - a] ≥ 1 / 10^9, a = 1 / (a + 4); z = z + 1]
Abs[-2 + √5 - a] // N
z
```

1.58903×10^{-10}

9

Bereits die 9. Zahl ist weniger als ein Milliardstel vom Grenzwert entfernt.

Aufgabe 19:
Beantworte bei den untenstehenden Zahlen folgende Fragen:

1) Bestimme die 50. und 51. Zahl. Wie gross ist der Unterschied zwischen ihnen?
2) Bestimme den Grenzwert der Zahlen algebraisch.
3) Die wievielte der Zahlen ist zum 1. Mal weniger als ein Billionstel vom Grenzwert entfernt?
4) Gibts einen andern Grenzwert, wenn man mit einer andern Zahl startet?

a) Starte die Zahlen mit a = 4 und die Zuordnung ist $a = \frac{1}{a+4}$.
b) Starte die Zahlen mit a = 5 und die Zuordnung ist $a = \frac{2}{a+1}$.
c) Starte die Zahlen mit a = $\frac{1}{2}$ und die Zuordnung ist $a = \frac{10-a}{10}$.

Natürlich kann man auch Funktionen, die Schleifenbefehlen enthalten, definieren. Als Beispiel eine Funktion "folge" mit der Zuordnung $a = \frac{1}{a+4}$ und dem Startwert a = 3, bei der man "die wievielte der Zahlen man berechnet haben will" als Argument eingeben kann:

```
folge[n_] := Do[a = 1 / (a + 4), {i, 1, n - 1}]
```

Die 100. Zahl bekommt man also mit:

```
a = 3
folge[100]
N[a, 50]
```

```
3
```

```
0.23606797749978969640917366873127623544061835961153
```

Wenn man auch noch den Startwert a als Argument auf natürliche Art in die Funktion einbauen will, klappt es nicht.
Versuche es! Schön wäre auch, dass die berechnete Zahl a direkt als Funktionswert rauskommt.
Das alles wird strukturiert und einfach mit dem Befehl **Module[ ]** gelöst.
Zuerst ein paar Bemerkungen zu diesem Befehl:
Man braucht **Module[ ]**vor allem, wenn man eine **Funktion definieren will, die mehrere Befehle und Variablen enthält.** Diese Variablen haben dann nur im Funktionskörper eine Belegung und ausserhalb nicht. Ein Beispiel dazu:

```
z = 5;
Module[{z}, z = 10]
z
```

```
10
```

```
5
```

Oben wurde der Variable z eigentlich der Wert 10 zugeordnet, und trotzdem hat z nachher immer noch den Wert 5!? Das ist so, weil die Zuordnung im Module-Befehl geschah und darum ausserhalb davon keine Bedeutung hat:
In der **geschweiften Klammer** im Module-Befehl werden die **Hilfsvariablen** definiert. Sie haben nur innerhalb des Module-Befehls eine Belegung. Solche Variablen nennt man **lokal**.
Im obigen Beispiel haben wir also eigentlich zwei Variablen:
Eine lokale Variable z = 10 und eine **globale** Variable z = 5. Weil lokale Variablen nur im Module-

Befehl eine Belegung haben und ausserhalb nicht, wird auch nach dem Module-Befehl immer noch 5 und nicht 10 für z gemeldet!
Jetzt verstehst du vielleicht, wieso unten die Variable "hase" nach dem Module-Befehl immer noch keine Belegung hat:

```
hase
Module[{hase}, hase = 10; hase]
hase
```

```
hase

10

hase
```

Das zweite Argument des Module-Befehls wird immer ausgegeben: Oben meldet also Module die Ausgabe
"hase = 10; hase". Weil der Strichpunkt ";" die Ausgabe von "hase=10" unterdrückt, wird nur einmal "10" ausgegeben.

Mit dem Module-Befehl kann man nun **mehrere Befehle schön strukturiert** in eine Funktion packen. Jetzt unsere obige Funktion "folge" mit 2 Argumenten für "Startwert" und "Wie viele Zahlen":

```
folge[start_, n_] := Module[{a}, a = start;
   Do[a = 1 / (a + 4), {i, 1, n - 1}];
  N[a, 50]]
```

Der Funktionskörper der Funktion "folge" besteht also aus 3 Befehlen, die durch Strichpunkte getrennt werden
müssen:
Eine Zuordnung , die Do-Schleife und die Ausgabe, bei der Print[ ] nicht nötig ist.

Unser Beispiel von oben als **ein** Funktionsaufruf:

```
folge[3, 100]
```

```
0.23606797749978969640917366873127623544061835961153
```

Noch ein zweites Beispiel:

```
folge[3 / 2, 30]
```

```
0.23606797749978969640917366873127623501425976501621
```

Wer etwas kritisch ist und sich selber was überlegt, fragt sich sicher, wieso eigentlich die Hilfsvariable a nötig ist. Kommt man nicht auch ohne aus? Kann man nicht direkt mit der globalen Variable a in die Do-Schleife gehen?

```
folge2[a_, n_] := Module[{}, Do[a = 1 / (a + 4), {i, 1, n - 1}]; N[a, 50]]
```

```
folge2[3, 100]
```

```
3.00000000000000000000000000000000000000000000000000
```

Mathematica reklamiert und meldet den Startwert. Man darf also nicht! Man darf mit dem globalen Argument a im Module-Befehl keine Zuweisungen machen. So ist man gezwungen strukturiert zu arbeiten und lokal in Funktionsaufrufen die globalen Variablen, Argumente nicht zu verändern.
Man benötigt in diesem Beispiel also die Hilfsvariable.

Als Bemerkung sei hier noch erwähnt, dass man auch keine oder mehrere Hilfsvariablen definieren kann und man die Zuweisungen zu den Hilfsvariablen bereits in der geschweiften Klammer machen darf. Als Beispiel die Funktion folge[ ] etwas anders mit 2 Hilfsvariablen programmiert:

```
folge3[start_, n_] :=
 Module[{a = start, ende = n - 1}, Do[a = 1 / (a + 4), {i, 1, ende}];
  N[a, 50]]

folge3[3, 100]
```

```
0.23606797749978969640917366873127623544061835961153
```

Weil der obige Abschnitt etwas schwierig war und nur gemacht wurde, damit du später bei eigenen schwierigen
Programmen vielleicht deinen Fehler findest, jetzt ein Beispiel, wie man Module normalerweise anwendet:

Ich möchte eine Funktion (einen Befehl) definieren, die bei einer Strasse mit dem Neigungswinkel α gegenüber der
Horizontalen die Steigung in Prozenten berechnet:
Wenn man strukturiert vorgeht, besteht die Lösung eigentlich aus 2 Schritten:
1) Man berechnet mit Hilfe des Tangens das Verhältnis Gegenkathete zu Ankathete.
2) Weil die Steigungsprozente als Länge der vertikalen Strecke mit horizontaler Strecke 100 m vorkommen,
muss man das in 1) berechnete Verhältnis noch mit 100 multiplizieren. (Wieso?)

Diese 2 obigen Schritte kann ich nun strukturiert mit Hilfe des Module-Befehls in eine Funktion "packen":
1) Zuerst wird die Hilfsvariable h berechnet: h = tan(α).
2) Die Steigung ist dann die auf 6 wesentliche Ziffern gerundete Zahl 100 · h.

Also sieht meine Funktion "steigung" strukturiert mit dem Module-Befehl so aus:

```
steigung[alpha_] := Module[{h = Tan[alpha]}, N[100 * h]]
steigung[10 °]
steigung[45 °]
```

```
17.6327
```

```
100.
```

Das obige strukturierte Vorgehen ist hilfreich, weil man später im Notebook vielleicht nochmals den Variablennamen "h" benutzt und dann froh ist, dass "h" nur im Module-Befehl eine Belegung mit einem konkreten Wert hatte! Dem globalen "h" ist also auch jetzt noch immer keinen Wert zugeordnet:

```
h
```

```
h
```

Wenn ich jetzt eine Gleichung nach h auflösen will, klappt alles:

```
Solve[q h + s == h, h]
```

$$\left\{\left\{h \to -\frac{s}{-1+q}\right\}\right\}$$

Wenn h ohne Module-Befehl zum Beispiel die Belegung h = arc tan(45°) hat und ich viel später in diesem Notebook die obige Gleichung lösen wollte, ginge es schief:

```
h = ArcTan[45 °];
Solve[q h + s == h, h]
```

```
Solve::ivar : ArcTan[45 °] is not a valid variable. >>
Solve[s + q ArcTan[45 °] == ArcTan[45 °], ArcTan[45 °]]
```

Aber natürlich kann man die obige Funktion "steigung" ohne Module und Hilfsvariablen programmieren:

```
steig[α_] := N[100 * Tan[α]]
steig[10 °]
steig[45 °]
17.632698070846498`
100.`
```

Aufgabe 20:
Der Flächeninhalt A eines allgemeinen Dreiecks mit Seiten *a*, *b*, *c* lässt sich mit der *Heronschen Flächenformel* berechnen:

$A = \sqrt{s(s-a)(s-b)(s-c)}$ wobei $s = \frac{a+b+c}{2}$.

Programmiere diese Formel mit Hilfe des Module-Befehls, und teste sie mit einem (mehreren) Dreiecken, deren
Flächeninhalt du per Hand berechnen kannst.

Aufgabe 21:
Programmiere die folgenden Funktionen strukturiert mit Hilfe des Module-Befehls, und teste sie mit ein paar Argumenten:

a) **`quadrate[n_]`** Druckt die Quadratzahlen bis zur n-ten aus!

b) **`halbieren[start_, n_]`** Halbiert die Zahl start n Mal!

c) **`millionaer[kapital_, zins_]`** meldet wie viele Jahre es geht, bis man Millionär ist bei einem Guthaben von kapital Franken und zins % jährlichen Zins.

Aufgabe 22:
Gegeben sind die Zahlen mit dem Start a = 10 und der Zuordnung $a = \frac{1}{2}\left(a + \frac{c}{a}\right)$ mit dem Parameter c > 0.

a) Programmiere die Funktion **`folge[c_]`**, die die 20. Zahl der gegebenen Zahlen auf 30 Stellen gerundet berechnet, wenn man den Parameter c als Argument eingibt.
Teste dann deine Funktion mit den Argumenten c = 2, 3, 4, 5. Was vermutest du für die Grenzwerte der Zahlen abhängig von c?

b) Beweise deine in a) gefundene Vermutung noch algebraisch.

Aufgabe 23: (freiwillige Aufgabe!)
Schreibe ein Programm, das drei Zahlen *a*, *b* und *c* als Argument nimmt und mit Hilfe des Satzes von Pythagoras testet, ob diese Seiten eines rechtwinkligen Dreiecks sein können. Solche Zahlen nennt man übrigens **Pythagoräische Tripel** ... . Als Ergebnis sollte dein Programm entweder "rechtwinklig, Hypotenuse ist ..." oder "nicht rechtwinklig" ausgeben.

Tipps: - Benutze wieder das Gleichheitszeichen "==", um auf Gleichheit zu testen.
- Du brauchst mehrere **If[]**-Bedingungen, um alle möglichen Fälle zu testen (*a* ist Hypotenuse, *b* ist Hypotenuse, *c* ist Hypotenuse).

Wie immer am Schluss des Kapitels ein kleiner Vorgeschmack, was *Mathematica* auch kann: Nämlich Musik machen! Wenn du mehr darüber erfahren willst, schaue im Helpbrowser nach, wo die Befehle genauer erklärt werden und es auch eine Einführung und ein Tutorial dazu hat.

Aufgabe 24:
Tippe die Befehle ab, und drücke "Play" in der Outputzelle:

a) `Sound[{SoundNote["C"], SoundNote["G"]}]`

b) `Sound[{"Flute", Table[SoundNote[i], {i, 0, 12}]}]`

c) `Play[Sin[440 × 2 Pi t] // Sign, {t, 0, 1}]`

d) `Play[Sin[2 π 20 000 t], {t, 0, 6}]`

e)
```
Sound[
  {Play[Sin[1000 t (1 + t^2)], {t, 0, .2}], Play[Sin[500 t (1 + t^3)], {t, 0, .5}]}]
```

Wir schliessen den 3.*Teil der Einführung in Mathematica* mit dem Kapitel

Algorithmik - Zwei Anwendungen: die Rekursion & die Iteration

Aufgabe:
Definiere die Begriffe **Algorithmik**, **Rekursion** und **Iteration**.

Vereinbarung: Wir werden nicht weiter die Rekursionen und die Iterationen unterscheiden, sondern sie als gleichwertig behandeln und somit synonym verwenden.

Aufgaben:
Konkrete Anwendungen:

a) Methoden zur Nullstellenbestimmung:
- Bisektionsverfahren
- Sekantenverfahren

b) Wurzelberechnungen nach Heron

c) Primzahlsuche mit den dem Sieb des Eratosthenes

d) Berechne mit Hilfe einer Rekursion den Inhalt und den Umfang einer *Koch'schen Schneeflocke*.
Baue selbständig ein vernünftiges Abbruchkriterium ein.
Versuche die Koch'sche Schneeflocke graphisch darzustellen.

Rechengenauigkeit und der N[ ]-Befehl

Mathematica rechnet exakt, wenn es geht. Zum Beispiel sind Brüche für ihn Brüche und keine gerundeten
Dezimalzahlen:

```
exakt = 1 / 3
```

$\frac{1}{3}$

Also kann er die Variable "exakt" mit jeder beliebigen Genauigkeit gerundet darstellen :

```
N[exakt, 20]
N[exakt, 50]
```

```
0.33333333333333333333
```

```
0.33333333333333333333333333333333333333333333333333
```

> Der Befehl `N[z, k]` ist ein **Formatierungsbefehl, der versucht die Zahl z auf k wesentliche Ziffern gerundet darzustellen.** Er rundet die Zahlen im Kernel eigentlich nicht, sondern stellt die Zahl gerundet dar.
> **Zum Runden gibts darum den Befehl Round[ ]!**

Etwas schwieriger zu beurteilen ist, wie exakt Mathematica bei Dezimalzahlen rechnet!
Ein erstes Beispiel:

```
2.123456789
```

```
2.12346
```

Die obige Dezimalzahl wird in der Outputzelle bereits auf 6 wesentliche Ziffern gerundet **dargestellt**!
Auch mit dem Befehl N[ ] kann man die Darstellung der Dezimalzahl nicht mehr verändern:

```
N[2.123456789, 8]
N[2.123456789, 2]
```

```
2.12346
```

```
2.12346
```

Die auf **6 wesentliche Stellen gerundete Darstellung** ist bereits durch das Setzen des Dezimalpunkts festgelegt.
Natürlich ist die obige Zahl intern immer noch exakt gespeichert. Man kann das mit dem Befehl **Precision[ ]**
nachprüfen, der angibt, mit wie vielen Ziffern der Wert intern gespeichert ist:

```
Precision[2.123456789]
```

```
MachinePrecision
```

```
$MachinePrecision
```

```
15.9546
```

Die Maschinengenauigkeit ist ungefähr auf 16 Ziffern eingestellt. Der obige ungenaue Wert entsteht, weil *Mathematica* intern nicht im Dezimalsystem, sondern im Binärsystem mit den Ziffern 0 und 1 mit

53 Ziffern rechnet:
(Die untere Zeile kannst du noch nicht verstehen, weil du den Zehnerlogarithmus noch nicht kennst!)

```
Log[10, 2^53] // N
```

```
15.9546
```

N[z] ist also nur ein Darstellungsbefehl, der die Zahl z in der Output-Zelle gerundet darstellt und hat darum seine "Macken".
Wenn man nicht nur die Darstellung gerundet haben will, benutzt man besser den Round-Befehl, der wirklich die Zahl rundet:

```
zahl1 = 6.4567
Round[zahl1]
zahl2 = 0.4567
Round[zahl2]
```

```
6.4567

6

0.4567

0
```

Ohne 2. Argument rundet Round[ ] auf ganze Zahlen. Mit einem 2. Argument kann man angeben, auf wieviele Stellen nach dem Komma man die Zahl gerundet haben will:

```
zahl1 = 6.4567
Round[zahl1, 1/100]
Round[zahl1, 0.01]
```

```
6.4567

323
---
50

6.46
```

Oben wurde in beiden Fällen auf Hunderstel gerundet. Im ersten Fall stellt *Mathematica* die Zahl sogar als gerundeten Bruch dar. Wer die negativen Exponenten verstanden hat, könnte das Obige auch so programmieren:

```
Round[zahl1, 10^(-2)]
Round[zahl1, 10^(-2.0)]
```

```
323
---
50

6.46
```

Zurück zur Rechengenauigkeit und dem N[ ]-Befehl:
Wenn man Dezimalzahlen mit mehr als 16 Stellen benutzt, erhöht *Mathematica* automatisch die Genauigkeit von 16 Stellen, damit man immer noch exakt damit rechnen kann:

```
2.123456789123456789
Precision[2.123456789123456789]
```

```
2.123456789123456789
```

```
18.327
```

Wie man oben sieht, wird die Dezimalzahl sogar überraschenderweise exakt in der Outputzelle dargestellt.
Sobald man die Maschinengenauigkeit von 16 Ziffern erhöht, wird auch die auf 6 Stellen gerundete Darstellung in der Outputzelle aufgehoben.
So kann man nun auf solche Zahlen den N[ ]-Befehl anwenden:

```
N[2.123456789123456789, 10]
N[2.123456789123456789, 3]
```

```
2.123456789
```

```
2.12
```

Was passiert, wenn man die Dezimalzahl 1.5 zur obigen Zahl addiert?

```
2.123456789123456789 + 1.5
```

```
3.62346
```

```
Precision[2.123456789123456789 + 1.5]
```

```
MachinePrecision
```

Weil die Zahl 1.5 intern wegen dem Dezimalpunkt mit 16 Stellen gespeichert ist, kann die Rechnung höchstens mit 16 Stellen Genauigkeit ein Resultat liefern. Also gibt *Mathematica* das Resultat nur noch mit Maschinengenauigkeit von 16 Ziffern an, um "auf der sicheren Seite" zu sein!
Natürlich kann man die obige Rechnung exakt durchführen. Dazu später mehr.

Wie genau werden unendliche Dezimalzahlen dargestellt? Zum Beispiel der anfängliche Bruch $\frac{1}{3}$ = `0.33333....` , wenn man ihn mit einem Dezimalpunkt schreibt?

```
numerisch1 = 1.0 / 3
Precision[numerisch1]
```

```
0.333333
```

```
MachinePrecision
```

Wegen dem Dezimalpunkt wird 1.0 und damit auch die restliche Berechnung (geteilt durch 3) mit 16 Ziffern Genauigkeit durchgeführt.
Wie oben wird also der Dezimalbruch auf 6 wesentliche Ziffern gerundet dargestellt und mit 16 Ziffern intern gespeichert. `1.0 / 3` entspricht also nicht mehr dem Bruch $\frac{1}{3}$, sondern nur noch der endlichen Dezimalzahl 0.33....333.
Wegen dem Dezimalpunkt wird die **ganze** untere Berechnung mit derselben Genauigkeit und Rundung wie bei 1.0
durchgeführt. So sieht man leider keinen Unterschied zwischen der Variablen "exakt" und "numerisch1", obwohl es einen gibt:

```
1 / 3 - 1.0 / 3
```

```
0.
```

Wenn man nun genauer, zum Beispiel mit 30 Stellen Genauigkeit arbeiten will, gibt es mehrere Möglichkeiten.
Die erste Möglichkeit benutzt den N[ ]-Befehl zusammen mit dem exakten Bruch:

```
numerisch2 = N[1 / 3, 30]
```

```
0.333333333333333333333333333333
```

Intern ist der Wert der Variablen "numerisch2" nicht mehr auf 16 Stellen genau abgespeichert, sondern auf 30 Stellen genau:

```
Precision[numerisch2]
```

```
30.
```

Nun werden auch weitere Rechnungen mit der Variablen automatisch mit 30 Ziffern ausgeführt:

```
rechnung1 = 2 numerisch2^3
Precision[rechnung1]
```

```
0.074074074074074074074074074074
```

```
29.5229
```

Natürlich muss man aufpassen, dass keine weiteren Dezimalzahlen in der Berechnung vorkommen:

```
rechnung2 = 2.0 numerisch2^3
Precision[rechnung2]
```

```
0.0740741
```

```
MachinePrecision
```

Die 2. Möglichkeit benutzt den **SetPrecision-Befehl**, der den "Accent Graph" als Kurzform hat:

```
numerisch3 = 1.0`30 / 3
```

```
0.333333333333333333333333333333
```

Der Dezimalbruch 1.0 wurde oben auf 30 Stellen genau definiert, also auch die weitere Rechnung, die ihn durch 3 teilt. Also ist 0.3333.... auf 30 Stellen genau gespeichert:

```
Precision[numerisch3]
```

```
30.
```

Auch hier werden nun weitere Rechnungen mit der Variablen automatisch mit 30 Ziffern ausgeführt:

```
rechnung3 = 2 numerisch3^3
Precision[rechnung3]
```

```
0.074074074074074074074074074074
```

```
29.5229
```

Aufgabe 25:
a) Was meldet *Mathematica* bei den Berechnungen? Überlege es dir vorher!

```
1 - 10^-10 ?
1.0 - 10^-10 ?
(1.0 - 10^-10) - (1.0 - 10^-10) ?
```

b) Führe die Rechnung `1.5 + 2.123456789123456789` exakt durch.

Man kann mit bis zu 1'000'000 Stellen rechnen. Man muss sich einfach im Klaren sein, dass die Berechnungen etwas länger dauern:

```
numerisch4 = N[Sqrt[5]];
rechnung4 = numerisch4^100 // Timing
Precision[rechnung4]
```

$\{0.000059, 8.88178 \times 10^{34}\}$

```
MachinePrecision
```

```
numerisch5 = N[Sqrt[5], 1 000 000];
Precision[numerisch5]
rechnung5 = numerisch5^100; // Timing
Precision[rechnung5]
```

$1. \times 10^{6}$

```
{1.76641, Null}
```

```
999 998.
```

Es dauert ca. 30'000 Mal länger, aber immer noch ziemlich schnell, wenn *Mathematica* mit 1 Million anstelle von 16 Ziffern rechnen muss.

Natürlich muss man die Maschinen-Genauigkeit von 16 Stellen selten verändern, sondern man arbeitet mit *Mathematica* meistens exakt und stellt dann die Resultate mit dem N[ ]-Befehl gerundet dar. Man sollte sich einfach bewusst sein, dass man nur mit 16 Stellen Genauigkeit arbeitet, sobald man einen Dezimalpunkt setzt!
Bei Schleifen ist das Setzen des Dezimalpunkt manchmal auch sehr nützlich, weil man damit verhindert, dass
Mathematica algebraisch exakt rechnen muss. Damit können Schleifen viel schneller durchlaufen werden.
Als Beispiel eine Schleife mit Bruch-Operationen, bei der verschachtelte Doppelbrüche entstehen, wenn *Mathematica* exakt rechnen muss :

```
a = 3;
Do[a = a + 1 / a, {i, 1, 22}] // Timing
N[a, 40]
```

```
{47.7494, Null}
```

```
7.342996076009128585091552673043861079282
```

Es braucht einige Zeit und sogar einige Minuten, wenn man die Schleifen-Anzahl von 22 auf zum Beispiel 25 erhöhen würde, weil *Mathematica* mit "Riesen-Doppelbrüchen" rechnen muss. Um einiges schneller wird es, wenn man mit
Dezimalzahlen arbeitet: **3.0** anstelle von **3**!
Zuerst mit 16-stelligen Dezimalzahlen:

```
a = 3.0;
Do[a = a + 1 / a, {i, 1, 22}] // Timing
N[a, 40]
Precision[a]
```

```
{0.000493, Null}
```

```
7.343
```

```
MachinePrecision
```

Mathematica meldet das auf 6 wesentliche Ziffern gerundete **exakte** Resultat 7.34300 ohne die Nullen!
Jetzt mit 100 Stellen Genauigkeit:

```
a = 3.0`100;
Do[a = a + 1 / a, {i, 1, 22}] // Timing
N[a, 40]
Precision[a]
```

```
{0.00081, Null}
```

```
7.342996076009128585091552673043861079282
```

```
100.
```

Das Resultat ist "auf 40 Stellen genau" genau so exakt wie beim algebraischen Berechnen, aber die Schleife wurden ca. 50'000 Mal schneller durchlaufen!

Auf diesen Kopien zum letzten Mal am Schluss des Kapitels ein kleiner Vorgeschmack, was *Mathematica* auch kann:

Aufgabe 26:
Tippe den unteren Befehl ein, und klicke in die Graphik. So solltest du mit Hilfe des Cursors die entstandene Welle im Raum drehen können:

```
Plot3D[Sin[x], {x, 0, 12}, {y, 0, 6}, Boxed → False, Axes → False]
```