

Skript: Wissenschaftliches Rechnen mit Python

Martin Guggisberg

22. Juni 2012

Inhaltsverzeichnis

1	Ausdrücke und mathematische Formeln	5
1.1	Das erste Programm	6
1.1.1	Kommandozeile als Taschenrechner	6
1.1.2	Python Programme schreiben	6
1.1.3	Einsatz von Variablen	8
1.1.4	Wahl der Variablen (Namensbezeichnungen)	9
1.1.5	Kommentare	9
1.1.6	Formatierte Ausgabe von Texten und Zahlen	10
1.2	Celsius - Fahrenheit Umrechnungsprogramm	11
1.2.1	Potenzielle Fehler bei einer Integer-Division	11
1.2.2	Integer-Division vermeiden	12
1.3	Mathematische Funktionen berechnen	13
1.3.1	Quadratwurzel	13
1.3.2	Beispiele mit mathematischen Formeln	14
1.4	Komplexe Zahlen	15
1.4.1	Rechnen mit komplexen Zahlen in Python	16
1.5	Zusammenfassung Kapitel 1	16
1.5.1	Abschliessendes Beispiel: Trajektorie eines Balles	17
1.6	Übungen zu Kapitel 1	18
2	Schleifen und Listen	21
2.1	Schleifen	21
2.1.1	Naive Lösung: Copy Paste	21
2.1.2	While-Schleife	22
2.1.3	Logische Ausdrücke	23
2.1.4	Berechnung von Summen	24
2.2	Listen	25
2.2.1	Funktionalität oder Operationen einer Liste	25
2.3	For-Schleife	28
2.3.1	Ausgabe in einer Tabelle	28
2.3.2	Die range() Funktion	29
2.3.3	For-Schleifen mit Indizes	30
2.4	Verschachtelte Listen	32
2.4.1	Extrahieren von Teillisten (sublist)	33
2.4.2	Traversieren durch verschachtelte Listen	33
2.4.3	Zusammenfassung Kapitel 2	34
2.4.4	Übungen zu Kapitel 2	36

3	Funktionen und Verzweigungen	41
3.1	Funktionen	41
3.1.1	Funktionen mit einer Variablen	41
3.1.2	Lokale und Globale Variablen	42
3.1.3	Mehrere Übergabeparameter	44
3.1.4	Mehrere Rückgabeparameter	45
3.1.5	Vordefinierte Übergabeparameter	47
3.1.6	Übergabeparameter des Typs Funktion	49
3.2	Verzweigung	49
3.2.1	IF-Else Block	50
3.3	Zusammenfassung Kapitel 3	51
3.4	Abschliessendes Beispiel	52
3.5	Übungen zu Kapitel 3	54
4	Rechnen mit Arrays / Darstellung von Kurven (Plots)	59
4.1	Python Arrays - Numpy	59
4.1.1	Koordinaten und Funktionswerte	60
4.2	Kurvendarstellung mit Hilfe der Bibliothek Matplotlib	60
4.3	Animierte Kurve	65
4.4	Funktionen mit höherem Schwierigkeitsgrad	66
4.4.1	Stückweise definierte Funktionen	67
4.4.2	Schnell variierende Funktionen	69
4.5	Zusammenfassung Kapitel 4	70
4.6	Abschliessendes Beispiel	71
4.7	Übungen zu Kapitel 4	73
Anhang		77
A	Installationsanleitung	77
A.1	ENTHOUGHT Python Distribution 7.2	77
A.2	Installation von Python 2.7	80
B	Installation von zusätzlichen Softwarepaketen	80
B.1	Matplotlib	80
B.2	NumPy und SciPy	80
B.3	SciTools	81
B.4	Weitere Informationen	81

Kapitel 1

Ausdrücke und mathematische Formeln

Einleitung

Dieses Skript soll einen mathematischen Einstieg in die Programmierung mit Python ermöglichen. Zahlreiche Beispiele aus dem mathematischen und naturwissenschaftlichen Bereich sollen Nutzen und Einsatz von numerischen Methoden aufzeigen. Als Programmiersprache wurde Python gewählt, weil es eine moderne Programmierung mit einer sehr kompakten und klaren Syntax ermöglicht. Python ist einfach zu lernen und wird an zahlreichen Schulen und Universitäten als Einstiegsprogrammiersprache unterrichtet.

Beim Einstieg in die Programmierung ist es hilfreich, zu lernen, was und wie Programmiererinnen und Programmierer denken. Ähnlich der Bedeutung des räumlichen Vorstellungsvermögens im mathematischen Fachbereich wird in der Informatik eine Vorstellung des zeitlichen Verlaufs eines Programms benötigt. Um ein Programm erstellen zu können, braucht es eine Vorstellung, wie dieses auf der Maschine ausgeführt wird. Anhand von klar formulierten Aufgaben sollen die Leserinnen und Leser schrittweise eingeführt werden. Spezielle Aufmerksamkeit wird auch dem Umgang mit möglichen Fehlern gewidmet.

Um mit diesem Skript zu arbeiten und die Beispiele lösen zu können, wird Python in der Version 2.7 empfohlen. Ab Kapitel 4 werden die Bibliotheken NumPy, Matplotlib und SciTools verwendet. Eine Anleitung zur Installation der nötigen Software wird im Anhang gegeben. Das Skript richtet sich nach dem englischsprachigen Buch *'A Primer on Scientific Programming with Python'* von Hans Petter Langtangen. Die Beispiele zu diesem Buch findet man unter <http://www.simula.no/intro-programming>

In Kapitel 1 werden Variablen eingeführt. Im Weiterem werden mathematische Ausdrücke mit Hilfe von Python berechnet. Es werden verschiedene Arten der Textformatierung behandelt. In Kapitel 2 werden mit Hilfe von Schleifen Berechnungen oder Iterationen automatisiert ausgeführt. Die Mächtigkeit von Python-Listen werden anhand von zahlreichen Beispielen aufgezeigt. In Kapitel 3 werden die fundamentalen Programmkonzepte bezüglich Funktionen und Verzweigungen behandelt. Kapitel 4 widmet sich der Ausgabe von Graphen. In Kapitel 5 werden einfache Monte Carlo Simulationen rund um Zufallszahlen beschrieben. Die letzten Kapitel widmen sich der numerischen Analysis.

Seit mehreren Jahren wird in zahlreichen Bereichen die Programmiersprache Python eingesetzt (zum Beispiel im Bereich der Simulationen, des wissenschaftlichen Rechnens, im Bereich Biocomputing und Datamining, usw.). Es ist mir ein Anliegen, dass Studierende einfach mit dem vorliegenden Skript arbeiten und die vorgestellten Beispiele direkt am eigenen Computer

ausführen können. Ich bin für alle Rückmeldungen und Hinweise auf Fehler dankbar.

Erst beim Durcharbeiten von Aufgaben können Programmierkonzepte erlernt und ausprobiert werden. Programmieren kann nur durch aktives Lösen von Übungen erlernt werden. Weitere Aufgaben und Beispiele sind auch auf der Webseite

<http://www.programmieraufgaben.ch> publiziert.

Basel, Februar 2012, Martin Guggisberg

1.1 Das erste Programm

Als erstes befassen wir uns mit der Berechnung, respektive Auswertung mathematischer Ausdrücke und Formeln. Wir betrachten eine vertikale Bewegung eines Balls (freie Masse) im Schwerfeld der Erde (vertikaler Wurf) ohne Berücksichtigung des Luftwiderstandes. Mit Hilfe des zweiten Gesetzes von Newton lässt sich eine Funktion für die vertikale Position y als Funktion der Zeit t herleiten.

$$y(t) = v_0 t - \frac{1}{2} g t^2 \quad (1.1)$$

In der verwendeten Gleichung ist v_0 die Anfangsgeschwindigkeit des Balls, g die Erdbeschleunigung und t die Zeit. Die Koordinaten sind so gewählt, dass der Ball von $y = 0$ zur Zeit $t = 0$ startet. Um herauszufinden, wie lange es dauert, bis der Ball sich nach oben bewegt hat und wieder zurück zur Position $y = 0$ kann die Gleichung $y(t) = 0$ betrachtet werden.

$$v_0 t - \frac{1}{2} g t^2 = t(v_0 - \frac{1}{2} g t) = 0 \Rightarrow t = 0 \text{ oder } t = 2v_0/g \quad (1.2)$$

Der Ball kehrt nach $2v_0/g$ Sekunden wieder an die Anfangsposition zurück. Damit lässt sich das zu untersuchende Zeitintervall auf $t \in [0, 2v_0/g]$ reduzieren.

1.1.1 Kommandozeile als Taschenrechner

Unser erstes Programm (die erste Programmzeile) soll den Ausdruck 1.1 für speziell vorgegebene Werte von v_0, g und t berechnen. Für $v_0 = 5m/s$ und $g = 9.81m/s^2$ kehrt der Ball nach $t = 2v_0/g \approx 1s$ zurück. Das zu untersuchende zeitliche Intervall ist $[0, 1]$. Mit Hilfe der Gleichung 1.1 kann nun die Höhe des Balls zur Zeit $t = 0.7$ berechnet werden.

$$y = 5 \cdot 0.7 - \frac{1}{2} \cdot 9.81 \cdot 0.7^2 \quad (1.3)$$

Dieser arithmetische Ausdruck kann mit Python wie folgt berechnet werden.

```
print 5*0.7 - 0.5*9.81*0.7**2
```

Die vier mathematischen Operationen werden in Python analog zu vielen anderen Programmiersprachen durch folgende Symbole (Syntax) $+$, $-$, $*$ und $/$ repräsentiert. Als Potenzzeichen wird zweimal ein Stern verwendet, z. B. wird 0.7^2 mit $0.7**2$ berechnet.

1.1.2 Python Programme schreiben

Bevor mit Programmieren begonnen werden kann, sollten Sie einen Order erstellen in welchem Sie ihre Python Programme aufbewahren, z. B. mit dem Namen *PyProgs* auf dem Schreibtisch oder in ihrem Dokumenten Ordner. Um die Programme auszuführen, brauchen Sie ein *Konsolen Fenster* (auch genannt *Terminal* oder *Python Shell*). Unter Windows starten Sie das

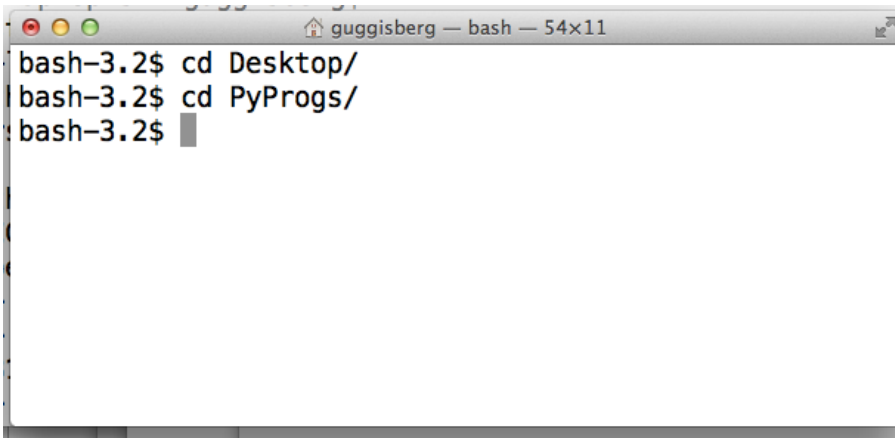
Konsolen Fenster mit dem Befehl *cmd*. Auf einem Mac starten Sie das Programm *Terminal* aus dem Ordner Programme/Dienstprogramme.

Nachdem ein Konsolen Fenster gestartet ist, können Sie mit dem Befehl *cd* (change Directory) zum gewünschten Ordner wechseln.

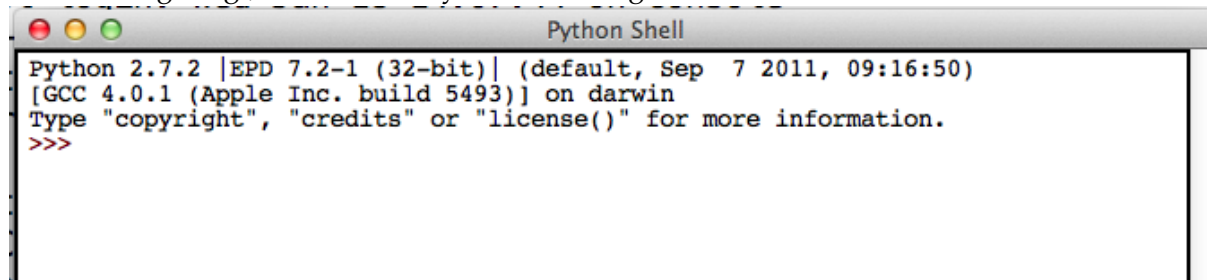
```
> cd Desktop  
> cd PyProgs
```

Damit die Befehle im Konsolen Fenster ausgeführt werden, müssen Sie die RETURN Taste (Zeilenumbruch) betätigen. Als Nächstes können Sie mit folgendem Befehl den Python Editor *Idle* starten.

```
> idle
```

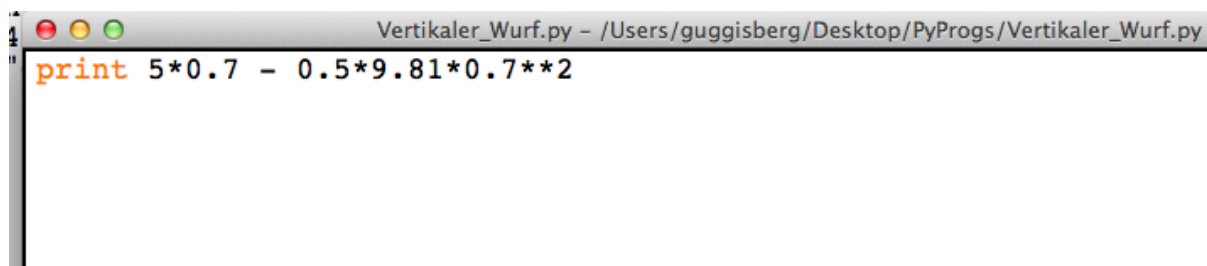


Alternativ kann *Idle* aus dem Startmenü (bei Windows) oder aus Spotlight (bei Mac) gestartet werden. Nach dem Start von *Idle* erscheint ein Fenster mit dem Titel *Python Shell*. In diesem Fenster erscheinen Rückmeldungen, Ausgaben oder Fehlermeldungen von Python. Beim Starten wird angezeigt, mit welcher Python Version gearbeitet wird.

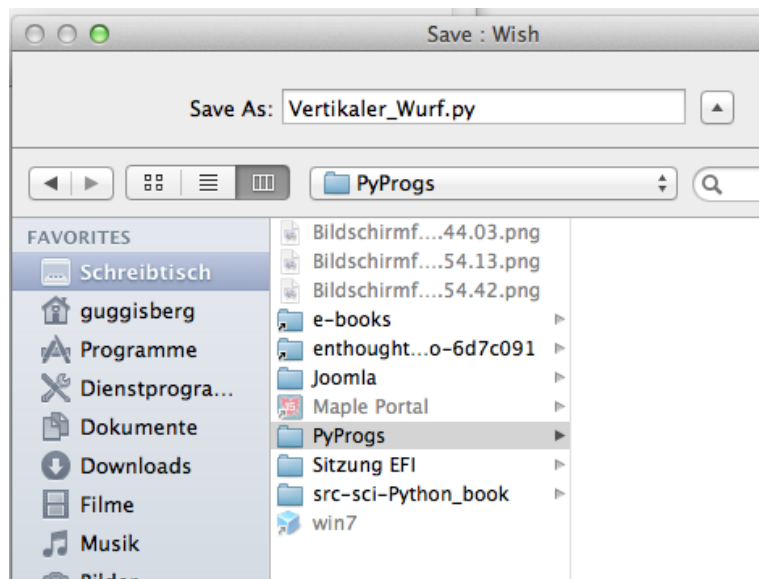


Für die Eingabe des ersten Programms muss ein neues Fenster geöffnet werden. Ein neues Fenster lässt sich aus dem Menü unter **File/New Window** öffnen. Schreiben Sie nun in die erste Zeile des neuen Fensters die folgende Zeile:

```
1 print 5*0.7 - 0.5*9.81*0.7**2
```



Bevor Sie das Programm ausführen, sollten Sie es in ihrem Order (z.B. *PyProgs*) abspeichern. Es ist eine allgemeine Konvention, dass Python Programme die Endung *.py* erhalten.



An dieser Stelle ist es sinnvoll, sich Gedanken zur Kommunikation zwischen Mensch und Maschine zu machen. Ein Programm, das von einer Maschine korrekt ausgeführt werden kann, muss **perfekt** und **präzis** sein. Jeder minimale Fehler beeinflusst den Programmverlauf, häufig führen Fehler dazu, dass ein Programm abbricht oder ein falsches Resultat liefert. Nachdem Sie nochmals überprüft haben, dass Sie die Zeile korrekt abgetippt haben, können Sie nun das Programm aus dem Menü *Run/Run Module* oder mit der Funktionstaste **F5** starten. Das Programm liefert das Resultat (die Ausgabe) im Fenster mit dem Titel *Python Shell*. Sie können ein Programm auch aus dem *Terminal* (oder *Kommando Zeile*) mit folgendem Befehl starten:

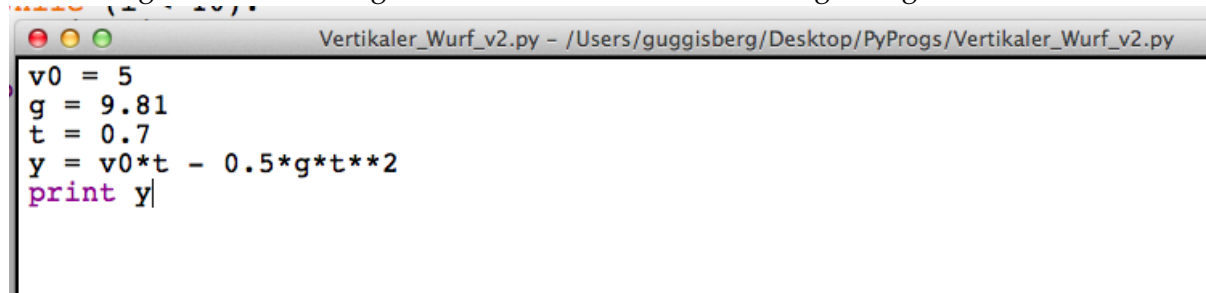
```
> python Vertikaler_Wurf.py
```

1.1.3 Einsatz von Variablen

Zur Berechnung der Höhe des Balls $y(t)$ zu einen neuen Zeitpunkt t und mit einer anderen Anfangsgeschwindigkeiten v_0 können sie die Werte entsprechend ändern. Dazu müssen Sie die entsprechenden Werte in der ersten Zeile anpassen. Bei dieser Methode besteht jedoch die Gefahr, dass eine falsche Zahl verändert wird und dadurch die Rechnung ein falsches Resultat liefert. Um Fehler zu vermeiden, werden in der Programmierung Variablen eingesetzt. Sie sind Platzhalter für die entsprechenden Werte. Variablen in der Informatik sind vergleichbar (aber nicht identisch) mit Variablen in der Mathematik. Ein Programm zur Berechnung der Höhe des Balles mit Hilfe von Variablen könnte etwa wie folgt aussehen:

```
1 v0 = 5
2 g = 9.81
3 t = 0.7
4 y = v0*t - 0.5*g*t**2
5 print y
```


In der folgenden Abbildung sehen Sie das korrekte lauffähige Programm im *Idle* Fenster.



```
Vertikaler_Wurf_v2.py - /Users/guggisberg/Desktop/PyProgs/Vertikaler_Wurf_v2.py
v0 = 5
g = 9.81
t = 0.7
y = v0*t - 0.5*g*t**2
print y
```

Diese Version des Programms (mit Variablen) ist viel besser lesbar, weil sie sich an der mathematischen Schreibweise der Formel orientiert.

1.1.4 Wahl der Variablen (Namensbezeichnungen)

Es erweist sich als nützlich, die Variablen Bezeichnung so nahe wie möglich an mathematischen Bezeichnungen anzulehnen. Damit lassen sich Formeln besser lesen und auch die Korrektheit eines Programms besser überprüfen. Variablennamen können Klein-, Grossbuchstaben und Ziffern beinhalten. Das erste Zeichen eines Variablennamens muss ein Buchstabe sein. Für das behandelte Problem könnten auch die folgenden Namen gewählt werden:

```
1 Anfangs_Geschwindigkeit = 5
2 Erdbeschleunigung = 9.81
3 Zeit = 0.7
4 Vertikale_Position_des_Balls = Anfangs_Geschwindigkeit*Zeit - \
5                               0.5*Erdbeschleunigung*Zeit**2
6 print Vertikale_Position_des_Balls
```

Bei so langen Variablennamen lässt sich die Berechnung nicht mehr in einer Zeile darstellen. Mit einem *Backslash* (\) kann man dem Python-Programm mitteilen, dass die Formel auf zwei Zeilen geschrieben wird. Es ist wichtig, dass nach dem *Backslash* kein weiteres Zeichen folgt, auch kein Leerzeichen. Obwohl die Variablennamen im zweiten Programm präzise definiert sind, ist diese Version viel schwieriger zu lesen als die Version des Programms mit den Namen t , v_0 und g . Als Regel sollte man die Variable möglichst gleich wählen, wie sie in mathematischen Formeln vorkommen.

1.1.5 Kommentare

Zur besseren Lesbarkeit der Programme ist es sinnvoll, lesbare Kommentare in das Programm einzubinden. Ein Kommentar beginnt nach dem Zeichen #. Jeglicher Text nach # wird in dieser Zeile vom Programminterpretierer als Kommentar betrachtet. Unser Programm könnte mit Kommentaren wie folgt aussehen:

```
1 # Programm zur Berechnung der Höhe eines Balls
2 v0 = 5      # Anfangsgeschwindigkeit
3 g = 9.81    # Erdbeschleunigung
4 t=0.7       # Zeit
5 y = v0*t - 0.5*g*t**2    # Vertikale Position
6 print y
```

Gute Kommentare zusammen mit gut gewählten Variablennamen erhöhen die Lesbarkeit von Programmen. Verwenden Sie Kommentare um wichtige Zusatzinformationen, die nicht aus dem Programm ersichtlich sind, anzubringen. Zum Beispiel, zur Bedeutung gewählter Variablennamen oder zur Beschreibung der allgemeine Idee einer Sequenz von Programmzeilen.

1.1.6 Formatierte Ausgabe von Texten und Zahlen

In einem weiteren Schritt soll das Programm zur Berechnung der Höhe anstelle einer Zahl einen Text ausgeben. Der Text soll wie folgt aussehen:

Nach $t=0.7$ s ist der Ball in einer Höhe von 1.10 m

Dabei sollen die Sekunden mit einer Stelle und die Höhe mit zwei Stellen nach dem Komma angegeben werden. Diese spezialisierte Ausgabe kann mit Hilfe des `print` Befehls und entsprechender Steuerzeichen realisiert werden. Auf den ersten Blick sieht die Syntax kompliziert aus, sie erweist sich jedoch als äusserst flexibel und kann für jegliche weitere Art der Formatierung verwendet werden. Die gewünschte Ausgabe kann durch folgende Zeile realisiert werden:

```
print 'Nach t=%g s ist der Ball in einer Höhe von %.2f m' % (t,y)
```

Tabelle 1.1: Wichtige Steuerzeichen der Formatierung

Steuerzeichen	Bedeutung
%s	String
%d	Ganze Zahl
%0xd	Ganze Zahl mit x voranstehenden 0
%f	Dezimalschreibweise mit 6 Stellen
%e	Kompakte wissenschaftliche Notation
%g	Kompakte dezimale oder wissenschaftliche Notation
%xz	Format z rechtsbündig in einem Feld der Grösse x
%-xz	Format z linksbündig in einem Feld der Grösse x
%.yz	Format z mit y Dezimalstellen
%x.yz	Format z mit y Dezimalstellen in einem Feld der Grösse x
%%	Prozentzeichen (%)

Der `print` Befehl gibt den gesamten Ausdruck innerhalb der Anführungszeichen (einzelne ' oder doppelte " Anführungszeichen) aus. Das Prozentzeichen % wird als Steuerzeichen benutzt. Die zwei Ausdrücke %g und %.2f sind Platzhalter für Variablen mit speziellen Formatierungsanweisungen. Für beide Platzhalter müssen am Ende des Ausdrucks zwei Variablen übergeben werden. Dies geschieht mit % (t,y) . Anstelle des ersten Platzhalters %g wird eine rationale Zahl in kompakter Schreibweise mit dem Wert der Variablen t eingefügt. Anstelle des zweiten Platzhalters %.2f wird eine rationale Zahl mit genau zwei Stellen nach dem Komma eingefügt. Zur Vollständigkeit folgt das gesamte Programm:

```
1 v0 = 5
2 g = 9.81
3 t = 0.7
4 y = v0*t - 0.5*g*t**2
5 print 'Nach t=%g s ist der Ball in einer Höhe von %.2f m' % (t,y)
```

Es gibt zahlreiche weitere Ausgabeformate. Zum Beispiel kann eine Zahl in der wissenschaftlichen Notation dargestellt werden, wie z. B. $1.2435e^{-03}$. Der Platzhalter `10.4f` definiert ein Ausgabeformat einer rationalen Zahl bestehend aus 4 Ziffern nach dem Komma und einer maximalen Grösse von zehn Zeichen. In der Tabelle 1.1 finden Sie eine Liste der wichtigen Steuerzeichen (*printf Formatierung*).

In einer Erweiterung unseres Programms werden wir nun verschiedene Steuerzeichen nutzen:

```

1 v0 = 5
2 g = 9.81
3 t = 0.7
4 y = v0*t - 0.5*g*t**2
5 print """
6 Nach t=%f s befindet sich der Ball
7 mit einer Anfangsgeschwindigkeit v0=%.3E m/s
8 in einer Höhe von %.2f m.
9 """ % (t, v0, y)

```

Ein Text innerhalb von drei Anführungszeichen (einzelne `'''` oder doppelte `"""`) kann über mehrere Zeilen gehen. In der `print` Anweisung wird die Zeit `t` als sechsstellige Dezimalzahl ausgegeben, die Geschwindigkeit wird in der zweiten Zeile in der wissenschaftlichen Notation mit 3 Stellen nach dem Komma ausgegeben. Die Höhe wird in der letzten Zeile auf zwei Stellen nach dem Komma gerundet und ausgegeben. Die Ausgabe des obigen Programms sieht wie folgt aus:

```

>>>
Nach t=0.700000 s befindet sich der Ball
mit einer Anfangsgeschwindigkeit v0=5.000E+00 m/s
in einer Höhe von 1.10 m.

```

1.2 Celsius - Fahrenheit Umrechnungsprogramm

Im nächsten Beispiel betrachten wir die Formel zur Umrechnung der Temperatur von Celsius nach Fahrenheit.

$$F = \frac{9}{5}C + 32 \quad (1.4)$$

In dieser Formel ist `C` die Variable der Temperatur in Grad Celsius und `F` die entsprechende Temperatur in Fahrenheit. Unser Ziel besteht nun darin, ein Computerprogramm zu entwickeln, das `F` anhand der Formel berechnen kann, falls die Temperatur `C` in Grad Celsius bekannt ist.

1.2.1 Potenzielle Fehler bei einer Integer-Division

Schreibt man ein Programm aus den Erkenntnissen des letzten Abschnitts, könnte dieses wie folgt aussehen:

```

1 C = 21
2 F = (9/5)*C + 32

```

```
3 print F
```

Die Ausführung dieses Programm liefert den Wert 53. Vergleicht man dieses Resultat mit dem Taschenrechner erhält man $\frac{9}{5}21 + 32 = 69.8$ und nicht den vom Programm ausgegebenen Wert 53. Was ist falsch? Die Formel im Programm scheint korrekt zu sein.

Gleitkomma und Integer Division. Der Fehler in obigem Programm ist einer der häufigsten Fehler bei Programmen mit numerischen Berechnungen. Für einen Programmierneuling ist es nicht klar worum es hier geht. Viele Programmiersprachen kennen zwei Arten der Division: eine Float-Division (Division mit Dezimalzahlen) und eine Integer-Division (Division mit ganzen Zahlen). Die Float-Division ist analog zur Division, die Sie von der Mathematik her kennen. Z. B. $9.0/5.0$ ergibt 1.8. Die Integer-Division rechnet mit ganzen Zahlen, deshalb liefert $9/5$ den Wert 1. Falls eine Integer-Division nicht aufgeht, wird der Rest einfach weggelassen. Der Bruch $\frac{a}{b}$ ergibt bei einer Integer-Division eine ganze Zahl (die Stellen nach dem Komma werden einfach abgeschnitten).

Mathematischer ausgedrückt ist das Ergebnis von $\frac{a}{b}$ die grösste ganze Zahl c , so dass $bc \leq a$. Dies bedeutet, dass eine Integer-Division von $\frac{9}{5}$ das Ergebnis 1 zurück gibt, weil $1 \cdot 5 = 5 \leq 9$, während $2 \cdot 5 = 10 > 9$. Ein weiteres Beispiel ist die Integer-Division $1/5$, sie ergibt 0 da $0 \cdot 5 \leq 1$ (und $1 \cdot 5 > 1$). Viele Programmiersprachen, einschliesslich Fortran, C, C++, Java und Python interpretieren eine Division a/b als Integer-Division, wenn beide Operanden a und b ganze Zahlen sind. Falls entweder a oder b eine Dezimalzahl (Gleitkommazahl) ist, impliziert a/b eine übliche mathematischen Float-Division.

Die Berechnung in obigem Programm soll nun schrittweise nachvollzogen werden. In Zeile 2 werden als erstes die Zahl 9 durch 5 dividiert. Weil beide Zahlen vom Typ Integer sind, verwendet Python standardmässig die Integer-Division und erhält als Zwischenresultat den Wert 1. Dieser Wert wird mit 21 multipliziert und dazu werden 32 addiert. Dies liefert den Wert 53, welcher vom Programm ausgegeben wird.

Im folgenden Abschnitt suchen wir nach einer korrekten Lösung für unsere Temperaturumrechnung.

1.2.2 Integer-Division vermeiden

Als generelle Regel gilt, dass beim wissenschaftlichen Rechnen die Integer-Division möglichst vermieden werden soll. Der einfachste Weg dazu ist, die Variablen und Terme in einem Ausdruck möglichst als Dezimalzahl (Float) zu definieren. In unserem Beispiel gibt es verschiedene Möglichkeiten dies zu erreichen.

```
F = (9.0/5)*C + 32
F = (9/5.0)*C + 32
F = (9.0/5.0)*C + 32
F = float(C)*9/5 + 32
```

Bei den ersten 3 Beispielen ist mindestens ein Operand der Division als Dezimalzahl definiert. Die letzte Variante definiert die Variable (C) am Anfang als Dezimalzahl (Float), dadurch wird die Division als Float-Division ausgeführt. Eine korrekte Lösung unseres Programms zur Umrechnung von Temperatur könnte wie folgt aussehen:

```
C = 21
F = (9.0/5)*C + 32
print F
```

Das Programm liefert nun die korrekte Ausgabe 69.8. Als Anmerkung sei hier erwähnt, dass wir auch beim ersten Beispiel Probleme mit einer Integer Division hätten bekommen können. Falls anstelle der Formel $0.5 * g * t ** 2$ die folgende Formel $(1/2) * g * t ** 2$ verwendet worden wäre.

1.3 Mathematische Funktionen berechnen

Mathematische Formeln beinhalten häufig Funktionen wie \sin , \cos , \tan , \sinh , \cosh , \exp , \log , etc.. Wie ein wissenschaftlicher Taschenrechner kann Python alle mathematischen Funktionen berechnen. In diesem Abschnitt erfahren Sie, wie \sin , \cos , und ähnliche Funktionen in einem Python-Kontext berechnet werden können.

1.3.1 Quadratwurzel

Erinnern Sie sich an die erste Aufgabe zur vertikalen Bewegung eines Balls in 1.1. Nun können wir uns die Frage stellen, wie lange es braucht, bis der Ball eine gewisse Höhe y_b erreicht? Um diese Frage zu beantworten, müssen wir die Gleichung 1.1 nach t auflösen.

$$y_b = v_0 t - \frac{1}{2} g t^2 \quad (1.5)$$

umformen:

$$\frac{1}{2} g t^2 - v_0 t + y_b = 0 \quad (1.6)$$

Diese Quadratische Gleichung hat die beiden bekannten Lösungen:

$$t_1 = (v_0 - \sqrt{v_0^2 - 2gy_b})/g, t_2 = (v_0 + \sqrt{v_0^2 - 2gy_b})/g \quad (1.7)$$

Es gibt meistens zwei Lösungen, weil der Ball die Höhe y_b auf dem Weg nach oben mit ($t = t_1$) und ein zweites Mal auf dem Weg zurück mit ($t = t_2 > t_1$) erreicht. Um die beiden Zeiten t_1 und t_2 zu eruieren, muss das Programm die Quadratwurzeln berechnen können. Die häufig benutzten mathematischen Funktionen befinden sich im Modul *math*. Um diese zu benutzen, muss das Modul *math* importiert werden. Es braucht dazu den Befehl `import math`. Danach kann für eine Variable *a* die Quadratwurzel mit dem Befehl `math.sqrt(a)` berechnet werden. Das folgende Programm berechnet beide Zeiten und gibt diese aus.

```

1 v0 = 5
2 g = 9.81
3 yb = 0.2
4 import math
5 t1 = (v0 - math.sqrt(v0**2 - 2*g*yb))/g
6 t2 = (v0 + math.sqrt(v0**2 - 2*g*yb))/g
7 print 'Nach t=%g s und %g s ist die Hoehe %g m.' % (t1, t2, yb)
```

Das Programm liefert die folgende Ausgabe:

```
Nach t=0.0417064 s und 0.977662 s ist die Hoehe 0.2 m.
```

Es gibt zwei Möglichkeiten Module zu importieren. Die erste Möglichkeit nutzt `import` und den Namen des Moduls.

```
import math
```

Um eine Funktion aus dem Modul `math` aufzurufen, verwendet man den Modulnamen, gefolgt von einem Punkt und dem Namen der Funktion. Im Fall einer Quadratwurzel ist das: `math.sqrt(x)`.

```
y = math.sqrt(x)
```

Manchen Personen ist das Schreiben von `math.sqrt(x)` zu aufwändig. Sie ziehen die Kurzschreibweise `sqrt(x)` vor. Um eine Quadratwurzel in einem Programm auf diese Art zu verwenden, muss der Befehl wie folgt eingebunden werden:

```
from math import sqrt
```

Möchten Sie noch weitere Funktionen aus dem Modul verwenden können Sie diese durch Komma getrennt importieren.

```
from math import sqrt, exp, log, sin, cos
```

Es ist sogar möglich alle Funktion des Mathematikmoduls einzubinden.

```
from math import *
```

Das sind die folgenden Funktionen: *sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, exp, log (base e), log10 (base 10), sqrt, wie auch die bekannten Zahlen e und pi*

Zusätzlich besteht die Möglichkeit einen eigenen neuen Namen zu definieren. Zum Beispiel wie folgt:

```
1 import math as m
2 # m ist neu der Name des math Moduls
3 v = m.sin(m.pi)
4
5 from math import log as ln
6 v = ln(5)
7
8 from math import sin as s, cos as c, log as ln
9 v = s(x)*c(x) + ln(x)
```

1.3.2 Beispiele mit mathematischen Formeln

In diesem Beispiel betrachten wir die folgende Definition von $\sinh(x)$

$$\sinh(x) = \frac{1}{2}(e^x - e^{-x}) \quad (1.8)$$

Mit Hilfe der Gleichung 1.8 können wir den Ausdruck $\sinh(x)$ auf drei Arten berechnen: i) durch den Aufruf von `math.sinh()`, ii) durch die Berechnung auf der rechten Seite mit Hilfe der Funktion `math.exp()` oder iii) mit der Eulerzahl `e`. Eine Berechnung sieht wie folgt aus:

```
1 from math import sinh, exp, e, pi
2 x = 2*pi
3 r1 = sinh(x)
4 r2 = 0.5*(exp(x) - exp(-x))
5 r3 = 0.5*(e**x - e**(-x))
6 print r1, r2, r3
```

Das Programm liefert wie gewünscht drei identische Resultate.

```
267.744894041 267.744894041 267.744894041
```

Wenn man die Resultate auf 16 Stellen nach dem Komma ausgibt, stimmen jedoch nicht mehr alle Resultate überein.

```
print '%.16f %.16f %.16f' % (r1, r2, r3)
```

Nun erhalten wir die folgende Ausgabe:

```
267.7448940410164369 267.7448940410164369 267.7448940410163232
```

Auf Grund der internen Repräsentation von Dezimalzahlen (Dualsystem) kann es im Bereich der 16ten Stelle nach dem Komma zu Rundungsfehlern kommen. Die beiden mathematischen Ausdrücke $1.0/49 * 49$ und $1.0/51 * 51$ sind beide identisch zu 1. Die numerische Berechnung von

```
print '%.16f %.16f' % (1/49.0*49, 1/51.0*51)
```

ergibt jedoch:

```
0.9999999999999999 1.0000000000000000
```

Falls Rechnungen mit einer höheren Genauigkeit durchgeführt werden müssen, kann das spezielle Modul *decimal* verwendet werden.

1.4 Komplexe Zahlen

Für die Gleichung $x^2 = 2$ lassen sich zwei Lösungen finden, $x_1 = \sqrt{2}$ und $x_2 = -\sqrt{2}$. Etwas schwieriger wird es mit der folgenden Gleichung $x^2 = -2$. Um Lösungen zu dieser Gleichung zu finden, müssen komplexe Zahlen verwendet werden.

Eine komplexe Zahl kann als Paar zweier reellen Zahlen a und b betrachtet werden. Meistens werden komplexe Zahlen mit der Syntax $a + bi$ geschrieben, i wird als imaginäre Einheit bezeichnet. Dabei gilt: $i = \sqrt{-1}$. Eine Lösung der Gleichung $x^2 = -2$ kann algebraisch gefunden werden. Man nutzt $\sqrt{-2} = \sqrt{-1 \cdot 2} = \sqrt{-1} \sqrt{2} = i \cdot \sqrt{2} = \sqrt{2}i$ und erhält zwei Lösungen $x_1 = \sqrt{2}i$ und $x_2 = -\sqrt{2}i$.

Es gelten im weiteren folgende Regeln:

$$\begin{aligned}
 \text{Sei } u &= a + bi \text{ und } v = c + di \\
 u = v &\Rightarrow a = c, b = d \\
 -u &= -a - bi \\
 u^* &= a - bi \text{ (complex conjugate)} \\
 u + v &= (a + c) + (b + d)i \\
 u - v &= (a - c) + (b - d)i \\
 uv &= (ac - bd) + (bc + ad)i \\
 u/v &= \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i \\
 |u| &= \sqrt{a^2 + b^2} \\
 e^{iq} &= \cos(q) + i\sin(q)
 \end{aligned} \tag{1.9}$$

1.4.1 Rechnen mit komplexen Zahlen in Python

Python unterstützt komplexe Zahlen. Die imaginäre Einheit wird in Python mit j bezeichnet, im Gegensatz zu einem i in mathematischen Texten. Die komplexe Zahl $2 - 3i$ wird in Python als $(2 - 3j)$ dargestellt. Es bleibt zu bemerken, dass die imaginäre Zahl i in Python durch zwei Ziffern $1j$ dargestellt wird. Im unten stehenden Beispiel werden komplexe Zahlen definiert und einfache Berechnungen ausgeführt.

```
>>> u = 2.5 + 3
>>> v = 2
>>> w = u + v
>>> w
(4.5+3j)
>>> a = -2
>>> b = 0.5
>>> s = a + b*1j
>>> s
(-2+0.5j)
>>> s*w
(-10.5-3.75j)
>>> s/w
(-0.25641025641025639+0.28205128205128205j)
>>> s.real          # Realteil von s
-2.0
>>> s.imag          # Imaginärteil von s
0.5
>>> s.conjugate()
(-2-0.5j)
```

1.5 Zusammenfassung Kapitel 1

Ein **Python Programm** wird als Textdatei mit der Endung *.py* abgespeichert. Es kann aus der Kommandozeile oder aus dem *Idle Editor* mit (F5) gestartet werden.

Die folgende Anweisung definiert eine Variable.

```
some_variable = obj
```

In diesem Fall steht *some_variable* für einen sinnvollen Variablennamen und *obj* für einen Ausdruck wie z. B. die folgende Formel: $v0 * t$.

In Python gibt es verschiedene Objekttypen, z. B.

Ganze Zahlen (integer),

```
x10 = 3
XYZ = 2
```

Dezimalzahlen (float),

```
max_temperature = 3.0
MinTemp = 1/6.0
```

Zeichenketten (string),


```
a = 'Dieser Text verlauft \n ueber zwei Zeilen.'
b = "Zeichenketten sind umschlossen von Anführungszeichen"
c = """ Zeichenketten innerhalb von drei Anführungszeichen
(nacheinander )
verlaufen über mehrere Zeilen
"""
```

oder **komplexe Zahlen (complex)**

```
a = 2.5 + 3j
real = 6; imag = 3.1
b = complex(real, imag)
```

Mathematische Funktionen können auf drei Arten importiert werden:

```
import math
a = math.sin(math.pi*1.5)
```

oder

```
from math import *
a = sin(pi*1.5)
```

oder

```
from math import sin, pi
a = sin(pi*1.5)
```

Ergebnisse können mit dem print Befehl formatiert ausgegeben und werden.

```
>>> print 'a=%g, b=%12.4E, c=%.2f, d=%5d' % (a, b, c, d)
a=5, b= -5.0000E+00, c=1.99, d= 33
```

1.5.1 Abschliessendes Beispiel: Trajektorie eines Balles

Beim schiefen Wurf fliegt ein Ball entlang einer Trajektorie (Bahnkurve) $y = f(x)$, in der folgenden Formel wird die Luftreibung nicht berücksichtigt. Mit Hilfe von Vektorgeometrie und dem zweiten Satz von Newton lässt sich die folgende Formel für die Bahnkurve eines Balles finden.

$$f(x) = x \tan(\Theta) - \frac{1}{2v_0^2} \frac{gx^2}{\cos^2(\Theta)} + y_0 \quad (1.10)$$

In dieser Formel ist x die horizontale Koordinate, g die Erdbeschleunigung, v_0 die Anfangsgeschwindigkeit. Der Geschwindigkeitsvektor am Anfang bildet mit der x -Achse einen Winkel Θ . Die Anfangsposition ist an der Stelle $(0/y_0)$. Das folgende Programm soll zu einer Position x mit entsprechenden Parametern die Höhe y des Balles ermitteln.

Das Programm besteht aus vier Teilen, einer Initialisierung der Variablen, einem Import von mathematischen Funktionen, einer Umwandlung der Einheiten und der anschliessenden Berechnung der Höhe y anhand der Formel 1.10.

```
1 g = 9.81          # m/s**2
2 v0=15.0           # km/h
3 theta = 60.0      # Winkleinheit Grad
```

```

4 x=0.5          # m
5 y0 = 1.0       # m
6
7 print """\
8 v0      = %.1f km/h
9 theta   = %d Grad
10 y0      = %.1f m
11 x       = %.1f m\
12 """ % (v0, theta, y0, x)
13
14 from math import pi, tan, cos
15 # Umwandlung km/h -> m/s und Grad -> Radian
16 v0 = v0/3.6
17 theta = theta*pi/180
18
19 y = x*tan(theta)-1/(2*v0**2)*g*x**2/((cos(theta))**2) + y0
20 print 'y =%.1fm' % y

```

Das Programm liefert die folgende Ausgabe:

```

>>>
v0 = 15.0 km/h
theta = 60 Grad
y0 = 1.0 m
x = 0.5 m
y =1.6m
>>>

```

1.6 Übungen zu Kapitel 1

Übung 1.1: Umrechnung Sekunden–Jahre

Kann ein neugeborenes Kind in Norwegen eine Milliarde (10^9) Sekunden alt werden? Schreiben Sie ein Programm für die Umrechnung von Sekunden in Jahre. Das Programm soll *seconds2years.py* heissen.

Übung 1.2: Umwandlung in britische Längeneinheiten

Erstellen Sie ein Programm, bei welchem Sie eine Länge x in Meter in die entsprechende Grösse in britische Einheiten (*inches, feet, yards* und *miles*) umrechnen und ausgeben. Benutzen Sie, dass ein inch 2.54 cm entspricht. Ein Fuss ist 12 Inches. Ein Yard ist drei Feet und eine britische Meile entspricht 1760 Yards. Zur Überprüfung ihres Programms können sie die folgenden Angaben verwenden: 640 Meter korrespondieren mit 25196.85 *inches*, 2099.74 *feet*, 699.91 *yards* oder 0.3977 *miles*. Benennen Sie das Programm *length_conversion.py*.

Übung 1.3: Berechnen Sie das Wachstum von Geld auf einer Schweizer Bank

Sei p der Jahreszins einer Bank und A das Startguthaben. Mit Hilfe folgender Formel lässt sich das Guthaben nach n Jahren berechnen: $A \left(1 + \frac{p}{100}\right)^n$ Schreiben Sie ein Programm, das berechnet wie gross ihr Guthaben nach drei Jahren ist, wenn Sie mit 1000 Euro begonnen haben und der Zins immer 5% war. Das Programm soll *bank.guthaben.py* heissen.

Übung 1.4: Finden Sie den Fehler

Jemand hat das folgende Programm geschrieben, das den Ausdruck $\sin(1)$ berechnen soll.

```
x=1; print 'sin(%g)=%g' % (x, sin(x))
```

Schreiben Sie das Programm ab und lassen Sie es laufen. Worin liegt das Problem? Wie kann dieses Programm verbessert werden?

Übung 1.5: Syntaxfehler im Programm

Das folgende Programm soll den Ausdruck $\sin(x)^2 + \cos(x)^2$ für ein beliebiges x berechnen. Suchen Sie die vorhandenen Fehler und speichern Sie das korrigierte Programm unter *sin2_plus_cos2.py* ab.

```
1 from math import sin, cos
2 x = pi/4
3 l_val = sin^2(x) + cos^2(x)
4 print l_val
```

Übung 1.6: Die glockenförmige Gauss-Kurve

$$f(x) = \frac{1}{\sqrt{2\pi}s} \exp \left[-\frac{1}{2} \left(\frac{x-m}{s} \right)^2 \right] \quad (1.11)$$

Die Gauss-Kurve wird häufig in naturwissenschaftlichen oder technischen Aufgaben verwendet. Die Parameter m und s sind reelle Zahlen mit $s > 0$. Schreiben Sie ein Programm, das den Funktionswert an der Stelle $x = 1$ mit den Parametern $m = 0$ und $s = 2$ berechnet. Nennen Sie das Programm *Gauss.Funktion.py*.

Übung 1.7: Wie kocht man ein perfektes Ei?

Beim Kochen eines Eies verlieren die Proteine Wasser und gerinnen schliesslich. Beim Erreichen einer kritischen Temperatur startet diese Reaktion und verläuft schneller und stärker bei weiterer Erhöhung der Temperatur. Das Eiweiss gerinnt bei einer Temperatur von 63°C , das Eigelb beginnt über 70°C zu gerinnen. Für ein weich gekochtes Ei sollte die Temperatur zwischen 63°C und 70°C sein. Für ein hartgekochtes Ei sollte die Temperatur im Innern 70°C erreichen.

Mit Hilfe der folgenden Formel kann die Zeit (in Sekunden) berechnet werden, um eine Temperatur T_y im Zentrum (Eigelb) zu erreichen:

$$t = \frac{M^{2/3} c \rho^{1/3}}{K \pi^2 (4\pi/3)^{2/3}} \ln \left[0.76 \frac{T_0 - T_w}{T_y - T_w} \right] \quad (1.12)$$

Der Parameter M ist die Masse des Eis, ρ ist die Dichte, c die spezifische Wärmekapazität und K die thermische Leitfähigkeit. Vernünftige Werte für ein Ei sind: $M = 47\text{g}$ für ein kleines Ei oder $M = 67\text{g}$ für ein grosses Ei, $\rho = 1.038\text{gcm}^{-3}$, $c = 3.7\text{Jg}^{-1}\text{K}^{-1}$, und $K = 5.4 \cdot 10^{-3}\text{Wcm}^{-1}\text{K}^{-1}$. Ausserdem ist T_w die Temperatur (in Grad Celsius) für kochendes Wasser. T_0 ist die Anfangstemperatur des Eies. Schreiben Sie ein Programm *egg.py*, welches die Kochzeit für ein kleines und ein grosses Ei berechnet. Wählen Sie $T_w = 100$, $T_0 = 4$ (Kühlschrank) oder $T_0 = 20$ (Raumtemperatur) und $T_y = 70$ (hart gekochtes Ei).

Übung 1.8: Programm läuft nicht

Erklären Sie, warum das folgende Programm nicht richtig läuft.

```
C=A+B
A=3
B=2
print C
```

Übung 1.9: Komplexen Zahlen, Terme berechnen

Schreiben Sie ein Programm *komplex_term.py*, das den Real- und Imaginärteil der folgenden Terme berechnet.

$$z_1 = \frac{2-i}{1+i} \quad z_2 = \frac{\sqrt{3} + 2\sqrt{2}i}{\sqrt{3} - \sqrt{2}i} \quad z_3 = \frac{(2+i)(3-2i)(1+2i)}{(1-i)^2}$$

Übung 1.10: Polarkoordinatendarstellung

Berechnen Sie eine Polarkoordinatendarstellung der folgenden komplexen Zahlen. Schreiben Sie dazu ein Python Programm mit dem Namen *polarkordinaten.py*

$$z_1 = -1 + 0i \quad z_2 = \frac{\sqrt{2}}{2}(1+i) \quad z_3 = 2 + 2\sqrt{3}i$$

Übung 1.11: Umwandlung von Polar in Kartesische Koordinaten

Schreiben Sie ein Programm *pol2koord.py*, welches die folgenden komplexen Zahlen in die Form $(a+bi)$ (mit $a, b \in \mathbb{R}$) umwandelt.

$$z_1 = 2e^{i\pi/6} \quad z_2 = 2e^{2\pi i/3} \quad z_3 = 4e^{-i\pi/4}$$

Kapitel 2

Schleifen und Listen

In diesem Kapitel werden Schleifen und Listen behandelt. Sich wiederholende Aufgaben können mit Hilfe einer Programm-Schleife automatisiert werden. Das spezielle Python Konstrukt für Listen kann dazu verwendet werden, um Elemente zu speichern. Ausserdem können verschiedene Operationen auf alle Elemente einer Liste (entsprechend der Reihenfolge ihres Vorkommens) automatisiert angewendet werden.

2.1 Schleifen

Als neue Aufgabe in diesem Kapitel möchten wir eine Konvertierungstabelle erstellen. Damit sollen Temperaturwerte in Grad Celsius in entsprechende Temperaturwerte in Fahrenheit umgerechnet werden können. Eine solche Tabelle könnte wie folgt aussehen:

Grad Celsius	Fahrenheit
-20	-4.0
-15	5.0
-10	14.0
-5	23.0
0	32.0
5	41.0
10	50.0
15	59.0
20	68.0
25	77.0
30	86.0
35	95.0
40	104.0

2.1.1 Naive Lösung: Copy Paste

Wir kennen die Formel 1.4 zur Umrechnung von Grad Celsius in Fahrenheit und können die Werte zeilenweise berechnen.

```
1 C=-20; F=9.0/5*C+32; print C,F
2 C=-15; F=9.0/5*C+32; print C,F
3 C=-10; F=9.0/5*C+32; print C,F
4 C=-5; F=9.0/5*C+32; print C,F
```

```

5 C= 0; F=9.0/5*C+32; print C,F
6 C= 5; F=9.0/5*C+32; print C,F
7 C= 10; F=9.0/5*C+32; print C,F
8 C= 15; F=9.0/5*C+32; print C,F
9 C= 20; F=9.0/5*C+32; print C,F
10 C= 25; F=9.0/5*C+32; print C,F
11 C= 30; F=9.0/5*C+32; print C,F
12 C= 35; F=9.0/5*C+32; print C,F
13 C= 40; F=9.0/5*C+32; print C,F

```

Dieses Programm gibt die gewünschte Tabelle aus. Es ist jedoch sehr unübersichtlich. Immer wieder werden die gleichen Schritte wiederholt. Alle Programmiersprachen haben für wiederholende Aufgaben spezielle Konstrukte, die sogenannten Schleifen. Python kennt zwei Arten von Schleifen, die `while`-Schleife und die `for`-Schleife.

2.1.2 While-Schleife

Die `while`-Schleife kann verwendet werden, um eine Folge (Sequenz) von Anweisungen wiederholt zu bearbeiten, solange eine Bedingung am Anfang erfüllt ist. Am besten kann dies an einem Beispiel veranschaulicht werden. Soll eine Umrechnungstabelle von Grad Celsius nach Fahrenheit erstellt werden, dann wird eigentlich dauernd das Gleiche (siehe oben) gemacht. Aus Temperaturwerten in Grad Celsius werden die entsprechenden Temperaturwerte in Fahrenheit berechnet und ausgegeben. Ein Programm mit einer Schleife könnte wie folgt aussehen: Wir beginnen mit einer Anfangstemperatur von $C = -20^{\circ}\text{C}$. Solange C kleiner gleich 40°C ist, soll der Temperaturwert in Fahrenheit berechnet werden, danach soll eine Ausgabe beider Werte erfolgen. Am Ende dieser Abfolge erhöhen wir die Variable C um 5°C und beginnen wieder mit der Schleife, bis die Bedingung am Anfang nicht mehr erfüllt ist. Diesen Algorithmus kann man in Pseudocode wie folgt beschreiben:

Algorithm 2.1 Prinzip einer While-Schleife

```

1: C = -20
2: while C ≤ 40 do
3:   F = 9.0/5 C + 32
4:   print C, F
5:   set C to C + 5
6: end while

```

Die drei eingerückten Zeilen (nach der Zeile mit 'while') werden so lange ausgeführt, wie die Bedingung $C \leq 40$ erfüllt ist. Auf diese Weise entsteht eine Tabelle mit C und entsprechenden F -Werten. Ein vollständiges Python-Programm für dieses Beispiel sieht wie folgt aus:

```

1 print '-----'
2 C = -20                # Startwert von C
3 dC = 5                 # Erhöhung pro Durchlauf
4 while C <= 40:         # Schleifenkopf
5     F = (9.0/5)*C + 32 # 1. Anweisung in der Schleife
6     print C, F         # 2. Anweisung in der Schleife
7     C = C + dC          # 3. Anweisung in der Schleife
8 print '-----'

```

Ein wichtiges Merkmal der Sprache Python kommt hier zum Vorschein. Der Block von Anweisungen, welcher in jedem Durchlauf der `while`-Schleife ausgeführt werden soll, muss im Programm eingerückt werden. In obigem Beispiel besteht dieser Block aus drei Zeilen. Alle diese Anweisungen müssen exakt die gleiche Einrückung haben (normalerweise 3x die Leertaste oder einmal TAB). Die erste Zeile, welche nicht mehr eingerückt ist, steht ausserhalb der Schleife und wird erst ausgeführt, nachdem die Bedingung im Schleifenkopf nicht mehr erfüllt ist. Ändern Sie das Beispiel von oben wie folgt ab: Rücken Sie die `print` Anweisung auf der letzten Zeile ein (wie die Zeilen innerhalb der Schleife).

Manchmal wird der Doppelpunkt am Ende der `while` Zeile (im Schleifenkopf) vergessen. Dieser Doppelpunkt ist von wesentlicher Bedeutung, er markiert den Beginn der Schleife. Um mit allfälligen Fehlern umgehen zu können, ist es wichtig, dass Sie das obige Programm vollständig verstehen. Deshalb betrachten wir es nun Zeile für Zeile.

In der 2. Zeile definieren wir den Startwert für die Abfolge von Celsius-Temperaturen: $C = -20$. In der 3. Zeile wird die Erhöhung (Schrittweite) dC , um die C innerhalb der Schleife erhöht wird, angegeben. In Zeile 4 wird überprüft, ob die Bedingung erfüllt ist, falls ja werden die Anweisungen in der Schleife ausgeführt. Beim ersten Durchlauf ist $C = -20$, das heisst die Anweisungen innerhalb der Schleife werden ausgeführt. In Zeile 5 wird die Temperatur in Fahrenheit berechnet. In Zeile 6 werden C und F ausgegeben. In Zeile 7 wird nun C erhöht, C erhält neu den Wert -15. Das Programm überprüft die Schleifenbedingung und wiederholt die Zeilen 5-7 solange, bis C grösser als 40 ist.

Einsteigern bereitet die Zeile 7 manchmal Mühe.

```
C = C + dC
```

Aus einer mathematischen Sichtweise sieht diese Zeile falsch aus. **Eine Anweisung mit einem = Operator wird bei der Programmierung nicht als Gleichung sondern als Zuweisung verstanden.** Die obige Zeile bedeutet, dass man der Speicherstelle mit dem Namen C zuerst den Inhalt von C (also sich selbst) zuordnet und dann den Wert dC addiert. Es ist ganz wichtig, dass Zuweisungen immer von links nach rechts gelesen werden, da sie im Inneren einem zeitlichen Ablauf entsprechen. Ausserdem darf links von einem Gleichheitszeichen nur ein einzelner Variablenbezeichner, aber kein Term stehen. Eine Zuweisung wie die Folgende wird mit einer Fehlermeldung kommentiert.

```
a + 2=5
```

Um Variablen zu erhöhen oder verkleinern gibt es auch eine Kurzschreibweise die wie folgt aussieht. Für die bessere Lesbarkeit von Programmen sollte diese Kurzschreibweise jedoch vermieden werden.

```
C+=dC    # entspricht C = C+dC
C-=dC    # entspricht C = C-dC
C*=dC    # entspricht C = C*dC
C/=dC    # entspricht C = C/dC
```

2.1.3 Logische Ausdrücke

Im vorangegangenen Beispiel war die Bedingung der entsprechenden Schleife, dass der Variablen Wert von C kleiner gleich 40 ist ($C \leq 40$), diese Bedingung kann wahr (True) oder falsch (False) sein. Die folgende Liste zeigt weitere logische Ausdrücke.

```
C==40    # C ist gleich 40
```

```
C!=40  # C ist ungleich 40
C>=40  # C ist grösser gleich 40
C>40   # C ist grösser 40
C<40   # C ist kleiner gleich 40
```

Zusätzlich zu den logischen Vergleichs-Operatoren gibt es den `not` Operator. Er ändert den Zustand `True` zu `False` und `False` zu `True`. Er kann auf jeden logischen Ausdruck angewendet werden. Der folgende Ausdruck `not C== 40` wird wie folgt ausgewertet. Falls die Variable `C` den Wert 1 beinhaltet, liefert der logische Ausdruck `C== 40` `False`. Das `not` vor dem gesamten Ausdruck wandelt den Wert zu `True`. Der Ausdruck `C!= 40` ist äquivalent dazu, jedoch viel einfacher zu lesen.

Logische Ausdrücke können auch mit den Operatoren `and` und `or` verknüpft werden.

```
while x > 0 and y <= 1:
    print x, y
```

Im obigen Fall wird die `print x,y` Anweisung nur ausgeführt, wenn `x` grösser als 0 und `y` kleiner gleich 1 ist. Logische Ausdrücke lassen sich direkt in einer Python Konsole überprüfen und auswerten.

```
>>>x=0; y=1.2
>>>x >= 0 and y < 1
False
>>>x >= 0 or y < 1
True
>>>x > 0 or y > 1
True
>>>x > 0 or not y > 1
False
>>>-1<x<=0    # -1<x and x<=0
True
>>>not (x > 0 or y > 0)
False
```

Der Umgang mit logischen Ausdrücken ist eine häufige Fehlerquelle. Aus diesem Grund sollte man sehr vorsichtig bei der Anwendung und besonders bei Verknüpfungen von logischen Operatoren sein. Am Besten überprüft man programmierte Ausdrücke direkt in einer Python Shell.

2.1.4 Berechnung von Summen

Es gibt viele mathematische Anwendungen mit Summen. Zum Beispiel lassen sich alle trigonometrischen Funktionen als Summen berechnen. Im folgenden Beispiel betrachten wir die Annäherung von $\sin(x)$ mit der Summenformel.

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \quad (2.1)$$

Die Fakultät lässt sich in Python mit `math.factorial(k)` berechnen. Die unendliche Summe in der Gleichung 2.1 entspricht genau $\sin(x)$. Eine endliche Summe entspricht jeweils nur einer Annäherung von $\sin(x)$. Als erstes werden wir die rechte Seite der Gleichung mit allen Summanden bis zu der Ordnung $N = 25$ berechnen. Wir brauchen einen Zähler von 1 bis

25 über alle ungeraden Zahlen. Es braucht eine Variable für die Zwischensummen, z. B. s und zusätzlich wird eine Variable für das Vorzeichen (die Summe ist alternierend $- + - + \dots$) benötigt. Ein mögliches Programm kann wie folgt aussehen:

```

1 x = 1.2      # x Wert für sin(x)
2 N = 25      # Maximale Ordnung
3 k = 1       # Zähler über die ungeraden Zahlen 1,3,5, ...
4 s = x       # Der erste Summand wird gleich zugeordnet
5 sign = 1.0
6 import math
7
8 while k < N:
9     sign = - sign
10    k=k+2
11    term = sign*x**k/math.factorial(k)
12    s = s + term
13
14 print 'sin(%g) = %g (angenähert bis Ordnung %d )' % (x, s, N)

```

Um das Programm zu verstehen, rechnen Sie es am Besten Zeile für Zeile schrittweise auf Papier durch und schreiben sich die Inhalte (Zustände) der Variablen auf.

Beim ersten Durchlauf ist die Schleifenbedingung $k < N$ erfüllt. In Zeile 9 wird der Wert des Vorzeichens für den ersten Summanden berechnet $\text{sign} = -1$. In Zeile 10 wird k auf 3 erhöht ($k = 3$). In Zeile 11 wird der erste Term berechnet $\text{term} = -1.0 \cdot x^3 / (3 \cdot 2 \cdot 1)$ (Achtung: die Variable sign ist vom Typ Float, damit es auf keinen Fall zu einer Integer-Division kommt). In Zeile 12 wird die Zwischensumme s um den neuen Term erhöht (sign ist -1.0 , wir addieren also eine negative Zahl). Die Variable k hat nun den Wert 5, das heisst die Anweisungen innerhalb der Schleife werden nochmals ausgeführt. In diesem Durchlauf enthält die Variable term in Zeile 11 den folgenden Inhalt $\text{term} = 1.0 \cdot x^5 / \text{math.factorial}(5)$, was dem dritten Summanden auf der rechten Seite der Gleichung 2.1 entspricht.

2.2 Listen

Bis jetzt waren Variablen meistens Platzhalter für Zahlen (dezimal oder ganzzahlig). Manchmal bilden Zahlen auch Reihen oder gruppieren sich in Listen, zum Beispiel eine Messreihe von Temperaturen. Eine Python-Liste repräsentiert eine solche Struktur. Die Abbildung 2.1 verdeutlicht den Unterschied zwischen einer Variable, welche als Platzhalter für eine einzelne Zahl steht und einer Variablen, welche als Platzhalter für eine Liste steht. Eine Python-Liste kann beliebige Elemente von unterschiedlichem Typ enthalten. Eine Liste besitzt eine mächtige Funktionalität in Python. Es gibt zahlreiche Operationen, welche auf einzelne oder alle Elemente einer Liste angewendet werden können.

2.2.1 Funktionalität oder Operationen einer Liste

Eine Liste bestehend aus Zahlen, zum Beispiel der Temperaturwerte in Celsius, kann auf einfache Weise einer Variablen C zugeordnet werden.

```
C = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
```

Die Zahlenreihe wird innerhalb von eckigen Klammern, getrennt durch Kommas, definiert. Die Variable C ist ein Platzhalter dieser Liste, welche 13 Elemente beinhaltet. Jedes Element in

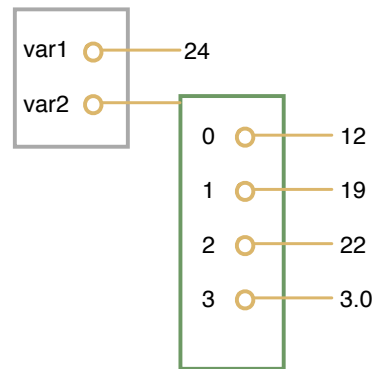


Abbildung 2.1: Schematische Darstellung zweier Variablen, *var1* vom Typ *Integer* und *var2* vom Typ *list*, eine Liste kann Elemente von beliebiger Anzahl und von beliebigem Typ beinhalten. *var2* in dieser Figur kann mit *var2=[12,19,22,3.0]* erzeugt werden.

einer Liste ist mit einem *Index* verknüpft, welcher die Position innerhalb der Liste widerspiegelt. Das erste Element in einer Liste hat den Index 0, das zweite den Index 1, das dritte den Index 2, usw.. Die vorher definierte Liste *C* hat 13 Elemente, beginnend mit einem Element mit Index 0 und endend mit einem Element mit Index 12. Um das vierte Element der Liste auszulesen (also das Element mit dem Index 3) können wir den Ausdruck *c[3]* verwenden.

Elemente können aus Listen gelöscht oder neu eingefügt werden. Entsprechende Funktionen (Funktionalität), welche auf Python-Listen operieren, werden mit Hilfe einer *Punkt-Notation* aufgerufen. *C.append(v)* fügt ein Element *v* am Ende der Liste *C* ein. Mit *C.insert(i,v)* wird ein Element *v* an die Position mit der Nummer *i* in die Liste eingefügt. Die Grösse einer Liste (Anzahl Elemente) kann mit dem Ausdruck *len(C)* ermittelt werden. Im folgenden Beispiel wird eine Liste erzeugt, ein Element angehängt und die Liste ausgegeben. Sie können dieses Beispiel direkt in der Python-Kommandozeile nachvollziehen.

```
>>> C = [-10, -5, 0, 5, 10, 15, 20, 25, 30]    # Liste erzeugen
>>> C.append(35)    # 35 am Ende anfügen
>>> C # Liste ausgeben
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35]
```

Zwei Listen können mit dem Plusoperator (+) zusammen gefügt werden:

```
>>> C = C + [40, 45] # C mit der Liste [40,45] erweitern.
>>> C
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

Im Zusammenhang mit Python-Listen bedeutet die Addition (+ Operation) das Zusammenfügen von Listen, d. h. die zweite Liste wird an die erste Liste angehängt. Mit dem *insert* Operator kann ein Element an einer bestimmten Stelle eingefügt werden.

```
>>> C.insert(0, -15) # Einfügen eines Elements 15 an der Stelle 0
>>> C
[-15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

Mit *del C[i]* wird das angesprochene Element aus der Liste gelöscht. Im folgenden Beispiel wird das 3. Element zweimal gelöscht. Die Einträge (Elemente) -5 und 0 werden aus der Liste entfernt.

```
>>> del C[2] # # löscht das dritte Element aus der Liste
```

```
>>> C
[-15, -10, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> del C[2] # löscht das dritte Element aus der Liste
>>> C
[-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> len(C) # Länge der Liste
11
```

Der Befehl `C.index(10)` gibt den Index des entsprechenden Elements mit dem Wert 10 (beim ersten Auftreten, falls 10 mehrmals vorkommt) in der Liste C an. In diesem Fall kommt 10 als viertes Element vor, also mit dem Index 3.

```
>>> C
[-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> C.index(10) # Liefert den Index des Elementes mit dem Wert 10
3
```

Um zu überprüfen, ob ein Element in einer Liste vorkommt, kann Folgendes geschrieben werden:

```
>>> 10 in C # ist 10 ein Element in C ?
True
```

In Python sind negative Indizes (Indexe) erlaubt. Es wird einfach in die umgekehrte Richtung gezählt. **Es gilt immer: das erste Element hat den Index 0.** Das Element mit dem Index -1 (`C[-1]`) ist das letzte Element in der Liste. `C[-2]` ist das zweitletzte Element in der Liste und so weiter.

```
>>> C
[-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>> C[-1] # Gibt das letzte Element aus
45
>>> C[-2] # Gibt das zweit letzte Element aus
40
```

Das Erstellen einer grossen Liste durch aneinander hängen von Zahlen kann zu einem langwierigen Prozess führen. In Python ist es einfach, eine Liste mit Hilfe einer Schleife automatisiert zu erstellen. Im folgenden Beispiel erstellen wir eine Liste von Temperaturen in einem Bereich von -50 bis 200 und Schritten von 2.5 Grad. Wir beginnen mit einer leeren Liste und füllen diese innerhalb einer Schleife.

```
1 C = []
2 C_value = -50
3 C_max = 200
4 while C_value <= C_max:
5     C.append(C_value)
6     C_value += 2.5
```

Im nächsten Abschnitt werden Sie sehen, dass diese 6 Zeilen durch 3 Zeilen ersetzt werden können.

2.3 For-Schleife

Python `for`-Schleifen können in vielen Situationen sehr gut eingesetzt werden. Häufig soll mit jedem Element einer Liste eine Operation durchgeführt werden. Eine klassische Anwendung dafür ist die Ausgabe aller Elemente einer Liste.

```
1 temperatur = [0, 10, 20, 40, 100]
2 for C in temperatur:
3     print 'Element der Liste: ', C
4 print 'Die Liste temperature hat', len(temperatur), 'Elemente'
```

Der Ausdruck `for C in temperatur` erzeugt eine Schleife über alle Elemente der Liste *temperatur*. In jedem Durchgang der Schleife nimmt die Variable *C* Bezug auf ein Element der Liste *temperatur*. Beim ersten Durchlauf hat *C* den Wert des ersten Elements in der Liste (`temperatur[0]`), beim nächsten Durchlauf bekommt *C* den Wert des nächsten Elements in der Liste (`temperatur[1]`) und so weiter. Beim letzten Durchlauf erhält *C* den Wert des letzten Elements `temperatur[n-1]` (wobei `n=len(temperatur)` ist).

Analog zur `while`-Schleife wird auch bei der `for`-Schleife der Schleifenkopf (die Deklaration der Schleife) mit einem Doppelpunkt abgeschlossen. Die nachfolgenden Zeilen, welche wiederholt ausgeführt werden, müssen eingerückt sein. Im Beispiel von oben wird innerhalb der Schleife jeweils eine Zeile ausgeführt, welche den Wert der Elemente ausgibt.

Betrachten wir den Programmablauf des obigen Beispiels Schritt für Schritt. In der Zeile 1 wird eine Liste mit dem Namen *temperatur* und fünf Elementen definiert. In Zeile 2 wird die Schleifenvariable (auch Iterator genannt) *C* definiert. Im ersten Durchlauf der Schleife verweist *C* auf das erste Element, also das *int*-Objekt mit dem Wert 0. Die `print` Anweisung in Zeile 3 gibt einen Text und den Wert des ersten Elements (0) aus. Es gibt keine weiteren Anweisungen innerhalb des Schleifen-Blocks, folglich beginnt der nächste Durchlauf. *C* bezieht sich dann auf das zweite Element mit dem Inhalt 10. Die Ausgabe (Zeile 3) druckt jetzt den Wert 10 nach dem Text 'Element der Liste:'. Nachdem alle Elemente der Liste abgearbeitet wurden, wird in der Zeile 4 noch die Anzahl Elemente in der Liste ausgegeben. Die ganze Ausgabe des Programms sieht wie folgt aus:

```
Element der Liste: 0
Element der Liste: 10
Element der Liste: 20
Element der Liste: 40
Element der Liste: 100
Die Liste temperature hat 5 Elemente
```

Die korrekte Einrückung von Anweisungen ist entscheidend bei der Programmierung mit der Sprache Python. An dieser Stelle wird empfohlen, dass Sie eigene Programme mit Schleifen schreiben und das Einrücken üben.

2.3.1 Ausgabe in einer Tabelle

Das neu erworbene Wissen zu Python-Listen und `for`-Schleifen kann verwendet werden, um das Beispiel einer Umrechnungstabelle von Grad Celsius nach Fahrenheit eleganter zu programmieren. Ein mögliches Programm kann in einer Liste alle Celsius Werte speichern. Mit Hilfe einer `for`-Schleife kann zu jeder Temperatur (Element in der Liste) die entsprechende Temperatur in Fahrenheit berechnet werden. Sowohl die Temperatur in Grad Celsius wie auch

die berechnete Temperatur in Fahrenheit kann ausgegeben werden. Ein Python Programm könnte wie folgt aussehen:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
for C in Cdegrees:
    F = (9.0/5)*C + 32
    print C, F
```

Die print-Anweisung (in Zeile 4) fügt zwischen der Temperatur in Celsius und der Temperatur in Fahrenheit standardmässig ein Leerzeichen ein. Die so erzeugte Ausgabe sieht nicht besonders schön aus. Eine bessere Formatierung kann durch die Anwendung der `printf`-Formatierung aus Kapitel 1 erreicht werden. Damit kann festgelegt werden, dass sowohl für Werte der Liste F, wie auch für Werte der Liste C fünf Stellen reserviert werden. Ausserdem kann in einer Kopfzeile der Name der Spalte angegeben werden.

```
1 Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
2 print '      C      F'
3 for C in Cdegrees:
4     F = (9.0/5)*C + 32
5     print '%5d %5.1f' % (C, F)
```

Das fünf Zeilen Programm von oben liefert die folgende Tabelle.

```
>>>
      C      F
-20  -4.0
-15   5.0
-10  14.0
-5   23.0
 0   32.0
 5   41.0
10   50.0
15   59.0
20   68.0
25   77.0
30   86.0
35   95.0
40  104.0
```

2.3.2 Die range() Funktion

Es kann sehr langwierig und mühselig sein, viele Elemente explizit in einer Liste, Element für Element, zu definieren. Es macht sicher keinen Spass die vielen Werte für eine gleichmässige Temperaturtabelle (z. B. Cdegrees) von Hand zu tippen. Die `range()` Funktion ist sehr nützlich zur Automatisierung solcher Aufgaben. Im Folgenden werden Beispiele für nützliche Anwendung der `range()` Funktion aufgelistet:

- `range(n)` erzeugt die Zahlen von 0 bis n-1, also 0, 1, 2, ... , n-1
- `range(start, stop, step)` erzeugt eine Sequenz von ganzen Zahlen
start, start+step, start+2*step, start+3*step, ... , start+(n-1)*step

- `range(2, 8, 3)` erzeugt die ganzen Zahlen 2, 5 (aber nicht 8)
- `range(1, 11, 2)` erzeugt die ganzen Zahlen 1, 3, 5, 7, 9
- `range(start, stop)` ist identisch mit `range(start, stop, 1)`
- `range(5, -1, -1)` zählt rückwärts von 5 bis 0, erzeugt also die folgenden ganzen Zahlen 5, 4, 3, 2, 1, 0

Eine Schleife über ganze Zahlen kann wie folgt geschrieben werden:

```
for i in range(start, stop, step):
```

Damit lässt sich in drei Zeilen automatisiert eine Temperaturliste in Grad Celsius erstellen. Die Liste beginnt bei -20, geht in 5 Grad Schritten bis zur Temperatur 40.

```
1 Cdegrees = []
2 for C in range(-20, 45, 5):
3     Cdegrees.append(C)
```

Es ist zu beachten, dass die obere Schranke > 40 ist, damit 40 als letzte Temperatur in der Liste erscheint.

Nun soll eine weitere Liste von Temperaturen erzeugt werden. Diese Liste soll von -10 Grad Celsius in 2.5 Grad Schritten bis zu einer Temperatur von 40 Grad Celsius gehen. Der `range` Befehl kann in diesem Fall nicht direkt genutzt werden. Die Temperaturwerte lassen sich mit der folgenden Formel berechnen. $C = -10 + i \cdot 2.5$ für $i = 0, 1, \dots, 20$. Das folgende Programm erzeugt die beschriebene Liste:

```
1 Cdegrees = []
2 for i in range(0, 21):
3     C = -10 + i*2.5
4     Cdegrees.append(C)
```

2.3.3 For-Schleifen mit Indizes

Anstatt direkt über die Elemente einer Liste zu iterieren,

```
for element in somelist:
    ...
```

kann man über alle Indizes der Liste iterieren und anhand dieser Indizes auf die Elemente der Liste zugreifen.

```
for i in range(len(somelist)):
    element = somelist[i]
    ...
```

Diese Methode der Iteration über Indizes kann sinnvoll genutzt werden, wenn mit mehreren Listen gleichzeitig gearbeitet werden soll. Als Beispiel erzeugen wir zuerst eine Liste mit Grad Celsius Werten `Cdegrees` und eine Liste mit Fahrenheit Werten `Fdegrees`. Als nächstes sollen die Werte aus beiden Listen in verschiedenen Spalten ausgegeben werden. Bei dieser Ausgabe werden Informationen aus beiden Listen benötigt. Das Iterieren über die Indizes ist eine praktische Methode für Fälle mit mehreren Listen.

```

Cdegrees = []
n = 21
C_min = -10
C_max = 40
dC = (C_max - C_min)/float(n-1) # increment in C
for i in range(0, n):
    C = -10 + i*dC
    Cdegrees.append(C)

Fdegrees = []
for C in Cdegrees:
    F = (9.0/5)*C + 32
    Fdegrees.append(F)

for i in range(len(Cdegrees)):
    C = Cdegrees[i]
    F = Fdegrees[i]
    print '%5.1f %5.1f' % (C, F)

```

Anstatt mit einer leeren Liste zu beginnen, können auch alle Elemente mit 0 (Integer) oder 0.0 (Float) initiiert werden. Mit Hilfe der Indizes können die Elemente der Liste anschliessend mit entsprechenden Werten überschrieben werden. Das Erzeugen einer Liste, bestehend aus n Elementen, welche alle auf 0 gesetzt werden, kann wie folgt erreicht werden.

```
somelist = [0]*n
```

Mit dieser Methode kann das obige Programm neu geschrieben werden. Die beiden Listen für Grad Celsius und Grad Fahrenheit werden in der entsprechenden Grösse mit 0 initialisiert. Im zweiten Schritt werden die Werte brechend und mit der und mit der Indizes Zuweisung in die Liste geschrieben. Dieser Weg ist viel effizienter als die Liste ständig mit der append() Funktion zu erweitern.

```

n = 21
C_min = -10
C_max = 40
dC = (C_max - C_min)/float(n-1) # increment in C

Cdegrees = [0]*n
for i in range(len(Cdegrees)):
    Cdegrees[i] = -10 + i*dC

Fdegrees = [0]*n
for i in range(len(Cdegrees)):
    Fdegrees[i] = (9.0/5)*Cdegrees[i] + 32

for i in range(len(Cdegrees)):
    print '%5.1f %5.1f' % (Cdegrees[i], Fdegrees[i])

```

In diesem Beispiel ist es wichtig, dass mit dem Konstrukt `[0]*n` beide Listen die richtige Grösse haben, damit nicht versucht wird, auf ein Element zuzugreifen (Falls ein Index grösser ist als die Liste gibt es eine Fehlermeldung), welches nicht definiert ist.

2.4 Verschachtelte Listen

Sind Elemente von Listen ebenfalls Listen, sprechen wir von verschachtelten Listen. Am besten lassen sich verschachtelte Listen an konkreten Beispielen erklären.

Um die die Temperaturumrechnungstabelle auszugeben, wurden im vorangegangenen Beispiel zwei Listen genutzt. Eine Liste beinhaltete die Temperaturen in Grad Celsius, die andere in Fahrenheit. Es wäre jedoch schön, eine Variable zu definieren, welche die gesamte Tabelle beinhaltet. Dies kann mit dem folgenden Programm erreicht werden.

```
Cdegrees = range(-20, 41, 5) # -20, -15, ..., 35, 40
Fdegrees = [0]*len(Cdegrees)
for i in range(len(Cdegrees)):
    Fdegrees[i] = (9.0/5)*Cdegrees[i] + 32

table = [Cdegrees, Fdegrees]
```

Mit dem Ausdruck `table[0]` wird das erste Element, die ganze Liste `Cdegrees` angesprochen und mit dem Ausdruck `table[0][2]` wird das dritte Element in der Liste `Cdegrees` ausgelesen, das gleiche Element, das auch mit `Cdegrees[2]` angesprochen werden könnte.

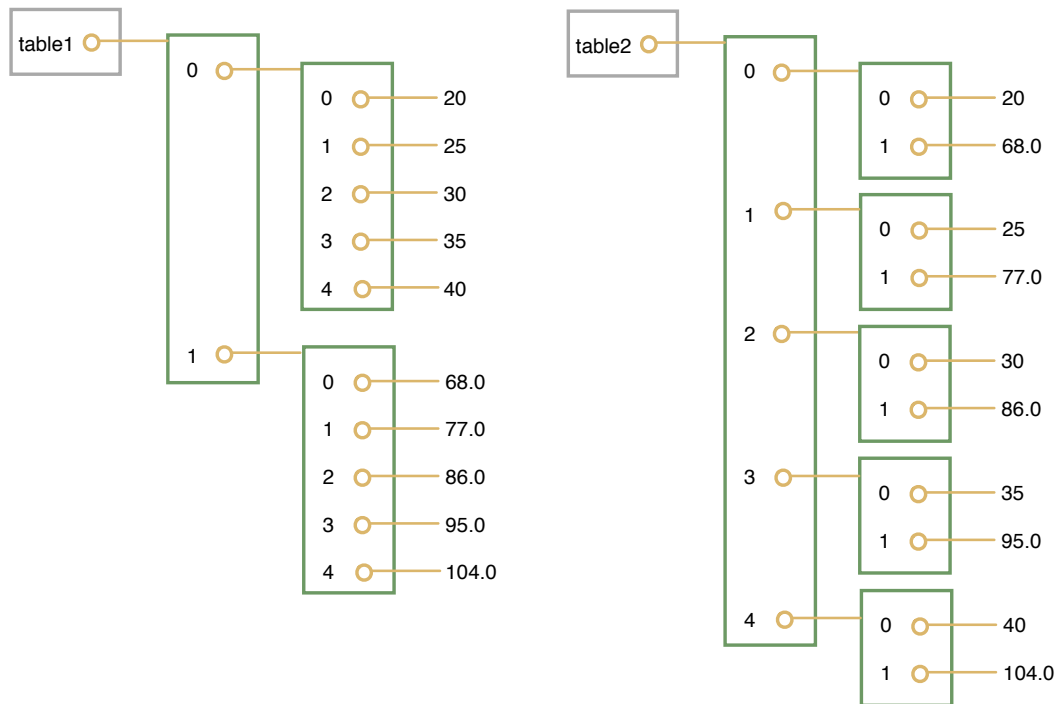


Abbildung 2.2: Schematische Darstellung zweier Möglichkeiten eine Temperaturumrechnungstabelle abzuspeichern. (Links) Die Werte sind kolonnenweise gespeichert, (rechts) als Wertepaare (C,F)

Um die Temperaturumrechnungstabelle für den Fall rechts zu erstellen, müssen die Wertepaare `C, F` in eine Liste abgespeichert werden. Das folgende Beispiel zeigt, wie eine solche verschachtelte Liste erstellt werden kann.

```
Cdegrees = range(-20, 41, 5) # -20, -15, ..., 35, 40
Fdegrees = [0]*len(Cdegrees)
for i in range(len(Cdegrees)):
    Fdegrees[i] = (9.0/5)*Cdegrees[i] + 32
```



```
table = []  
for C, F in zip(Cdegrees, Fdegrees):  
    table.append([C, F])
```

Die beiden Listen werden synchron durchlaufen. In jedem Durchlauf wird aus den beiden Werten C und F eine neue Liste erzeugt und diese Liste als Element der Liste `table` hinzugefügt. Nun beinhaltet `table[0]` eine Liste des ersten Wertepaars $[C, F]$.

2.4.1 Extrahieren von Teillisten (sublist)

Python hat eine schöne intuitive Syntax um Teile einer Liste zu extrahieren und eine Teilliste zu erzeugen.

Zum Beispiel liefert `A[i:]` eine Teilliste, beginnend beim Element mit dem Index i bis zum Ende der Liste von `A`.

```
>>> A = [2, 3.5, 8, 10]  
>>> A[2:]  
[8, 10]
```

`A[i:j]` ergibt eine Teilliste, beginnend mit dem Element mit dem Index i und weiterlaufend bis zum Element mit dem Index $j-1$. Es ist wichtig, dass auch bei den Teillisten j eine obere Schranke ist und das Element `A[j]` selbst nicht zur Rückgabe gehört.

```
>>> A[1:3]  
[3.5, 8]
```

`A[:i]` ist eine Teilliste, beginnend mit dem ersten Element mit dem Index 0 bis zum Element mit dem Index $i-1$.

```
>>> A[:3]  
[2, 3.5, 8]
```

`A[1:-1]` extrahiert alle Elemente ausser das erste und das letzte. Der Ausdruck `A[:]` liefert eine Kopie der ganzen Liste.

```
>>> A[1:-1]  
[3.5, 8]  
>>> A[:]  
[2, 3.5, 8, 10]
```

Bei verschachtelten Listen können Teillisten bezüglich des ersten Indexes erstellt werden. Zum Beispiel folgendermassen:

```
>>> table[4:]  
[[0, 32.0], [5, 41.0], [10, 50.0], [15, 59.0], [20, 68.0],  
[25, 77.0], [30, 86.0], [35, 95.0], [40, 104.0]]
```

2.4.2 Traversieren durch verschachtelte Listen

Ein erstes Beispiel eines Durchlaufes einer verschachtelten Liste wurde im Abschnitt 2.4 gezeigt.

```
for C, F in table:  
    # process C and F
```

Dieses Programm ergibt sich, wenn bekannt ist, dass es sich um eine Liste von zwei gleich grossen Listen handelt [C,F]. Falls die Struktur der verschachtelten Listen nicht bekannt ist, muss ein anderes Vorgehen gewählt werden.

Stellen Sie sich vor, dass in einer Liste Ranglisten von Spielern gespeichert werden. `scores[i]` speichert alle erreichten Resultate eines Spielers. Die Spieler können unterschiedlich oft gespielt haben. Am besten zeigt dies das folgende Beispiel:

```
scores = []
scores.append([12, 16, 11, 12]) # player no. 0
scores.append([9]) # player no. 1
scores.append([6, 9, 11, 14, 17, 15, 14, 20]) # player no. 2
```

Die Liste `scores` beinhaltet drei Elemente, welche die Resultate von drei Spielern repräsentieren. Das Element mit der Nummer k aus der Liste `scores[p]` repräsentiert das Spielresultat mit der Nummer k des Spielers mit dem Index p . In einem allgemeinen Fall hat man n Spieler mit einer unterschiedlichen Anzahl von Spielresultaten.

Es braucht eine doppelte Schleife um die oben definierte verschachtelte Liste zu durchlaufen.

```
for p in range(len(scores)):
    for g in range(len(scores[p])):
        score = scores[p][g]
        print '%4d' % score,
    print
```

Das Komma am Ende der `print` Zeile verhindert einen Zeilenumbruch. Eine alternative Implementierung könnte wie folgt aussehen:

```
for player in scores:
    for game in player:
        print '%4d' % game,
    print
```

2.4.3 Zusammenfassung Kapitel 2

While-Schleifen: Schleifen werden benutzt, um wiederkehrende Programmsequenzen mehrmals automatisiert aufzurufen. Alle Anweisungen, welche innerhalb der Schleife sind, müssen in Python gleich eingerückt sein (normalerweise 3x die Leertaste). Eine While-Schleife wird solange ausgeführt, wie die Schleifenbedingung am Anfang erfüllt ist, d.h. der logische Ausdruck den Wert `True` hat.

```
>>> t=0; dt =0.5;T=2
>>> while t<T:
...     print t
...     t +=dt
...
0
0.5
1.0
1.5
```

Listen: Listen werden verwendet um eine Anzahl von Werten oder Variablen in einer Sequenz mit einer Ordnung (Indizes) zu speichern.

```
>>> meineListe = [t,dt,T, 'daten.dat', 100]
```

Ein Listenelement kann von jeglichem Typ sein, z.B. Ganze Zahlen, Dezimalzahlen, Zeichenketten, Funktionen, andere Listen, usw. Die Tabelle 2.1

Tabelle 2.1: Zusammenfassung der wichtigen Operationen auf Listen

Python Syntax	Bedeutung
<code>a = []</code>	Initialisierung einer leeren Liste
<code>a = [1, 4.4, 'run.py']</code>	Initialisierung einer Liste mit den entspr. Elementen
<code>a.append(elem)</code>	Der Liste a wird ein Objekt elem angehängt
<code>a + [1,3]</code>	Zwei Listen zusammenfügen (addieren)
<code>a.insert(i, e)</code>	Das Element e wird an der Stelle i eingefügt
<code>a[3]</code>	Das 4. Element wird referenziert (angesprochen)
<code>a[0]</code>	Das erste Element wird referenziert
<code>a[-1]</code>	Das letzte Element wird referenziert
<code>a[1:3]</code>	Eine Kopie einer Teilliste wird erstellt
<code>del a[3]</code>	Das 4. Element wird gelöscht
<code>a.remove(e)</code>	Das Element e wird aus der Liste entfernt
<code>a.index('run.py')</code>	Der Index zum Element 'run.py' wird ermittelt
<code>'run.py' in a</code>	Es wird überprüft, ob das Element 'run.py' in der Liste a vorkommt
<code>a.count(v)</code>	Alle Elemente mit dem Inhalt v werden gezählt
<code>a.reverse()</code>	Invertiert die Reihenfolge der Elemente
<code>len(a)</code>	Anzahl Elemente der Liste a
<code>min(a)</code>	Das kleinste Element der Liste a
<code>max(a)</code>	Das Grösste Element der Liste a
<code>sum(a)</code>	Die Summe aller Elemente der Liste
<code>sorted(a)</code>	Gibt eine sortierte Liste a aus

Verschachtelte Listen: Wenn Elemente einer Liste wiederum Listen sind, sprechen wir von verschachtelten Listen. Der folgende Python-Shell Ausdruck zeigt eine Traversierung durch geschachtelte Listen:

```
>>> n1 = [[0, 0, 1], [-1, -1, 2], [-10, 10, 5]]
>>> n1[0]
[0, 0, 1]
>>> n1[-1]
[-10, 10, 5]
>>> n1[0][2]
1
>>> n1[-1][0]
-10
>>> for p in n1:
...     print p
...
[0, 0, 1]
```

```

[-1, -1, 2]
[-10, 10, 5]
>>> for a, b, c in nl:
...     print '%3d %3d %3d' % (a, b, c) ...
    0    0    1
   -1   -1    2
  -10   10    5

```

For-Schleifen: For-Schleifen werden eingesetzt, wenn über alle Elemente einer Liste iteriert werden soll, d.h. wenn mit allen Elementen einer Liste etwas gemacht werden soll. Das untenstehende Beispiel gibt alle Elemente einer Liste aus.

```

>>> for elem in [10, 20, 25, 27, 28.5]:
...     print elem,
...
10 20 25 27 28.5

```

Das angehängte Komma am Ende der print Zeile verhindert einen Zeilenumbruch. Alle Elemente der Liste werden in einer Zeile ausgegeben.

Die range-Funktion wird häufig genutzt, wenn eine Schleife über eine Sequenz von ganzen implementiert werden soll. Der Aufruf sieht wie folgt aus: `range(start, stop, step)`. Das Ende wird als obere Grenze definiert und folglich nicht erreicht (z. B. `range(3,7)` liefert die Liste `[3,4,5,6]`).

```

>>> for elem in range(1, 5, 2):
...     print elem,
...
1 3
>>> range(1, 5, 2)
[1, 3]

```

Eine mathematische Summe kann sehr kompakt mit Python berechnet werden. Die Summe $\sum_{j=M}^N q(i)$ mit $q(i)$ als mathematischem Term mit einer Abhängigkeit zu Index i kann für $q(i) = 1/i^2$ wie folgt definiert werden:

```

s = 0.0 # Teilsumme vom Typ Gleitkommazahl
for j in range(M, N+1, 1):
    s += 1.0/j**2

```

2.4.4 Übungen zu Kapitel 2

Übung 2.1: Fahrenheit-Celsius Umwandlungstabelle

Schreiben Sie ein Programm, das in einer ersten Spalte Temperaturwerte in Fahrenheit und in einer zweiten Spalte die entsprechenden Temperaturwerte in Celsius ausgibt. Der Temperaturbereich der ersten Spalte soll von 0 bis 100 Fahrenheit in 10er Schritten definiert werden. Nennen Sie das Programm `f2c_table_while.py`.

Übung 2.2: Erzeugung von ungeraden Nummern

Schreiben Sie ein Programm, das alle ungeraden Nummern von 1 bis n erzeugt. Weisen Sie n in der ersten Zeile einen Wert zwischen 20 und 100 zu. Benutzen Sie eine while-Schleife um die Nummern zu berechnen. Nennen Sie das Programm *ungerade.py*

Übung 2.3: Ungerade Nummern in einer Liste speichern

Ändern Sie das Programm 2.4.4 so ab, dass die erzeugten ungeraden Nummern in einer Liste gespeichert werden. Beginnen Sie mit einer leeren Liste und fügen sie innerhalb der Schleife die berechnete Nummer der bestehenden Liste an. Geben Sie am Ende die gesamte Liste aus. Nennen Sie das Programm *ungerade_list1.py*.

Übung 2.4: Funktionstabelle erstellen

Schreiben Sie ein Programm, das eine Tabelle ausdruckt. In der ersten Spalte soll die Variable t und in der zweiten Spalte soll der Funktionswert $y(t) = v_0 t - 0.5gt^2$ ausgegeben werden. Teilen Sie das Zeitintervall $[0, 2v_0/g]$ in n gleiche Teile ein. Benutzen Sie $v_0 = 1, g = 9.81$ und $n = 11$. Nennen Sie das Programm *ball_tabelle1.py*.

Übung 2.5: Programm beschreiben

Welche Ausgabe liefert das folgende Programm ?

```

1 a = [1, 3, 5, 7, 11]
2 b = [13, 17]
3 c=a+b
4 print c
5 b[0] = -1
6 d = [e+1 for e in a] # Alle Elemente in a werden um 1 erhoeht
7 print d
8 d.append(b[0] + 1)
9 d.append(b[-1] + 1)
10 print d[-2:]

```

Übung 2.6: x-Koordinaten mit gleichem Abstand

Es sollen x-Koordinaten zwischen 1 und 2 erzeugt werden. Die x-Koordinaten sollen einen Abstand von 0.01 haben. Die Koordinaten sollen in eine Liste $k1$ geschrieben werden. Das erste Element dieser Liste soll den Wert 1.0 haben, das letzte Element den Wert 2.0. Nennen Sie das Programm *koord1.py*

Übung 2.7: Berechnung einer mathematischen Summe

```

1 s=0; k=1; M=100
2 while k < M:
3     s += 1.0/k
4 print s

```

Im oben stehenden Programm sollte folgende Summe $s = \sum_{k=1}^M \frac{1}{k}$ berechnet werden. Wo liegt der Fehler? Falls Sie das Programm starten, passiert nichts. Drücken Sie Ctrl-c (Ctrl-Taste und c) um das nicht korrekt funktionierende Programm zu beenden. Schreiben Sie ein korrektes Programm und speichern Sie dieses unter dem Namen *sum_while.py* ab.

Übung 2.8: Berechnung einer Summe mit einer for-Schleife

Berechnen Sie die folgende Summe $s = \sum_{k=1}^{55} \frac{1}{k^2}$ mit Hilfe einer for-Schleife. Geben Sie die einzelnen Summanden und den entsprechenden Index aus. Geben Sie am Ende ihrer Berechnung die Summe aus. Nennen Sie das Programm `sum_while2.py`

Übung 2.9: Berechnung eines beliebigen Polynoms

Gegeben sind $n + 1$ Nullstellen r_0, r_1, \dots, r_n eines Polynoms $p(x)$ der Ordnung $n + 1$; $p(x)$ kann mit der Produktformel berechnet werden.

$$p(x) = \prod_{i=0}^n (x - r_0)(x - r_1) \dots (x - r_{n-1})(x - r_n)$$

Speichern Sie die Nullstellen in einer Liste. Berechnen Sie mit Hilfe einer Schleife den Wert des Polynoms. Benennen Sie das Programm `poly1.py` und testen Sie es mit den Nullstellen -1, 1 und 2.

Übung 2.10: Summe aller Elemente

Schreiben Sie ein Programm `sum.py`, das alle Elemente in der Liste `l=[1,3,5,-5,7,23,-13,-33,27.5]` summiert. Überprüfen Sie ihr Programm mit der Python-Funktion `sum()`.

```
sum(l)
```

Übung 2.11: Verschachtelte Listen

Es sei q eine verschachtelte Liste.

```
q = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h']]
```

Extrahieren Sie mit Hilfe von Indizes i) den Buchstaben 'a', ii) die Liste ['d', 'e', 'f'], iii) das letzte Element 'h', iv) das Element 'd'. Erklären Sie warum `q[-1][-2]` den Wert g liefert.

Übung 2.12: logische Ausdrücke

Bestimmen Sie die Werte aller logischen Ausdrücke in folgender Liste:

```
C = 41
C == 40
C != 40 and C < 41
C!=40 or C<41
not C == 40
not C > 40
C <= 41
not False
True and False
False or True
False or False or False
True and True and False
False == 0
True == 0
True == 1
```

Achtung: Es macht nur Sinn die ganzen Zahlen 0 und 1 mit True oder False zu vergleichen.

Übung 2.13: Verschachtelte Listen II

Lesen Sie das folgende Beispiel und versuchen Sie vorausszusagen, was die Ausgabe dieses Programms ist.

```

1 n=3
2 for i in range(-1, n):
3     if i != 0:
4         print i
5
6 for i in range(1, 13, 2*n):
7     for j in range(n):
8         print i, j
9
10 for i in range(1, n+1):
11     for j in range(i):
12         if j:
13             print i, j
14
15 for i in range(1, 13, 2*n):
16     for j in range(0, i, 2):
17         for k in range(2, j, 1):
18             b=i>j>k
19             if b:
20                 print i, j, k

```

Übung 2.14: Pi approximieren

Nutzen Sie die folgende Formel um Pi anzunähern. Nennen Sie ihr Programm *pi1.py*.

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{\pi}{4}$$

Übung 2.15: Pi approximieren

Schreiben Sie ein Programm *pi2.py*, das die Kreiszahl Pi mit Hilfe des folgenden Kettenbruchs approximiert.

$$\frac{4}{\pi} = 1 + \frac{1^2}{3 + \frac{2^2}{5 + \frac{3^2}{7 + \frac{4^2}{9 + \frac{5^2}{\ddots}}}}}$$

Übung 2.16: sin(x) approximieren

Berechnen Sie sin(x) mit einer Genauigkeit von 0.0001 mit Hilfe der folgenden Produktformel. Nennen Sie ihr Programm *sin.py*. Wählen Sie für x $\pi/6$

$$\sin(x) = x \prod_{k=1}^{\infty} \left(1 - \frac{x^2}{k^2 \pi^2} \right)$$

Kapitel 3

Funktionen und Verzweigungen

In diesem Kapitel werden zwei fundamentale Programmierkonzepte besprochen, die Funktionen und die Verzweigungen. Funktionen helfen Programme zu strukturierung und damit besser lesbar zu machen. Mit Hilfe von Verzweigungen kann der Programmfluss gesteuert werden.

3.1 Funktionen

Eine Python-Funktion fasst eine Sequenz von Anweisungen in einem aufrufbaren Unterprogramm zusammen. Eine Funktion kann an einer beliebigen Stelle im Programm beliebig oft ausgeführt werden. Programme lassen sich mit Hilfe von Funktionen in einfachere Unterprogramme mit einer klaren Funktionalität unterteilen. Die Vorgehensweise, ein Programm in immer kleinere Teile aufzuteilen, ist auch unter den Begriffen 'Schrittweise Verfeinerung' oder 'funktionale Dekomposition' bekannt.

Im Allgemeinen helfen Funktionen dabei, dass gleiche Programmteile nur an einem Ort stehen und an verschiedenen Stellen genutzt werden können. Damit soll ein Duplizieren von gleichen Programmsequenzen verhindert werden. Bei allfälligen Änderungen zu einem späteren Zeitpunkt müssen Änderungen nur an einer Stelle durchgeführt werden.

Bis jetzt haben wir verschiedene interne Python-Funktionen, wie z. B. `math.sqrt()`, `range()` oder `len()` kennen gelernt. In diesem Abschnitt werden wir auf selbst definierte Funktionen eingehen.

3.1.1 Funktionen mit einer Variablen

Zum Einstieg erstellen wir eine Python-Funktion, die eine mathematische Funktion auswertet. Wir betrachten dazu die bekannte Umrechnung von Grad Celsius nach Grad Fahrenheit, $F(C) = \frac{9}{5} \cdot C + 32$. Der Python-Funktion `F` muss ein Celsius Wert übergeben werden. Diese Funktion sieht in Python wie folgt aus:

```
1 def F(C):  
2     return (9.0/5)*C + 32
```

Jede Python-Funktion beginnt mit dem Schlüsselwort `def`. Es folgt der Name der Funktion und anschliessend die zu übergebenden Argumente innerhalb einer Klammer. Im obigen Fall gibt es nur ein Argument `C`. Der Funktionskopf endet mit einem Doppelpunkt. Im Funktionsblock können die übergebenen Argumente als Variablen angesprochen werden. Alle Anweisungen im Funktionsblock müssen eingerückt sein (analog zu einem Schleifenblock). Am Ende eines

Funktionsblocks wird normalerweise ein Wert zurück geliefert, dies geschieht mit Hilfe der `return`-Anweisung.

Um eine selbst definierte Funktion zu nutzen, muss diese irgendwo im Programm aufgerufen werden. Die meisten Funktionen liefern einen Rückgabewert, dieser Wert wird häufig in einer Variablen gespeichert. Im folgenden Programm wird die Rückgabe der Funktion `F()` in der Variable `F1` abgespeichert.

```
a = 10
F1 = F(a)
temp = F(15.5)
print F(a+1)
sum_temp = F(10) + F(20)
```

Im nächsten Beispiel wird die Funktion `F()` auf alle Elemente einer Liste angewendet. Die Rückgabewerte werden wieder in einer Liste mit dem Namen `Fdegrees` gespeichert.

```
Fdegrees = [F(C) for C in Cdegrees]
```

Im folgenden Beispiel wird anstelle einer Dezimalzahl ein formatierter Text (Zeichenkette) zurückgegeben. Das Beispiel zeigt zusätzlich, dass Funktionen auch im interaktiven Modus, der Python Shell, definiert werden können.

```
>>> def F2(C):
...     F_wert = 9.0/5.0*C+ 32
...     return "%.1f Grad Celsius entspricht "\
...           "%.1f Grad Fahrenheit" % (C,F_wert)
...
>>> s1 = F2(21)
>>> s1
'21.0 Grad Celsius entspricht 69.8 Grad Fahrenheit'
```

3.1.2 Lokale und Globale Variablen

Lassen Sie uns die Funktion `F2()` aus dem vorangegangenen Abschnitt etwas genauer betrachten. Die Variable mit dem Namen `F_wert` ist eine lokale Variable, da sie innerhalb einer Funktion definiert wurde.

```
>>> c1 = 37.5
>>> s2 = F2(c1)
>>> F_value

NameError: name 'F_value' is not defined
```

Auf die lokale Variable `F_wert` kann ausserhalb der Funktion nicht zugegriffen werden. Falls das trotzdem versucht wird, liefert Python die folgende Fehlermeldung:

`NameError: name 'F_value' is not defined` zurück. Globale Variablen, welche ausserhalb von Funktionen definiert werden (im obigen Beispiel sind das `c1` und `s2`), können an jeder Stelle im Programm angesprochen werden. Lokale Variablen haben nur Gültigkeit innerhalb der Funktion, in welcher sie definiert wurden. Um etwas mehr über diese Tatsache zu lernen, können wir die folgende interaktive Session betrachten:

```

>>> def F3(C):
...     F_wert = 9.0/5.0*C+32
...     print "In F3: C=%s F_wert=%s r=%s" % (C, F_wert, r)
...     return "%.1f Grad Celsius entspricht "\
...     "%.1f Grad Fahrenheit" % (C, F_wert)
...
>>> C = 60 # definiert eine globale Variable C
>>> r = 21 # definiert eine globale Variable r
>>> s3 = F3(r)
In F3: C=21 F_wert=69.8 r=21
>>> s3
'21.0 Grad Celsius entspricht 69.8 Grad Fahrenheit'
>>> C
60

```

Innerhalb der Funktion F3() werden die Variablen F_wert, C und r ausgegeben. Die Variable r ist global, sie wird nicht innerhalb der Funktion definiert und auch nicht als Argument übergeben. Sie muss vor dem Aufruf der Funktion F3() global definiert sein.

Im folgenden betrachten wir, was geschieht, wenn ein Variablenname mehrmals verwendet wird.

```

1 print sum # sum ist eine interne Python Funktion
2 sum = 500 # Neudefinition des Namens sum
3 print sum # sum ist nun eine globale Variable
4
5 def myfunc(n):
6     sum = n + 1
7     print sum # sum ist an dieser Stelle eine lokale Variable
8     return sum
9
10 sum = myfunc(2) + 1 # neuer Wert für sum (globale Variable)
11 print sum

```

In der ersten Zeile wird der Inhalt einer Variablen sum ausgegeben. Python sucht nach einer globalen Variablen sum und findet eine interne Python-Funktion mit dem Namen sum. Es erscheint die folgende Ausgabe <built-in function sum>. In der zweiten Zeile wird der gleiche Variablenname nochmals verwendet und der Variablen sum den Wert 500 zugeordnet. In der dritten Zeile wird der Inhalt der neu definierten Variablen ausgegeben.

In den Zeilen 5-8 wird eine Funktion mit dem Namen myfunc definiert, diese verwendet intern die Variable sum. In der Zeile 10 wird die Funktion myfunc(2) mit dem Argument 2 aufgerufen. Innerhalb dieser Funktion wird der lokalen Variable sum= n+1 (also 2+1) zugeordnet und in der folgenden Zeile mit print sum ausgegeben. Innerhalb des Funktionsblocks (Zeile 6-8) hat sum den Wert 3 und nicht den Wert 500 der global definierten Variable sum.

Nach dem Aufruf von myfunc(2) (Zeile 10) wird zum Rückgabewert 3 die Zahl 1 addiert und der global definierten Variable sum zugeordnet. Die letzte Zeile gibt dann die Zahl 4 aus.

Globale Variablen können innerhalb von Funktionen gelesen werden, sie können jedoch nicht verändert werden. Um eine globale Variable innerhalb einer Funktion zu verändern, kann das Schlüsselwort global verwendet werden.

```

1 a= 20;b=-2.5      # globale Variable
2
3 def f1(x):
4     a=21           # Neue lokale Variable
5     return a*x + b
6
7 print a           # 20 wird ausgegeben
8
9 def f2(x):
10    global a
11    a=21           # die globale Variable wird verändert
12    return a*x + b
13
14 f1(3); print a    # 20 wird ausgegeben
15 f2(3); print a    # 21 wird ausgegeben

```

Im obigen Beispiel sind $f1()$ und $f2()$ Funktionen, welche Funktionswerte einer mathematisch linearen Funktion ($a \cdot x + b$) berechnen. Innerhalb der Funktion $f1()$ wird der Parameter a als lokale Variable definiert. Innerhalb der Funktion $f2()$ wird der globalen Variablen a der Wert 21 zugeordnet. Nach dem Aufruf der Funktion $f1()$ besitzt a den Wert 20, welcher in der ersten Zeile global definiert wurde. Im zweiten Fall verändert die Funktion $f2()$ die globale Variable a . Auch ausserhalb der Funktion hat die Variable a nun den Wert 21.

3.1.3 Mehrere Übergabeparameter

Einer Python-Funktion können mehrere Argumente übergeben werden. Im folgenden Beispiel soll die Funktion $yfunc()$ die vertikale Position eines Balles berechnen, dazu wird der Funktion die Zeit t und die Anfangsgeschwindigkeit $v0$ als Argument übergeben.

```

1 def yfunc(t, v0):
2     g = 9.81
3     return v0*t - 0.5*g*t**2

```

Die Gravitationskonstante g wird als lokale Variable innerhalb der Funktion definiert. Im folgenden werden verschiedene korrekte Aufrufe der Python-Funktion aufgezeigt:

```

1 y = yfunc(0.1, 6)
2 y = yfunc(0.1, v0=6)
3 y = yfunc(t=0.1, v0=6)
4 y = yfunc(v0=6, t=0.1)

```

Es besteht die Möglichkeit, dass nur die benötigten Werte einer Python-Funktion übergeben werden. Dabei muss beachtet werden, dass die Reihenfolge der Argumente eine Rolle spielt. Falls die Argumente mit Variablennamen übergeben werden, spielt die Reihenfolge keine Rolle mehr.

Aus mathematischer Sicht kann argumentiert werden, dass $yfunc(t)$ eine Funktion von genau einer unabhängigen Variable, der Zeit, sein soll. Dies kann wie folgt realisiert werden:

```

1 def yfunc(t):
2     g = 9.81
3     return v0*t - 0.5*g*t**2

```

Es ist jedoch zu beachten, dass bevor die Funktion `yfunc(t)` ein erstes Mal aufgerufen wird, die Variable `v0` global definiert sein muss. Sonst erhalten Sie die Fehlermeldung `NameError: global name 'v0' is not defined`. Das folgende interaktive Beispiel verdeutlicht diese Problematik:

```
>>> def yfunc(t):  
...     g = 9.81  
...     return v0*t - 0.5*g*t**2  
...  
>>> yfunc(0.6)  
  
NameError: global name 'v0' is not defined  
>>>
```

Nachdem die Variable `v0` definiert ist, kann die Funktion `yfunc()` aufgerufen werden.

```
>>> v0=5  
>>> yfunc(0.6)  
1.2342
```

Bis jetzt wurden Python-Funktionen im mathematischen Sinne eingesetzt. Die Python-Funktion im nächsten Beispiel soll eine Zahlenliste erstellen, welche durch einen Startwert, einen Endwert und eine Schrittweite definiert wird.

```
1 def makelist(start, stop, inc):  
2     value = start  
3     result = []  
4     while value <= stop:  
5         result.append(value)  
6         value = value + inc.  
7     return result  
8  
9 mylist = makelist(0, 100, 0.2)  
10 print mylist # Ausgabe: 0, 0.2, 0.4, 0.6, ... 99.8, 100
```

Die Funktion `makelist()` verwendet drei Argumente: `start`, `stop` und `inc`. Innerhalb des Funktionsblocks sind diese drei Variablen, wie auch die Variable `result` lokale Variablen. Nur die Variable `mylist` in der Zeile 10 ist eine globale Variable. Sie erhält die generierte Liste als Rückgabe der Funktion `makelist()`.

3.1.4 Mehrere Rückgabeparameter

Python-Funktionen erlauben als Rückgabe mehrere Werte. Im folgenden Beispiel wird die y -Position des Balls und als weiteres Resultat die Geschwindigkeit des Balls zurückgegeben.

```
1 def yfunc(t, v0):  
2     g = 9.81  
3     y = v0*t - 0.5*g*t**2  
4     dydt = v0 - g*t  
5     return y, dydt
```

Diese Funktion kann an einer beliebigen Stelle im Programm aufgerufen werden. Es muss beachtet werden, dass beide Rückgabewerte je einer Variablen zugeordnet werden.

```
position, velocity = yfunc(0.6, 3)
```

Die folgende Anwendung erzeugt eine formatierte Tabelle mit y-Position und Geschwindigkeit des Balls zu verschiedenen Zeiten.

```
1 t_values = [0.05*i for i in range(10)]
2 for t in t_values:
3     pos, vel = yfunc(t, v0=5)
4     print 't=%-10g position=%-10g velocity=%-10g' % (t, pos, vel)
```

Die Formatierungsanweisung %-10g reserviert 10 Stellen, die Zahlen werden in wissenschaftlich kompakter Form ausgegeben. Das Minuszeichen veranlasst, dass alle Zahlen linksbündig ausgegeben werden.

t=0	position=0	velocity=5
t=0.05	position=0.237737	velocity=4.5095
t=0.1	position=0.45095	velocity=4.019
t=0.15	position=0.639638	velocity=3.5285
t=0.2	position=0.8038	velocity=3.038
t=0.25	position=0.943437	velocity=2.5475
t=0.3	position=1.05855	velocity=2.057
t=0.35	position=1.14914	velocity=1.5665
t=0.4	position=1.2152	velocity=1.076
t=0.45	position=1.25674	velocity=0.5855

Als nächstes soll die folgende Summe berechnet werden:

$$L(x; n) = \sum_{i=1}^n \frac{1}{i} \left(\frac{x}{1+x} \right)^i \quad (3.1)$$

Mathematisch kann gezeigt werden, dass die Summe $L(x; n)$ eine Annäherung der Funktion $\ln(1+x)$ ist. Im Grenzfall für n gegen unendlich gilt:

$$\ln(1+x) = \lim_{n \rightarrow \infty} L(x; n)$$

Um eine Summe zu berechnen wird häufig eine Schleife verwendet. Eine lokale Variable speichert die Teilsumme. Ein Programm, welches die Summe $\sum_{i=1}^n c(i)$ berechnet, sieht in Python wie folgt aus:

```
s=0.0
for i in range(1, n+1):
    s += c(i)
```

Im Fall der Summe 3.1 entspricht $c(i)$ dem Term $(1/i)(x/(1+x))^i$

```
1 s=0.0
2 for i in range(1, n+1):
3     s += (1.0/i)*(x/(1.0+x))**i
```

Diese Summe kann mit Hilfe einer Python-Funktion implementiert werden. Als Argumente werden x und n der Funktion übergeben.

```

1 def L(x, n):
2     s=0.0
3     for i in range(1, n+1):
4         s += (1.0/i)*(x/(1.0+x))**i
5     return s

```

Zusätzlich zur berechneten Summe können weitere Informationen über die Güte der verwendeten Approximation von $\ln(1+x)$ durch die Summe $L(x;n)$ zurückgegeben werden. Im folgenden Beispiel wird zusätzlich eine Abschätzung des Fehlers gemacht und der exakte Fehler berechnet und zurückgegeben.

```

1 def L(x, n):
2     s=0.0
3     for i in range(1, n+1):
4         s += (1.0/i)*(x/(1.0+x))**i
5     value_of_sum = s
6     first_neglected_term = (1.0/(n+1))*(x/(1.0+x))**(n+1)
7     from math import log
8     exact_error = log(1+x) - value_of_sum
9     return value_of_sum, first_neglected_term, exact_error
10
11 # typischer Aufruf
12 x = 3.4
13 value, approximate_error, exact_error = L(x, 100)

```

3.1.5 Vordefinierte Übergabeparameter

Die Programmiersprache Python erlaubt, dass im Funktionskopf Argumente mit Standardwerten vordefiniert werden.

```

>>> def einefunktion(arg1, arg2, arg3=True, arg4=0):
...     print arg1, arg2, arg3, arg4

```

Im oben stehenden Beispiel besitzt die Funktion `einefunktion()` vier Argumente. Das dritte und das vierte Argument haben vordefinierte Werte. In einer interaktiven Python-Konsole kann die Funktion `einefunktion` auf verschiedene Weise aufgerufen werden:

```

>>> einefunktion('Hallo', [3, 5])
Hallo [3, 5] True 0
>>> einefunktion('Hallo', [3, 5], arg3='Ja')
Hallo [3, 5] Ja 0
>>> einefunktion('Hallo', [3, 5], 'Ja')
Hallo [3, 5] Ja 0
>>> einefunktion('Hallo', [3, 5], arg3='Ja', arg4=45)
Hallo [3, 5] Ja 45

```

Um die Funktion korrekt aufzurufen, müssen für die ersten zwei Argumente Werte übergeben werden. Falls für das dritte und vierte Argument keine Werte übergeben werden, verwendet die Funktion die vordefinierten Standardwerte.

Ein konkretes Beispiel für eine Python-Funktion mit vordefinierten Übergabeparametern ist die Funktion einer gedämpften Schwingung.

$$f(t; A, a, \omega) = Ae^{-at} \sin(\omega t) \quad (3.2)$$

Diese Funktion kann als Python-Funktion folgendermassen definiert werden:

```
1 from math import pi, exp, sin
2 def f(t, A=1, a=1, omega=2*pi):
3     return A*exp(-a*t)*sin(omega*t)
```

Der Parameter für die Amplitude A , der Parameter für die Dämpfung a , sowie auch der Parameter für die Winkelgeschwindigkeit ω sind vordefiniert. Die abhängige Variable t muss bei einem Aufruf zwingend angegeben werden. Im Folgenden werden fünf Möglichkeiten eines korrekten Funktionsaufrufs gezeigt:

```
v1 = f(0.2)
v2 = f(0.2, omega=1)
v3 = f(1, A=5, omega=pi, a=pi**2)
v4 = f(A=5, a=2, t=0.01, omega=0.1)
v5 = f(0.2, 0.5, 1, 1)
```

Im nächsten Beispiel greifen wir nochmals die Näherungsformel für die Funktion $\ln(1+x)$ nach Gleichung 3.1 auf. Der zweite Parameter definiert eine Art Genauigkeit. Die Summation bricht ab, falls ein Summand kleiner als epsilon ist.

```
def L2(x, epsilon=1.0E-6):
    x = float(x)
    i=1
    term = (1.0/i)*(x/(1+x))**i
    s = term
    while abs(term) > epsilon:      # abs(x) ist |x|
        i += 1
        term = (1.0/i)*(x/(1+x))**i
        s += term
    return s, i
```

Das unten stehende Programm berechnet für einen x Wert verschiedene Näherungen von $\ln(1+x)$ anhand der Summe 3.1.

```
1 from math import log
2 x = 10
3 for k in range(4, 14, 2):
4     epsilon = 10**(-k)
5     approx, n = L2(x, epsilon=epsilon)
6     exact = log(1+x)
7     exact_error = exact - approx
8     print 'epsilon: %5.0e, exact error: %8.2e, n=%d' \
9           % (epsilon, exact_error, n)
```

Ein Vergleich zwischen dem Übergabeparameter epsilon und dem exakt berechneten Fehler zeigt, dass die Grössenordnung des exakten Fehlers mit der Grössenordnung von epsilon

übereinstimmt. Somit ist gezeigt, dass epsilon als Mass für die Genauigkeit dieser Approximation verwendet werden kann.

```
epsilon: 1e-04, exact error: 8.18e-04, n=55
epsilon: 1e-06, exact error: 9.02e-06, n=97
epsilon: 1e-08, exact error: 8.70e-08, n=142
epsilon: 1e-10, exact error: 9.20e-10, n=187
epsilon: 1e-12, exact error: 9.31e-12, n=233
```

3.1.6 Übergabeparameter des Typs Funktion

Besonders für mathematische Anwendungen im Bereich der Analysis ist es oftmals hilfreich, dass Python-Funktionen Variablen zugeordnet werden können. Sie können somit als Argumente an weitere Python-Funktionen übergeben werden. Zum besseren Verständnis wollen wir uns gleich einem Beispiel zuwenden. Numerisch kann eine zweite Ableitung einer Funktion $f(x)$ an der Stelle x wie folgt angenähert werden:

$$f''(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} \quad (3.3)$$

Diese mathematische Formel lässt sich direkt in eine Python-Funktion überführen.

```
1 def diff2(f, x, h=1E-6):
2     r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
3     return r
```

Bei genauen Betrachten erkennt man, dass f in der Zeile 2 als Funktion angewendet wird. Um das obige Programm testen zu können, müssen wir zuerst eine Funktion von einer abhängigen Variablen definieren.

```
1 def g(t):
2     return t**(-6)
3
4 t = 1.2
5 d2g = diff2(g, t)
6 print "g''(%f)=%f" % (t, d2g)
```

Im obigen Beispiel wird $g(t) = t^{-6}$ als Funktion gewählt. An der Stelle $t = 1.2$ soll nun der Wert der zweiten Ableitung approximiert werden. Das Programm liefert den Wert 9.767798. Ein Vergleich zwischen der Wissensmaschine Wolframalpha (Abbildung 3.1) und dem vom Programm berechneten Wert zeigt, dass beide Resultate in den ersten drei Stellen nach dem Komma übereinstimmen. An dieser Stelle muss bemerkt werden, dass zu kleine Werte für h ($h < 1E-8$) zu starken Rundungsfehlern führen können.

3.2 Verzweigung

Das Konstrukt von Verzweigungen wird benötigt, falls der Programmverlauf (je nach Situation) verschiedene Wege nehmen kann. Ein einfaches Beispiel für eine Verzweigung ist die folgende Funktion

$$f(x) = \begin{cases} \sin(x) & \text{für } 0 \leq x \leq \pi \\ 0 & \text{sonst} \end{cases} \quad (3.4)$$

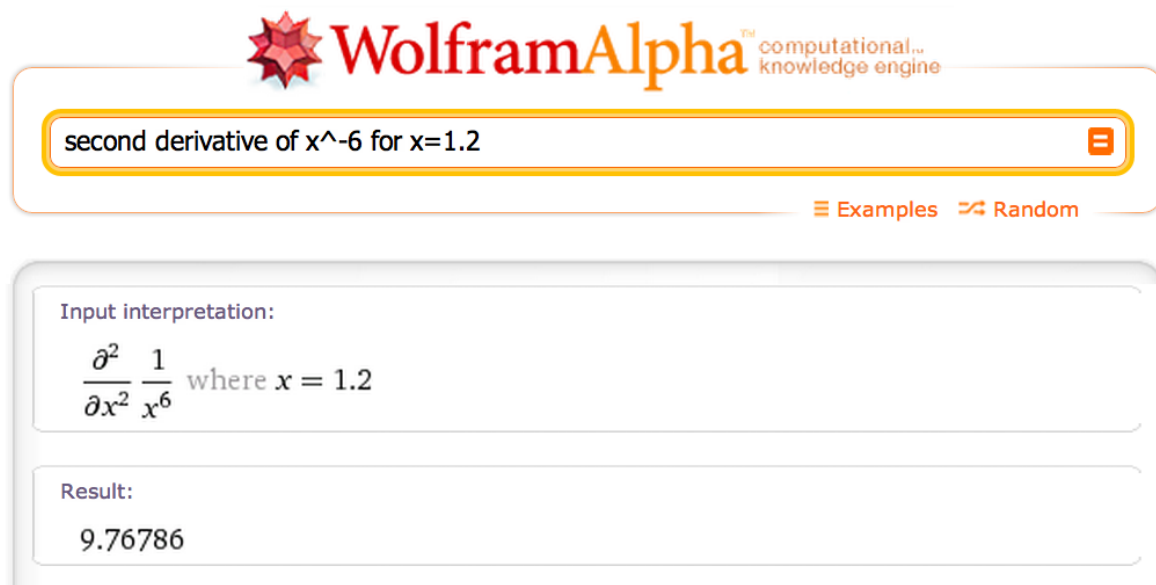


Abbildung 3.1: Vergleich bei WolframAlpha, <http://www.wolframalpha.com/>

In Python kann die oben beschriebene mathematische Funktion wie folgt implementiert werden.

```

1 def f(x):
2     if 0 <= x <= pi:
3         value = sin(x)
4     else:
5         value = 0
6     return value

```

3.2.1 IF-Else Block

Falls die Bedingung erfüllt ist, wird der eingerückte Block nach der `if` Anweisung ausgeführt. Falls die Bedingung nicht erfüllt ist, wird der eingerückte Block nach der `else` Anweisung ausgeführt. Wie bei Schleifen und Python-Funktionen müssen die entsprechenden Blöcke eingerückt sein! Im folgenden Beispiel wird überprüft, ob ein Temperaturwert physikalisch sinnvoll ist:

```

1 if C < -273.15:
2     print '%g degrees Celsius is non-physical!' % C
3     print 'The Fahrenheit temperature will not be computed.'
4 else:
5     F = 9.0/5*C + 32
6     print F
7 print 'end of program'

```

Falls ein Temperaturwert in Grad Celsius kleiner als -273.15 ist, gibt das Programm die Rückmeldung, dass die Angabe unphysikalisch ist, ansonsten wird die Temperatur in Grad Fahrenheit umgerechnet und ausgegeben.

Für komplexere Verzweigungen kann das Schlüsselwort `elif` verwendet werden.

```

if condition1:
    <block of statements>
elif condition2:
    <block of statements>
elif condition3:
    <block of statements>
else:
    <block of statements>
<next statement>

```

Ein ähnlicher Programmfluss kann am folgenden Beispiel konkret aufgezeigt werden. Als Beispiel dient die sogenannte *Hut-Funktion*, sie kann mathematisch wie folgt definiert werden:

$$N(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \leq x < 1 \\ 2 - x, & 1 \leq x < 2 \\ 0, & x \geq 2 \end{cases} \quad (3.5)$$

Eine Implementierung dieser Funktion nutzt das `elif` Schlüsselwort.

```

1 def N(x):
2     if x < 0:
3         return 0.0
4     elif 0 <= x < 1:
5         return x
6     elif 1 <= x < 2:
7         return 2 - x
8     elif x >= 2:
9         return 0.0

```

Die gleiche Funktion kann etwas kompakter wie folgt definiert werden:

```

1 def N(x):
2     if 0 <= x < 1:
3         return x
4     elif 1 <= x < 2:
5         return 2 - x
6     else:
7         return 0

```

3.3 Zusammenfassung Kapitel 3

Selbst definierte Funktionen sind nützlich, (i) wenn eine Sequenz von Anweisungen mehrmals aufgerufen wird, (ii) wenn ein grosses Programm in kleinere Unterprogramme unterteilt wird, um ein bessere Übersicht zu erreichen. Argumente, respektive Übergabeparameter werden zu lokalen Variablen innerhalb des Funktionsblocks. Im Folgenden wird eine Funktion für ein Polynom zweiten Grades definiert:

```

1 # Funktionsdefinition
2 def quadratic_polynomial(x, a, b, c):
3     value = a*x*x + b*x + c

```

```

4     derivative = 2*a*x + b
5     return value, derivative
6
7 # Funktionsaufruf
8 x=1
9 p, dp = quadratic_polynomial(x, 2, 0.5, 1)
10 p, dp = quadratic_polynomial(x=x, a=-4, b=0.5, c=0)

```

Die Reihenfolge der Übergabeparameter spielt eine wichtige Rolle, solange die Argumente nicht mit Variablennamen angegeben werden (name=wert).

Vordefinierte Standardwerte verbessern das Verständnis der Anwendung einer Python-Funktion. Sie ermöglichen eine effiziente und flexible Nutzung.

```

1 from math import exp, sin, pi
2
3 def f(x, A=1, a=1, w=pi):
4     return A*exp(-a*x)*sin(w*x)
5
6 f1 = f(0)
7 x2 = 0.1
8 f2 = f(x2, w=2*pi)
9 f3 = f(x2, w=4*pi, A=10, a=0.1)
10 f4 = f(w=4*pi, A=10, a=0.1, x=x2)

```

Das Beispiel oben zeigt, dass die Reihenfolge der Übergabeparameter keine Rolle spielt, wenn die Variablennamen angegeben werden.

Die Schlüsselwörter `if` und `else` ermöglichen Verzweigungen innerhalb des Programmflusses, so dass je nach Bedingung unterschiedliche Blöcke ausgeführt werden.

```

1 def f(x):
2     if x < 0:
3         value = -1
4     elif x >= 0 and x <= 1:
5         value = x
6     else:
7         value = 1
8     return value

```

3.4 Abschliessendes Beispiel

Ein Integral

$$\int_a^b f(x) dx$$

kann mit dem Simpson Verfahren angenähert werden.

$$\frac{b-a}{3n} \left(f(a) + f(b) + \sum_{i=1}^{n/2} f(a + (2i-1)h) + 2 \sum_{i=1}^{n/2-1} f(a + 2ih) \right) \quad (3.6)$$

Zusätzlich muss $h = (b - a)/n$ gelten und n muss eine gerade ganze Zahl sein. Die Aufgabe liegt darin, eine Funktion `Simpson(f,a,b,n=500)` zu definieren, welche als Rückgabe die Formel 3.6 berechnet. Überprüfen lässt sich die Simpson Methode mit einem bekannten Integral, z. B. mit $\frac{3}{2} \int_0^{\pi} \sin^3 x dx$, welches den exakten Wert 2 hat.

Eine Summe $\sum_{i=M}^N q(i)$ lässt sich wie folgt berechnen:

```
1 s=0
2 for i in range(M, N):
3     s += q(i)
```

Die Simpson () Funktion kann wie folgt implementiert werden:

```
1 def Simpson(f, a, b, n=500):
2     h = (b - a)/float(n)
3     sum1 = 0
4     for i in range(1, n/2 + 1):
5         sum1 += f(a + (2*i-1)*h)
6     sum2 = 0
7     for i in range(1, n/2):
8         sum2 += f(a + 2*i*h)
9     integral = (b-a)/(3*n)*(f(a) + f(b) + 4*sum1 + 2*sum2)
10    return integral
```

Die Funktion $h(x) = \frac{3}{2} \int_0^{\pi} \sin^3 x dx$ kann wie folgt in Python implementiert werden:

```
1 def h(x):
2     return (3./2)*sin(x)**3
```

Im Folgenden wird das Simpsonverfahren genutzt, um das Integral mit unterschiedlichen n Werten zu berechnen.

```
1 from math import sin, pi
2 def application():
3     print 'Integral of 1.5*sin^3 from 0 to pi:'
4     for n in 2, 6, 12, 100, 500:
5         approx = Simpson(h, 0, pi, n)
6         print 'n=%3d, approx=%18.15f, error=%9.2E' \
7             % (n, approx, 2-approx)
```

Die folgende Ausgabe zeigt das Kovergenzverhalten des Simpsonverfahrens.

```
Integral of 1.5*sin^3 from 0 to pi:
n=  2, approx= 3.141592653589793, error=-1.14E+00
n=  6, approx= 1.989171700583579, error= 1.08E-02
n= 12, approx= 1.999489233010781, error= 5.11E-04
n=100, approx= 1.999999902476350, error= 9.75E-08
n=500, approx= 1.99999999844138, error= 1.56E-10
```

3.5 Übungen zu Kapitel 3

Übung 3.1: Umrechnung von Fahrenheit – Celsius

Schreiben Sie eine Funktion $C(F)$, welche Grad Fahrenheit in Celsius umrechnet. Nutzen Sie dazu die folgende mathematische Gleichung: $C = \frac{5}{9}(F - 32)$. Überprüfen Sie ihre Funktion mit der Umkehrfunktion $F(C)$ aus Abschnitt 3.1.1 auf ihre Korrektheit. Vergleichen Sie eine beliebige Temperatur c in Grad Celsius und der zweifachen Umrechnung $C(F(c))$ dieser gewählten Temperatur.

Übung 3.2:

Die Funktion $s(M)$ soll die folgende Summe $s = \sum_{k=1}^M \frac{1}{k^2}$ berechnen. Der Aufruf $s(3)$ soll die Teilsumme mit 3 Summanden berechnen. Nennen Sie das Programm *sum_func.py*

Übung 3.3: Fläche eines Dreiecks

Ein beliebiges Dreieck in der Ebene kann durch die Angabe seiner Eckpunkte eindeutig beschrieben werden. Die Eckpunkte sollen im Gegenuhrzeigersinn nummeriert werden und als Koordinaten (x_1, y_1) , (x_2, y_2) , (x_3, y_3) bekannt sein. Der Flächeninhalt dieses Dreiecks lässt sich mit der folgenden Formel berechnen:

$$A = \frac{1}{2} |x_2 y_3 - x_3 y_2 - x_1 y_3 + x_3 y_1 + x_1 y_2 - x_2 y_1| \quad (3.7)$$

Schreiben Sie eine Funktion *flaeche(vektoren)*, welche den Flächeninhalt eines Dreiecks berechnet und als Dezimalzahl zurück gibt. Der Übergabeparameter *vektoren* soll vom Typ einer geschachtelten Liste sein. Zum Beispiel können die folgenden Vektoren übergeben werden $[[0,0],[1,0],[0,2]]$. Hierbei handelt es sich um ein rechtwinkliges Dreieck mit einem Flächeninhalt von 1. Nennen Sie ihr Programm *flaeche_dreieck.py*.

Übung 3.4: Länge eines Pfades

Um eine Bewegung auf einer Ebene zu beschreiben, kann die Position eines Objektes (x, y) zu verschiedenen Zeitpunkten festgehalten werden. Diese Methode speichert eine Liste von n Punkten, z. B. (x_0, y_0) , (x_1, y_1) , (x_2, y_2) , ..., (x_{n-1}, y_{n-1}) . Die Gesamtlänge des Pfades von (x_0, y_0) nach (x_{n-1}, y_{n-1}) ist die Summe aller Strecken von benachbarten Punkten (Strecke (x_{i-1}, y_{i-1}) nach (x_i, y_i)), $i = 1, \dots, n - 1$)

$$L = \sum_{i=1}^{n-1} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2} \quad (3.8)$$

Die Gleichung 3.8 kann verwendet werden um die Länge zu berechnen. Erstellen Sie eine Funktion *pfadlaenge(punkte)* welche die Länge des Pfades berechnet. Als Übergabeparameter soll eine geschachtelte Liste von der Form $[[x_0, y_0], [x_1, y_1], [x_2, y_2], \dots]$ verwendet werden. Überprüfen Sie ihre Funktion mit dem folgenden Pfad $[[0,0], [1,0],[1,2], [0,2]]$, dessen Gesamtlänge 4 ist. Nennen Sie ihr Programm *pfadlaenge.py*.

Übung 3.5: π approximieren

Die Zahl π ist gleich dem Umfang eines Kreises mit Radius $\frac{1}{2}$. Ein Kreis mit Radius $\frac{1}{2}$ kann durch ein Polygon mit $N + 1$ Punkten geometrisch angenähert werden. Die Länge des Polygons kann mit Hilfe der *pfadlaenge()* Funktion bestimmt werden (siehe Aufgabe 3.4). Berechnen Sie die $N + 1$ Punkte $[x_i, y_i]$ auf einem Kreis mit dem Radius $\frac{1}{2}$ anhand der folgenden

Formeln

$$x_i = \frac{1}{2} \cos(2\pi i/N), \quad y_i = \frac{1}{2} \sin(2\pi i/N), \quad i = 0, \dots, N \quad (3.9)$$

Approximieren Sie π mit $N = 2^k, k = 2, 3, \dots, 10$. Benutzen Sie die `pfadlaenge()` Funktion. Nennen Sie das Programm `pi_approx.py`.

Übung 3.6: Fourier, Annäherung einer Funktion durch eine Summe

Wir betrachten die stückweise konstante Funktion.

$$f(t) = \begin{cases} 1, & 0 < t < T/2 \\ 0, & t = T/2 \\ -1, & T/2 < t < T \end{cases} \quad (3.10)$$

Die Funktion $f(t)$ kann mit der folgenden Summe angenähert werden.

$$S(t; n) = \frac{4}{\pi} \sum_{i=1}^n \frac{1}{2i-1} \sin\left(\frac{2(2i-1)\pi t}{T}\right) \quad (3.11)$$

Schreiben Sie eine Python-Funktion `S(t, n, T)`, welche die Summe 3.11 berechnet. Schreiben Sie eine Python-Funktion `f(t, T)`, um die Funktion $f(t)$ zu berechnen. Untersuchen Sie den Fehler $f(t) - S(t; n)$ in Abhängigkeit von $n = 1, 3, 5, 10, 30, 100$ und $t = 0.0628, 1.57, 3.08$. Geben Sie die Fehler tabellarisch aus. Nennen Sie das Programm `sinsum.py`.

Übung 3.7: Gauss Funktion

Erstellen Sie eine Python-Funktion `gauss(x, m=0, s=1)`, welche die Gauss Funktion berechnet.

$$f(x) = \frac{1}{\sqrt{2\pi}s} \exp\left[-\frac{1}{2} \left(\frac{x-m}{s}\right)^2\right] \quad (3.12)$$

Rufen Sie die Gaussfunktion für die Werte $x = -5, -4.9, -4.8, \dots, 4.9, 5$ auf und geben Sie den Rückgabewert aus. Nennen Sie das Programm `gauss_funktion.py`.

Übung 3.8: numerische Ableitung

Mit der Näherung

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \quad (3.13)$$

kann die erste Ableitung einer Funktion $f(x)$ an der Stelle x berechnet werden. Schreiben Sie eine Funktion `diff(f, x, h=1E-6)`, welche mit Hilfe der Approximation 3.13 die erste Ableitung einer Funktion $f(x)$ an der Stelle x berechnet.

Benutzen Sie die numerische Ableitung für folgende Funktionen: $f(x) = e^x$ für $x = 0$, $f(x) = e^{-2x^2}$ für $x = 0$, $f(x) = \cos(x)$ für $x = 2\pi$ und $f(x) = \ln x$ für $x = 1$. Verwenden Sie in allen Fällen $h = 0.001$. Untersuchen Sie den Fehler bezüglich der exakten Ableitung. Nennen Sie das Programm `diff.py`.

Übung 3.9: Numerische Integration I

Eine Näherung des Integrals der Funktion $f(x)$ über dem Segment $[a, b]$ ist die Fläche unter der Gerade durch die Punkte $(a, f(a))$ und $(b, f(b))$. Erstellen Sie eine Python-Funktion `integral1(f, a, b)`, welche die oben beschriebene Fläche berechnet (Trapez). Das erste Argument soll eine Python-Funktion $f(x)$ sein. Die Argumente a, b definieren die Integrationsgrenzen. Nennen Sie das Programm `int1_f.py`.

Übung 3.10: Numerische Integration II

Nähern Sie ein Integral durch n Trapeze mit gleicher Breite an und schreiben Sie dazu eine Python-Funktion `integral2(f,a,b,n=100)`. Berechnen Sie die Fläche der Trapeze mit Hilfe der Funktion `integral1(f,a,b)` aus der Übung 3.9.

Berechnen Sie die folgende Integrale: $\int_0^{\ln 3} e^x dx$, $\int_0^{\pi} \cos x dx$ und $\int_0^{\pi/2} \sin x dx$ mit $n = 5, 20, 50, 100$

Untersuchen Sie den Fehler bezüglich der exakten Lösung. Nennen Sie das Programm `int2.f.py`.

Übung 3.11: Geschwindigkeit und Beschleunigung

Sei $x(t)$ ein Ort eines sich auf der x -Achse bewegenden Objektes. Die Geschwindigkeit $v(t)$ und die Beschleunigung $a(t)$ lassen sich mit Hilfe der beiden folgenden Formeln annähern:

$$v(t) \approx \frac{x(t + \Delta t) - x(t - \Delta t)}{2\Delta t}, \quad a(t) \approx \frac{x(t + \Delta t) - 2x(t) + x(t - \Delta t)}{\Delta t^2} \quad (3.14)$$

Δt ist ein kleines Zeitintervall. Für den Fall $\Delta t \rightarrow 0$ entsprechen die obigen Annäherungen der ersten und zweiten Ableitung der Ortsfunktion $x(t)$.

Schreiben Sie die Funktion `kinematics(x,t, dt=1E-4)`. Diese soll aus den Ortsdaten x die Position $x(t)$, die Geschwindigkeit v und die Beschleunigung a mit Hilfe der obigen Näherungen berechnen. Verwenden Sie als Testdaten die folgende Ortsfunktion $x(t) = e^{(t-4)^2}$ und wählen Sie als Zeitpunkt $t = 5$ (mit $\Delta t = 10^{-5}$). Nennen Sie das Programm `kinematic1.py`.

Übung 3.12: Geschwindigkeit und Beschleunigung II (x-y-Ebene)

Ein Objekt bewegt sich auf der x - y -Ebene, es ist zu jedem Zeitpunkt t der entsprechende Ort $(x(t), y(t))$ bekannt. Sowohl der Geschwindigkeitsvektor, wie auch der Beschleunigungsvektor lassen sich zu einem beliebigen Zeitpunkt t mit folgenden Formeln annähernd bestimmen:

$$v(t) \approx \left(\frac{x(t + \Delta t) - x(t - \Delta t)}{2\Delta t}, \frac{y(t + \Delta t) - y(t - \Delta t)}{2\Delta t} \right) \quad (3.15)$$

$$a(t) \approx \left(\frac{x(t + \Delta t) - 2x(t) + x(t - \Delta t)}{\Delta t^2}, \frac{y(t + \Delta t) - 2y(t) + y(t - \Delta t)}{\Delta t^2} \right) \quad (3.16)$$

Diese Annäherung ist nur gültig für kleine Zeitintervalle Δt . Im Falle $\Delta t \rightarrow 0$ ist $v(t) = (x'(t), y'(t))$ und $a(t) = (x''(t), y''(t))$ (entsprechend der physikalischen Definitionen von v und a).

Programmieren Sie eine Funktion `kinematics2(r,t, dt=1E-4)`, welche die Geschwindigkeit und Beschleunigung eines Objekts nach den obigen Formeln berechnet. (t soll t und dt soll Δt entsprechen). Die Funktion soll 3 Zweivektoren für den Ort $r(t) = (x(t), y(t))$, die Geschwindigkeit $v(t)$ und die Beschleunigung $a(t)$ zu einem Zeitpunkt t zurückgeben. Testen Sie ihre Funktion an folgender Kreisbewegung $r(t) = (R \cos(\omega t), R \sin(\omega t))$. Berechnen Sie die Geschwindigkeit und Beschleunigung zum Zeitpunkt $t = 1$ mit $R = 1$, $\omega = 2\pi$ und $\Delta t = 10^{-5}$. Nennen Sie das Programm `kinematic2.py`.

Übung 3.13: Max / Min Element in einer Liste

Sei a eine Python-Liste mit gleichen Elementen. Schreiben Sie eine Funktion `max(a)` (später analog eine Funktion `min(a)`), welche das grösste Element in der Liste findet. Sie können wie folgt vorgehen: Initialisieren Sie eine Variable `max_element` mit dem Wert des ersten Elements

in der Liste (`max_element=a[0]`). Als nächstes vergleichen Sie `max_element` mit allen verbleibenden Elementen der Liste (`a[1:]`). Falls ein Element grösser als `max_element` ist, wird dieses neu zu `max_element`. Analog zu dieser Methode kann auch das Minimum gefunden werden. Nennen Sie das Programm *maxmin_list.py*. Sie können die Korrektheit ihres Programms mit den eingebauten Python-Funktionen `min()`, `max()` verifizieren.

Übung 3.14: Heaviside-Stufen-Funktion

Die Heaviside-Stufen-Funktion kann mathematisch wie folgt definiert werden:

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases} \quad (3.17)$$

Schreiben Sie eine Python-Funktion `H(x)` und nennen Sie das Programm *heaviside.py*.

Übung 3.15:

Die folgende Summe ist eine Näherung für $\ln(1+x)$:

$$L(x;n) = \sum_{i=1}^n c_i, \quad c_i = \frac{1}{i} \left(\frac{x}{1+x} \right)^i \quad (3.18)$$

Zwischen c_i und c_{i-1} besteht eine Beziehung der Form $c_i = a c_{i-1}$. Damit kann man schrittweise (Induktion) die Summe berechnen. Es kann mit c_1 begonnen werden. Jeder folgende Term c_2, c_3, \dots, c_i kann in der Summe wie folgt berechnet werden:

```
term = a*term
```

Verändern Sie die Python-Funktion `L2()` aus dem Abschnitt 3.1.5 so, dass diese effiziente Methode genutzt wird. Nennen Sie das Programm *L2_dyn.py*.

Übung 3.16: Sieb des Eratosthenes

Schreiben Sie eine Funktion, welche alle Primzahlen kleiner einer oberen Schranke N findet. Lesen Sie den entsprechenden Wikipedia Artikel und implementieren Sie diesen Algorithmus. Nennen Sie das Programm *finde_prim.py*.

Kapitel 4

Rechnen mit Arrays / Darstellung von Kurven (Plots)

In Kapitel 2 wurden Elemente (Zahlen) in Listen gespeichert. Python-Listen ermöglichen flexibles Bearbeiten der Elemente einer Liste (z. B. einfügen, suchen, löschen, sortieren, usw.). In diesem Kapitel wird ein neues Python-Objekt mit dem Namen `array` vorgestellt. Arrays sind weniger flexibel, mit ihnen kann jedoch viel effizienter gerechnet werden. Besonders bei grossen Zahlenmengen ($n > 200$) lohnt sich der Einsatz von Arrays. Im zweiten Teil werden zahlreiche Beispiele zum Einsatz der `matplotlib` vorgestellt.

4.1 Python Arrays - Numpy

Ein Array Objekt kann als eine spezielle Art Liste betrachtet werden. Dabei müssen alle Elemente eines Arrays vom gleichen Typ sein (z.B. `float`, `integer`, usw.). Zusätzlich muss die Anzahl aller Elemente im Voraus bekannt sein. Ausserdem gehören Arrays nicht zum Standard-Python, es wird ein zusätzliches Modul mit dem Namen *Numerical Python* (`{numpy}`) benötigt. Im Anhang finden sie eine Installationsanleitung. Im Modul `numpy` sind zahlreiche Funktionen, welche direkt auf Arrays (Vektoren und Matrizen) angewendet werden können. Es gibt gute Tutorials zum Arbeiten mit `numpy`. Diese finden Sie unter: <http://scipy.org/>. Empfehlenswert sind die folgenden Beiträge: *NumPy Tutorial*, *NumPy User Guide*, *NumPy Reference*, *Guide to NumPy*, and *NumPy for Matlab Users*. Im Weiteren werden wir uns auf die Teile von `NumPy` konzentrieren, welche für die graphische Darstellung von Funktionen wichtig sind.

Meistens wird das Modul `NumPy` wie folgt importiert:

```
import numpy as np
```

Die folgende Anweisung konvertiert eine Liste `r` in einen Array mit dem Namen `a`

```
a = np.array(r)
```

Häufig soll ein Array der Grösse n mit Nullen (`float`) initialisiert werden, dies kann wie folgt realisiert werden:

```
a = np.zeros(n)
```

Für viele Anwendungen wird eine gleichmässige Verteilung von n Zahlen in einem Wertebereich $[p,q]$ benötigt. Die `numpy`-Funktion `np.linspace()` erzeugt ein solches Array:

```
a = np.linspace(p, q, n)
```

Elemente oder Abschnitte (engl. slice) eines Arrays werden auf die gleiche Weise wie Listen angesprochen. Im Unterschied zu Listen werden nicht Kopien der Elemente zurück gegeben, sondern es wird immer Bezug auf die ursprünglich definierten Elemente genommen (Referenzierung). Dies kann am folgenden Beispiel gezeigt werden:

```
1 b = a[1:-1]
2 b[2] = 0.1
```

b zeigt auf alle Werte des Arrays a ausser auf den ersten Wert. In der Zeile 2 wird b[2] auf den Wert 0.1 gesetzt. Diese Zuweisung ändert auch den Wert von a[3], weil b[2] und a[3] die gleiche Speicherstelle referenzieren.

4.1.1 Koordinaten und Funktionswerte

Mit den beschriebenen Grundoperationen können Arrays erzeugt werden, welche x-Koordinaten und y-Funktionswerte beinhalten. In der folgenden interaktiven Python-Shell werden einem Array x-Koordinaten und einem anderen Array y-Funktionswerte zugeordnet.

```
>>> import math
>>> import numpy as np
>>> n = 7
>>> x = np.linspace(0,1,n)
>>> y = np.zeros(n)
>>> for i in range(n):
...     y[i]=math.sin(x[i])
...
>>> y
array([ 0.          ,  0.16589613,  0.3271947 ,  0.47942554,
        0.6183698,  0.74017685,  0.84147098])
```

Im obigen Beispiel muss ein Array mit der entsprechenden Grösse (Anzahl y-Werte) initialisiert werden ($y = \text{np.zeros}(n)$), bevor Werte in dieses Array geschrieben werden können ($y[i]=\text{math.sin}(x[i])$).

Normalerweise werden für gleichmässig verteilte x-Koordinaten y-Funktionswerte mit Hilfe einer Funktion berechnet. Dies kann mit numpy-Funktionen sehr kompakt gelöst werden.

```
>>> x2 = np.linspace(0, 1, n)
>>> y2 = np.array([f(xi) for xi in x2])
```

f() ist eine Pythonfunktion, die jedem Wert innerhalb eines Definitionsbereichs genau einen Wert zuordnet.

4.2 Kurvendarstellung mit Hilfe der Bibliothek Matplotlib

Im ersten Beispiel soll die Funktion $f(t) = t^2 \exp(-t^2)$ als Graph dargestellt werden. Dazu benötigen wir die Matplot-Bibliothek, sie kann, wie im Anhang beschrieben, installiert werden.

```
1 from matplotlib.pyplot import *
2 from math import *
3 import numpy as np
4
```

```
5 def f(t):  
6     return t**2*exp(-t**2)  
7  
8 t = np.linspace(0, 3, 51)           # 51 Punkte zwischen 0 und 3  
9 y = np.zeros(len(t))               # Array mit gleicher Groesse  
10 for i in xrange(len(t)):  
11     y[i] = f(t[i])                 # Funktionswerte berechnen  
12  
13 plot(t, y)  
14 show()
```

In der ersten Zeile werden die Funktionen der Matplotlib-Bibliothek importiert. Damit mathematische Funktionen verwendet werden können, muss die math Bibliothek importiert werden. Alle Objekte und Funktionen für Arrays werden in der dritten Zeile importiert. Um diese zu verwenden, muss der *Prefix* np. vor die entsprechenden Funktionen gestellt werden. In Zeile 5 und 6 definieren wir die mathematische Funktion $f(t) = t^2 \exp(-t^2)$. Mit Hilfe der Funktion np.linspace() werden 51 Koordinaten dem Array t zugeordnet (Zeile 8). Es werden für alle 51 t-Koordinaten y-Funktionswerte berechnet (Zeile 7 und 8). Zuvor muss in Zeile 9 das entsprechende Array y initialisiert werden. In Zeile 11 wird der Graph der Funktion $f(t)$ erstellt und anschliessend mit dem Befehl show() in der letzten Zeile angezeigt. Die Ausgabe zu diesem Programm sieht wie folgt aus: Alle erzeugten Graphen können in verschiedenen

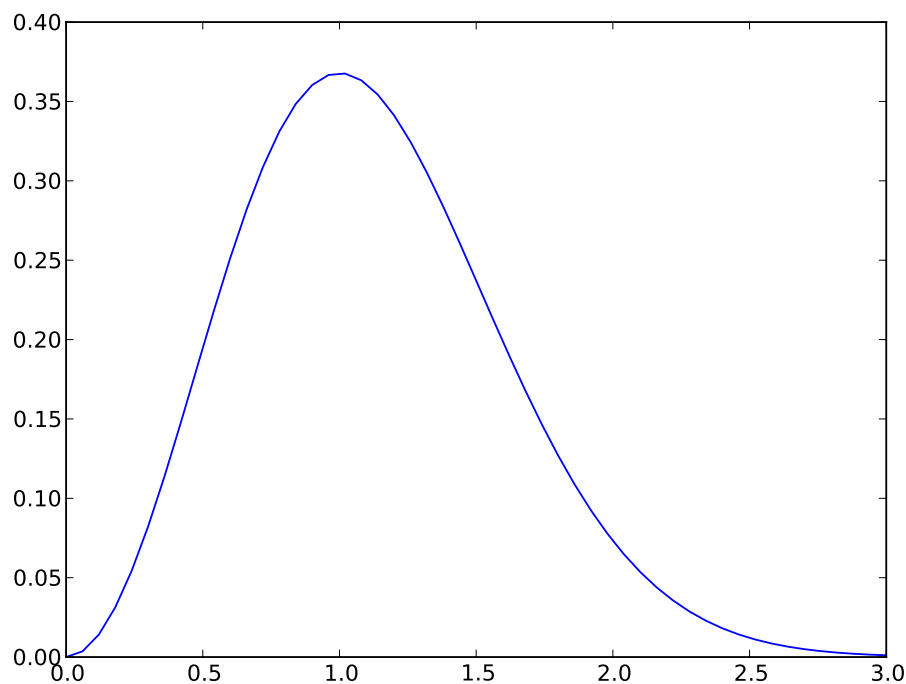


Abbildung 4.1: Graph der Funktion $f(t) = t^2 \exp(-t^2)$

Grafikformaten, z. B. als PNG oder PDF, mit dem Befehl savefig() abgespeichert werden:

```
savefig('tmp2.png')  
savefig('tmp2.pdf')
```

Die Berechnung der Funktionswerte mit Hilfe einer Schleife ist umständlich. Die Bibliothek `numpy` ermöglicht die Berechnung eines ganzen Arrays in einem Schritt. Im folgenden Beispiel werden für alle Werte des Arrays `t` (`[0, 0.5, 1, 1.5, 2]`) die Quadratzahlen berechnet und in einem Array `y` abgespeichert. In der vierten Zeile wird auf alle Elemente des Arrays `t` die Funktion $\sin(t)$ angewendet und in einem neuen Array `y2` abgespeichert.

```
1 import numpy as np
2 t = np.linspace(0, 2, 5)  #[0, 0.5, 1, 1.5, 2]
3 y = t*t
4 y2 = np.sin(t)
```

Es ist wichtig, dass alle mathematischen Funktionen aus der `numpy`-Bibliothek verwendet werden (z.B. `np.exp()`, `np.sin()`, `np.cos()`, `np.sqrt()`, usw.). Nur Funktionen der `numpy`-Bibliothek können mit Arrays rechnen. Funktionen der `math`-Bibliothek können nur einzelne Zahlen verarbeiten.

Im folgenden Beispiel werden zwei Funktionen für ganze Arrays definiert und in einem Graphen dargestellt:

```
1 from matplotlib.pyplot import *
2 import numpy as np
3
4 def f1(t):
5     return t**2*np.exp(-t**2)  #Vektor Funktion exp() verwenden
6 def f2(t):
7     return t**2*f1(t)
8
9 t = np.linspace(0, 3, 51)
10 y1 = f1(t)
11 y2 = f2(t)
12
13 plot(t, y1, 'r-')
14 plot(t, y2, 'bo')
15 xlabel('t')
16 ylabel('y')
17 legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
18 title('Zwei Kurven in einer Abbildung')
19 show()
```

Wir betrachten nun das obige Programm Zeile für Zeile. In den ersten zwei Zeilen werden Python Objekte der `matplotlib`- und der `numpy`-Bibliothek importiert. In Zeile 4 und 5 wird die Funktion $f_1(t) = t^2 * \exp(-t^2)$ definiert. Die Funktion `f2()` quadriert t und multipliziert das Ergebnis mit `f1(t)`. Dies entspricht mathematisch $f_2(t) = t^4 * \exp(-t^2)$. In Zeile 9 werden 51 gleichmässig verteilte Koordinaten zwischen 0 und 3 dem Vektor (Array) `t` zugeordnet. In Zeile 10 und 11 werden die Funktionswerte für alle Elemente in `t` berechnet und den beiden Arrays `y1` und `y2` zugeordnet.

In den Zeilen 13 - 18 wird das Aussehen des Graphen definiert. `plot(t, y1, 'r-')` bewirkt, dass die erste Kurve $y_1(t)$ als rote Linie gezeichnet wird. Mit `plot(t, y2, 'bo')` werden für $y_2(t)$ blaue Kreise gezeichnet. Auf der Webseite <http://matplotlib.sourceforge.net/> finden Sie eine ausführliche Dokumentation mit vielen Beispielen zu verschiedenen Darstellungsmöglichkeiten. In Zeile 15 - 18 wird der Graph entsprechend beschriftet. Die Abbildung 4.2 zeigt den mit obigem Programm erzeugten Graphen.

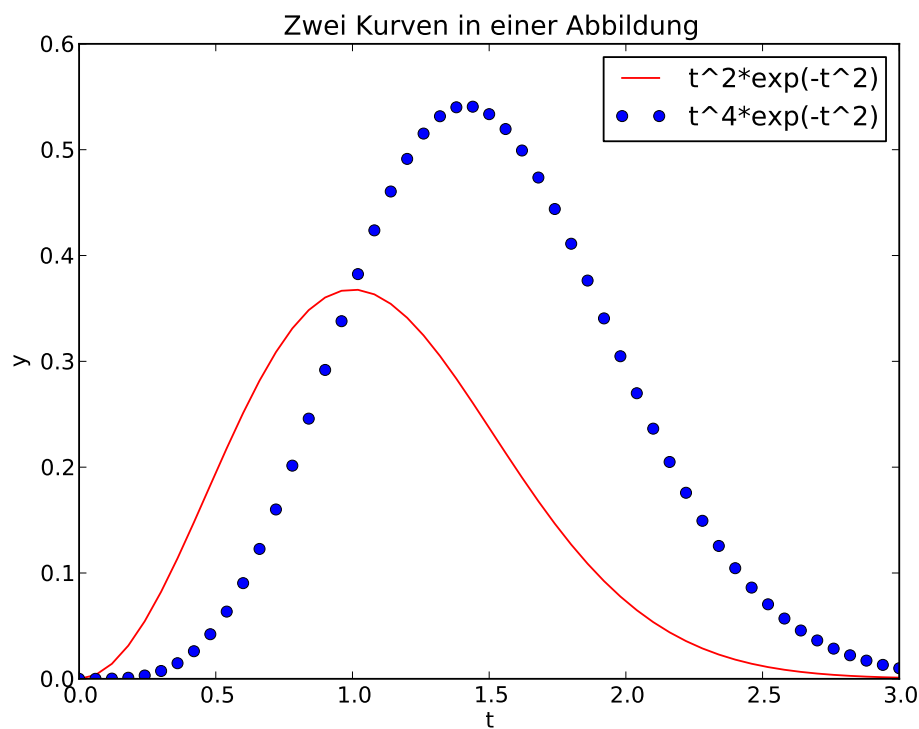


Abbildung 4.2: Zwei Kurven zu den Funktionen $f_1(t) = t^2 * \exp(-t^2)$, $f_2(t) = t^4 * \exp(-t^2)$

Im folgenden Beispiel sollen nun zwei Grafiken übereinander mit jeweils zwei Kurven dargestellt werden:

```

1 from matplotlib.pyplot import *
2 import numpy as np
3
4 def f1(t):
5     return t**2*np.exp(-t**2)
6 def f2(t):
7     return t**2*f1(t)
8
9 figure() # Zwei Plots
10 t = np.linspace(0, 3, 51)
11 y1 = f1(t)
12 y2 = f2(t)
13
14 subplot(2, 1, 1)
15 plot(t, y1, 'r-', t, y2, 'bo')
16 ylabel('y')
17 axis([t[0], t[-1], min(y2)-0.05, max(y2)+0.5])
18 legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
19 title('Obere Abbildung')
20
21 subplot(2, 1, 2)
22 t3 = t[::4]

```

```

23 y3 = f2(t3)
24
25 plot(t, y1, 'm--', t3, y3, 'kD')
26 xlabel('t')
27 ylabel('y')
28 axis([0, 4, -0.2, 0.6])
29 legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
30 title('Untere Abbildung')
31
32 show()

```

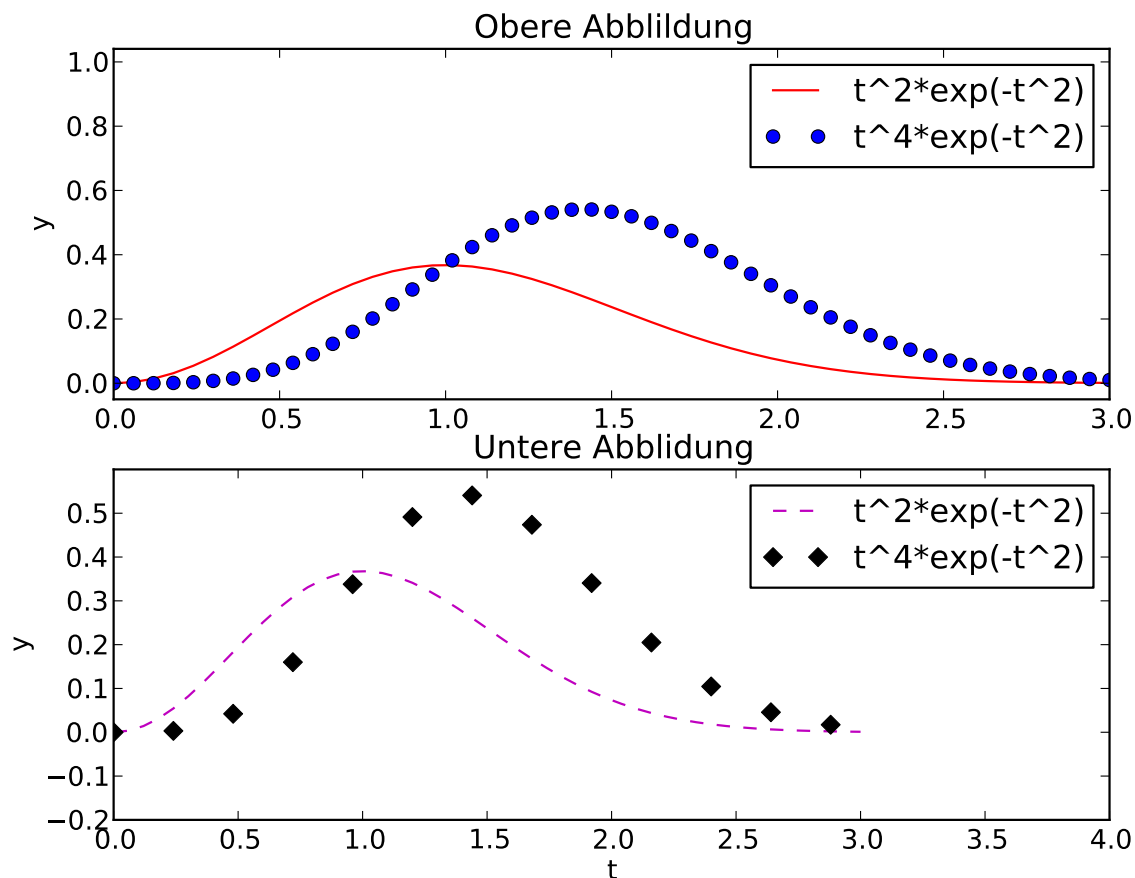


Abbildung 4.3: Es werden zwei Grafiken mit jeweils den gleichen Funktionen $f_1()$, $f_2()$ in einer Abbildung übereinander dargestellt. Im unteren Graphen wird nur jeder vierte Funktionswert dargestellt.

Die Zeilen 1 - 12 sind identisch mit dem vorangegangenen Beispiel. Die Funktion `subplot(2, 1, 1)` definiert, dass zwei Graphen untereinander dargestellt werden. Der erste Parameter gibt die Anzahl Graphen bezüglich vertikaler Richtung an. Der zweite beinhaltet die Anzahl Graphen bezüglich horizontaler Richtung. Der dritte Parameter definiert die Nummer des aktuellen Graphs. Alle Formatierungsanweisungen ab Zeile 14 beziehen sich auf den aktuellen (ersten) Graphen.

Der `plot()`-Funktion in Zeile 15 werden 6 Argumente übergeben und damit gleich zwei Funktionen gezeichnet. Das erste und das vierte Argument übergeben die t -Koordinaten, das

zweite und das dritte Argument übergeben die Funktionswerte (y-Koordinaten). Der dritte und der sechste Parameter legen die Darstellung der Kurve fest. In Zeile 16 - 19 wird der erste Graph beschriftet (Label, Legende, Titel). Nach dem Aufruf der Funktion `subplot(2, 1, 2)` (Zeile 21) bezieht sich nun alles auf den zweiten Graphen (vertikal 2 Graphen, horizontal 1 Graph). In Zeile 22 wird jeder vierte Wert der t-Koordinaten (des Zeitvektors) `t3 = t[::4]` einem neuen Vektor `t3` zugeordnet. In der folgenden Zeile werden die Funktionswerte zu diesem Vektor (`t3`) berechnet und im Vektor `y3` abgespeichert. In Zeile 25 werden die Kurven $y(t)$ und $y3(t3)$ gezeichnet. Die folgenden Zeilen beschriften den zweiten Graphen entsprechend. Obwohl es sich um die gleichen Funktionen handelt, erscheinen die Kurven in der unteren Figur steiler und höher, da in Zeile 28 ein neuer Darstellungsbereich definiert wird. Die Abbildung 4.3 zeigt die Ausgabe des Programms.

4.3 Animierte Kurve

Die Variation eines Parameters kann durch eine Sequenz von Kurven visualisiert werden. In einem ersten Schritt werden die einzelnen Graphen als Bilddatei abgespeichert. Danach werden sie automatisiert zu einer Animation zusammengefügt. Die Ausgabe des folgenden Programms finden Sie unter Youtube <http://youtu.be/ITnM1-rUZfU>

$$f(x; m, s) = (2\pi)^{-1/2} s^{-1} \exp \left[-\frac{1}{2} \left(\frac{x-m}{s} \right)^2 \right] \quad (4.1)$$

Die Gaussfunktion (Gleichung 4.1) wird in vielen Bereichen verwendet. Es ist es eine Funktion einer unabhängigen Variablen x und zweier Parameter m, s . Im folgenden Beispiel wird der Parameter $m = 0$ gewählt, der Parameter s wird zwischen 0.2 und 2 variiert.

```

1 from matplotlib.pyplot import *
2 from math import *
3 import numpy as np
4
5 def f(x, m, s):
6     return (1.0/(np.sqrt(2*pi)*s))*np.exp(-0.5*((x-m)/s)**2)
7 m=0
8 s_start = 2; s_stop = 0.2
9 s_values = np.linspace(s_start, s_stop, 20)
10 x = np.linspace(m-3*s_start, m+3*s_start, 1000)
11
12 max_f = f(m, m, s_stop)
13
14 counter = 0
15 for s in s_values:
16     figure()
17     y = f(x, m, s)
18     plot(x,y)
19     axis([x[0], x[-1], -0.1, max_f])
20     xlabel('x'); ylabel('f')
21     legend(['s=%4.2f' % s])
22     savefig('tmp%04d.pdf' % counter)
23     counter += 1

```

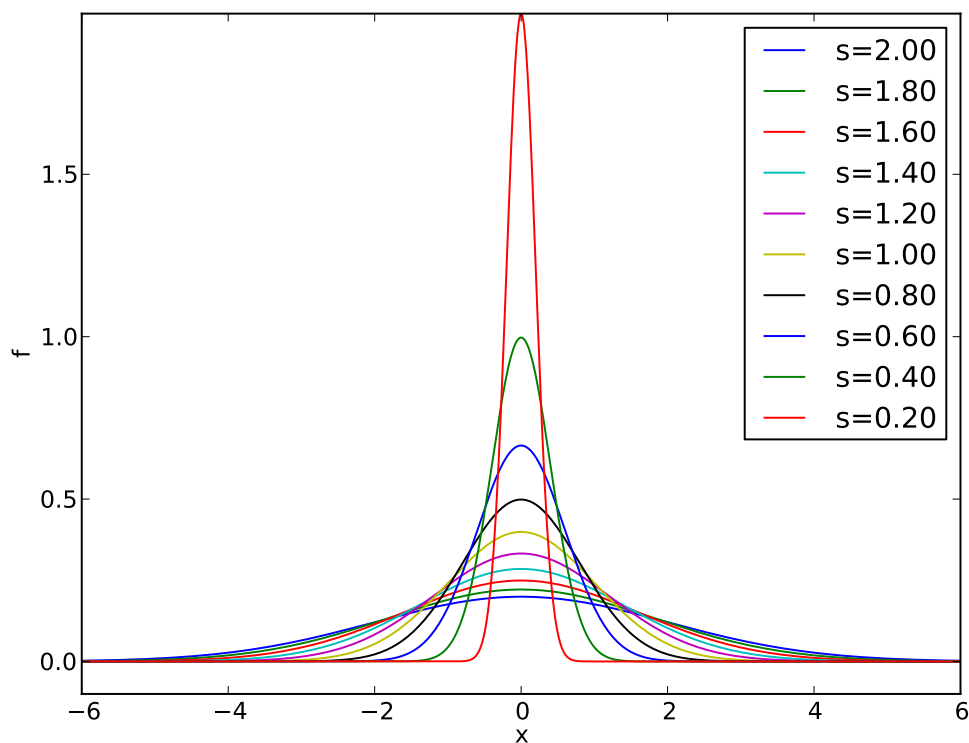


Abbildung 4.4: Kurvenschar der Gaussfunktion mit Variation des Parameters s von 2 bis 0.2

In den Zeilen 1-3 werden die benötigten Bibliotheken importiert. In Zeile 5 und 6 wird die Gaussfunktion definiert. In Zeile 7 - 10 werden die Parameter der Kurvenschar definiert. In Zeile 13 wird der maximale Funktionswert berechnet. Die Variable counter (Zeile 14) nummeriert die Dateinamen. In der Schleife (ab Zeile 16) wird für jeden s -Wert aus dem Array `s_values` ein Graph erzeugt und ein Bild davon abgespeichert. Der Aufruf `figure()` erstellt einen neuen Graphen (falls mehrere Kurven in eine Abbildung gezeichnet werden sollen, kann `figure()` ausserhalb der Schleife stehen). In Zeile 18 werden die Funktionswerte für die aktuellen Parameter berechnet und dann in der nächsten Zeile mit `plot(x,y)` gezeichnet. Der Darstellungsbereich wird in Zeile 20 fest definiert. In Zeile 23 wird der aktuelle Wert von s in die Legende geschrieben. Am Ende, in Zeile 24, werden die einzelnen Graphen nummeriert abgespeichert. Mit Hilfe von open source software kann aus der Bildsequenz eine Animation erstellt werden. Eine mögliche Software ist Firefogg, ein *Addon* für Firefox <http://firefogg.org/>.

4.4 Funktionen mit höherem Schwierigkeitsgrad

In diesem Abschnitt sollen spezielle Klassen von Funktionen visualisiert werden. Die stückweise definierten Funktionen sind tückisch (bei ihrer Definition) in der Programmiersprache Python. Schnell variierende Funktionen sind schwierig graphisch darzustellen.

4.4.1 Stückweise definierte Funktionen

Die *Heaviside*-Stufen-Funktion kann mathematisch wie folgt definiert werden:

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases} \quad (4.2)$$

In einem ersten Ansatz liefert die folgende Funktion korrekte Funktionswerte für jeweils eine x-Koordinate.

```
1 def H(x):
2     return (0 if x < 0 else 1)
```

Die Verwendung dieser Python-Funktion hätte zur Folge, dass zu jedem Element eines x-Koordinaten Arrays die entsprechenden y-Koordinaten mit Hilfe einer Schleife berechnet werden müssten. Dies kann durch den folgenden Programmabschnitt bewerkstelligt werden:

```
1 def H_loop(x):
2     r = np.zeros(len(x))
3     for i in xrange(len(x)):
4         r[i] = H(x[i])
5     return r
```

Da wir uns in diesem Kapitel an das kompakte Rechnen mit Arrays gewöhnt haben, wird im Folgenden ein Beispiel für eine vektorielle *Heaviside*-Stufen-Funktion mit Hilfe einer neuen numpy-Funktion gezeigt:

```
1 from matplotlib.pyplot import *
2 import numpy as np
3
4 def Hv(x):
5     return np.where(x < 0, 0.0, 1.0)
6
7 x = np.linspace(-10, 10, 9)
8 x2 = np.linspace(-10, 10, 50)
9 plot(x, Hv(x), 'r')
10 plot(x2, Hv(x2), 'b')
11 legend(['5 points', '50 points'])
12 axis([x[0], x[-1], -0.1, 1.1])
13 show()
```

Die Vektor-Funktion `np.where(x < 0, 0.0, 1.0)` liefert zu jedem Element des Vektors x den Wert 0.0 oder 1.0, je nachdem, ob die Bedingung $x < 0$ erfüllt ist oder nicht. Die *Heaviside*-Stufen-Funktion besteht aus zwei waagrechten Linien, jeweils beim y-Wert $y = 0$ und $y = 1$. Die rote Kurve wird mit 5 Punkten, die blaue Kurve mit 50 Punkten gezeichnet (Abbildung 4.5). Auch mit 50 Punkten erscheint die Stufe an der Stelle $x = 0$ nicht vollkommen vertikal.

Eine sogenannte *Hut-Funktion* kann mathematisch wie folgt definiert werden:

$$N(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \leq x < 1 \\ 2 - x, & 1 \leq x < 2 \\ 0, & x \geq 2 \end{cases} \quad (4.3)$$

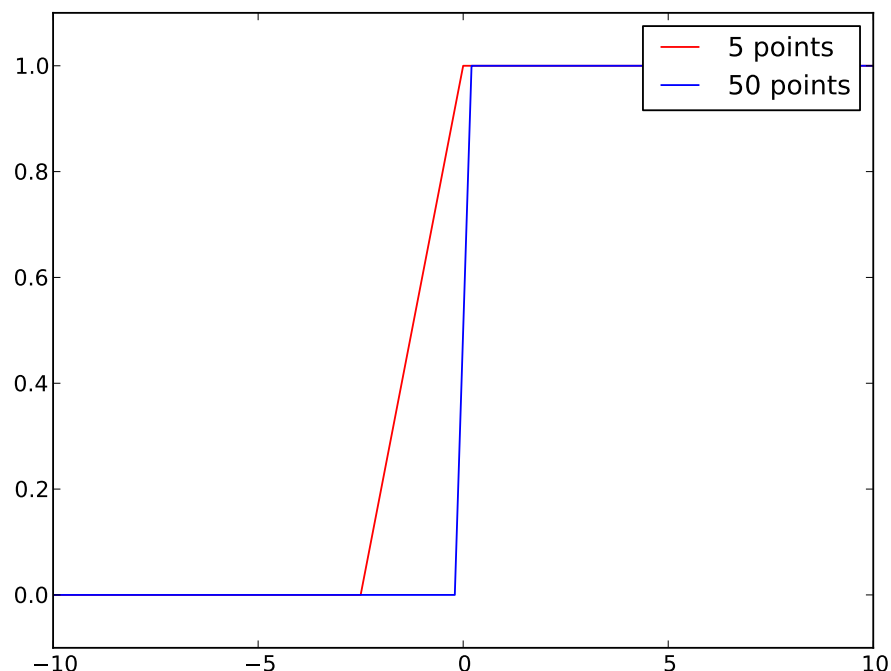


Abbildung 4.5: Heaviside-Stufen-Funktion mit 5 und 50 Punkten

Schwierigkeiten bei der Implementierung in der Programmiersprache Python bereitet die zweite Bedingung der *Hut-Funktion*. Sie ist für das zweite Stück zwischen 0 und 1 definiert ($0 \leq x < 1$). Die Definition dieses Bereichs beinhaltet eine logische Verknüpfung der beiden Bedingungen $x \geq 0$ and $x < 1$. Damit jedoch eine logische *and*-Verknüpfung auf einen ganzen Vektor angewendet werden kann, muss ein entsprechender Operator `condition = operator.and_(0 <= x, x < 1)` aus der Python-Bibliothek eingesetzt werden. Das folgende Programm definiert eine vektorielle *Hut-Funktion*:

```
from matplotlib.pyplot import *
import numpy as np
import operator

def Hv(x):
    r = np.where(x < 0, 0.0, x)
    condition = operator.and_(0 <= x, x < 1)
    r = np.where(condition, x, r)
    condition = operator.and_(1 <= x, x < 2)
    r = np.where(condition, 2-x, r)
    r = np.where(x >= 2, 0.0, r)
    return r

x = np.linspace(-1, 3, 10)
x2 = np.linspace(-1, 3, 200)
plot(x, Hv(x), 'ro')
```

```

plot(x2, Nv(x2), 'b')
legend(['10 points', '200 points'])
axis([x[0], x[-1], -0.1, 1.1])
show()

```

Im obigen Beispiel werden für 10 x-Koordinaten die entsprechenden Funktionswerte berechnet. Bei Funktionen mit sprunghaftem Verhalten muss darauf geachtet werden, dass genügend Punkte berechnet werden, wenn sie als verbundene Kurve dargestellt werden sollen.

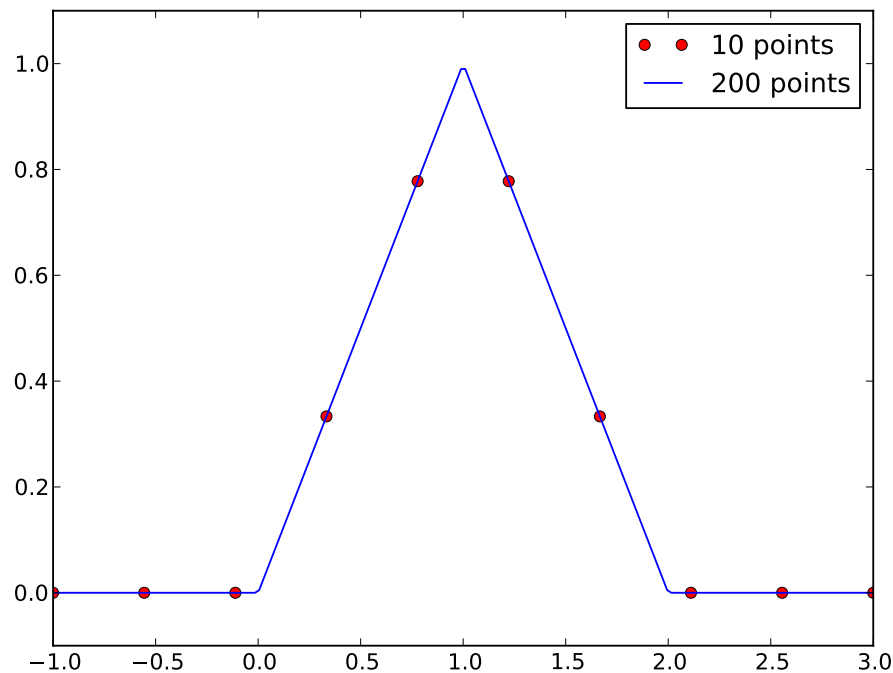


Abbildung 4.6: Hut-Funktion mit 10 und 200 Punkten

4.4.2 Schnell variierende Funktionen

Die Funktion $f(x) = \sin(1/x)$ ist an der Stelle $x = 0$ stetig. In der Nähe von $x = 0$ variiert die Funktion sehr stark. Mit dem folgenden Programm werden zwei Graphen der gleichen Funktion gezeichnet, der erste mit 10 Punkten, der zweite mit 1000 Punkten.

```

1 from matplotlib.pyplot import *
2 import numpy as np
3
4 def f(x):
5     return np.sin(1.0/x)
6
7 x1 = np.linspace(-0.5, 0.5, 10)
8 x2 = np.linspace(-0.5, 0.5, 1000)
9 #figure()
10 subplot(211)
11 plot(x1, f(x1))

```

```

12 legend(['%d points' % len(x1)])
13
14 subplot(212)
15 plot(x2, f(x2))
16 legend(['%d points' % len(x2)])
17 show()

```

In Zeile 4 ist die Funktion $f(x) = \sin(1/x)$ vektoriell definiert (d.h. x ist ein Vektor mit n x -Koordinaten). Der Vektor $x1$ beinhaltet 10, der Vektor $x2$ beinhaltet 1000 gleichmässig verteilte x -Koordinaten (Zeile 7 und 8). In Zeile 10 beginnt durch die Anweisung `subplot(211)` der obere Graph mit 10 Punkten. Ab Zeile 14 wird mit `subplot(212)` der untere Graph mit 1000 Punkten definiert. Als einfacher Test, um festzustellen, ob ein Graph mit einer genügenden Anzahl Punkten dargestellt wird, kann die Anzahl Punkte verdoppelt werden. Falls sich das Aussehen nicht stark verändert, ist die Anzahl Punkte adäquat gewählt.

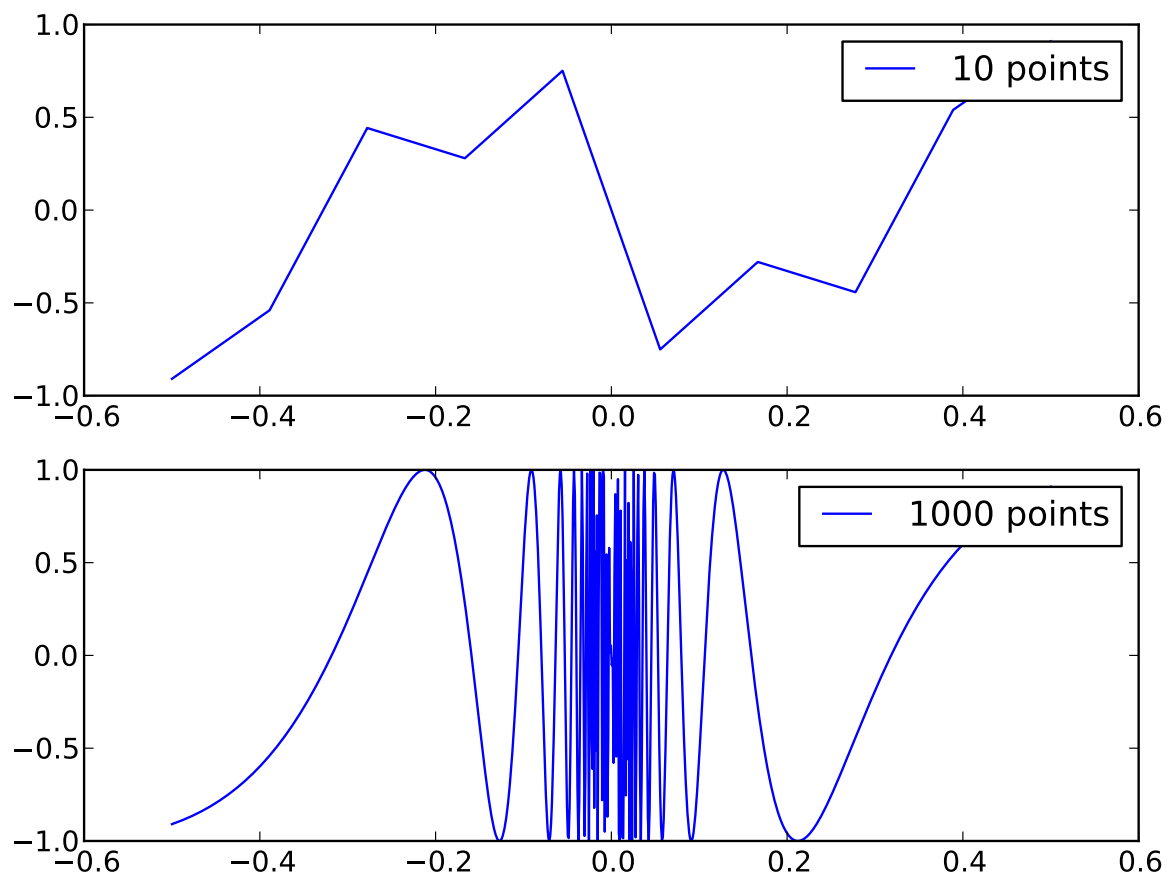


Abbildung 4.7: *schnell $\sin 1/x$*

4.5 Zusammenfassung Kapitel 4

Rechen mit Arrays. Falls eine Funktion $f(x)$ auf einen Vektor (Array) x angewendet werden soll, müssen alle benutzten Funktion mit Vektoren umgehen können. Die Bibliothek `numpy` liefert

diese benötigten Funktionen. Die *if* Anweisung kann nicht auf ein ganzes Array angewendet werden. In den meisten Fällen hilft die Funktion `np.where()` weiter.

Kurven darstellen. Mit Hilfe der Bibliothek `matplotlib` können mathematische Funktionen als Graphen professionell dargestellt werden. In diesem Kapitel wurden zahlreiche Beispiele gezeigt.

Tabelle 4.1: Zusammenfassung der wichtigsten Funktionen der Bibliothek Numerical Python (*numpy*)

Anweisung	Bedeutung
<code>array(ld)</code>	Erzeugt ein Array aus allen Elementen der Liste <code>ld</code>
<code>zeros(n)</code>	Erzeugt ein Array vom Typ <code>Float</code> mit <code>n</code> Elementen mit Wert <code>0.0</code>
<code>zeros(n, int)</code>	Erzeugt ein Array vom Typ <code>Integer</code> mit <code>n</code> Elementen mit Wert <code>0</code>
<code>zeros((m,n))</code>	Erzeugt ein 2D Array mit den Dimensionen <code>m x n</code>
<code>linspace(a,b,m)</code>	gleichförmig verteilte Sequenz mit <code>m</code> Werten zwischen <code>a</code> und <code>b</code>
<code>len(a)</code>	Anzahl Elemente von <code>a</code>
<code>a[i]</code>	Element <code>i</code> des Arrays <code>a</code>
<code>a[i,j]</code>	Zugriff auf 2d Array <code>a</code>
<code>a[1:k]</code>	Unterteilt das Array <code>a</code> , liefert die Elemente 1 bis <code>k-1</code>
<code>a[1:8:3]</code>	Unterteilt das Array <code>a</code> und liefert jedes dritte Element, mit <code>i=1,4,7</code>
<code>b = a.copy()</code>	Macht eine Kopie des Arrays
<code>sin(a), exp(a), ...</code>	math. Funktionen anwendbar auf Arrays
<code>c = concatenate(a, b)</code>	verbindet das Array <code>a</code> mit <code>b</code>
<code>c = where(cond, a1, a2)</code>	<code>c[i] = a1[i]</code> if <code>cond[i]</code> , sonst <code>c[i] = a2[i]</code>
<code>isinstance(a, ndarray)</code>	ist <code>True</code> , falls <code>a</code> ein Array ist

4.6 Abschliessendes Beispiel

Als abschliessendes Beispiel soll untersucht werden, wie sich die Temperatur im innern der Erde ändert. Die Temperatur an der Oberfläche oszilliert zwischen einem maximalen Wert am Nachmittag und einem minimalen Wert am frühen Morgen. Ein Frage könnte sein, wie ändert sich die Temperatur 10 Meter unter dem Boden, wenn die Oberflächentemperatur zwischen 2°C und 15°C variiert.

Wir wählen ein Koordinatensystem, in welchem die z -Achse in Richtung Erdmittelpunkt zeigt und $z = 0$ auf der Erdoberfläche liegt. Die Temperatur an einer Stelle z unter der Erde zu einem Zeitpunkt t kann beschrieben werden mit $T(z, t)$. Die Oberflächentemperatur soll periodisch variieren:

$$T(0, t) = T_0 + A \cos(\omega t) \quad (4.4)$$

Mit dem mathematischen Modell für die Wärmeleitung kann die Temperatur im Erdinnern mit der folgenden Formel beschrieben werden:

$$T(z, t) = T_0 + Ae^{az} \cos(\omega t - az), \text{ mit } a = \sqrt{\left(\frac{\omega}{2k}\right)} \quad (4.5)$$

Der Parameter k ist die thermische Leitfähigkeit des Bodens. Die Aufgabe besteht nun darin, das Temperaturprofil in Abhängigkeit der Zeit als Animation darzustellen. ω wird so gewählt, dass die Periodendauer 24h entspricht. Die Durchschnittstemperatur T_0 wird auf 10°C gesetzt.

Die Amplitude der Temperaturänderung soll 10°C betragen. Die thermische Leitfähigkeit des Bodens k wird auf $1\text{mm}^2/\text{s}$ festgelegt (das entspricht $10^{-6}\text{m}^2/\text{s}$ in SI-Einheiten).

Um die Animation $T(z, t)$ zu berechnen, wird eine Schleife über verschiedene Zeitpunkte benötigt, zu jedem dieser Zeitpunkte soll ein Graph des Temperaturprofils erstellt und als Bild gespeichert werden. Am Ende soll daraus eine Animation erstellt werden.

Im Programm wird eine verallgemeinerte Funktion `animate()` verwendet. Diese erlaubt es Animationen für beliebige Funktionen $f(z, t)$ zu erstellen. Der darzustellende y-Bereich soll zu jedem Zeitschritt gleich gewählt sein, deshalb werden der `animate()`-Funktion die beiden Argumente `ymin` und `ymax` übergeben.

Bei jeder Simulation ist die richtige Zuordnung der physikalischen Parameter entscheidend. Das Programm verwendet die Parameter `n`, `D`, `T0`, `A`, `omega`, `dt`, `tmax`, und `k`. Die Periodendauer P der Oszillation beträgt 24h. Das ergibt für $\omega = 2\pi/P$. Da wir mit Sekunden rechnen, erhalten wir für $P = 24\text{h} = 24 \times 60 \times 60\text{ s}$. Es sollen 3 Tage simuliert werden $t_{\text{max}} = 3P$.

```
k = 1E-6
P = 24*60*60
dt=P/24
tmax = 3*P
T0 = 10
A = 10
a = sqrt(omega/(2*k))
D = -(1/a)*log(0.001)
n = 501
```

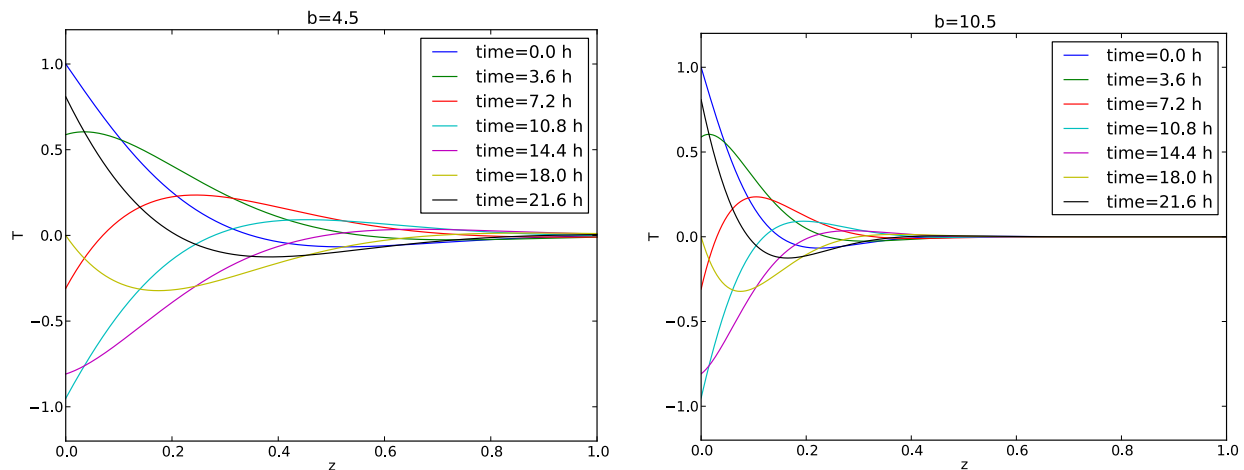


Abbildung 4.8: Kurvenschar eines Temperaturprofils während 12h für $b = 4.5$ und $b = 10.5$

Mit Hilfe einer Transformation der Variablen kann die Anzahl der physikalischen Parameter auf einen wesentlichen Parameter b reduziert werden. Wir erhalten folgende Funktionsgleichung:

$$T(z, t) = e^{-bz} \cos(t - bz), \text{ mit } b = D\sqrt{\omega/(2k)} (\approx 6.9) \quad (4.6)$$

Der Parameter b bestimmt, wie tief die Temperaturveränderung in den Boden eindringt. Ein kleines b lässt die Temperatur tiefer eindringen. In der Abbildung 4.8 werden Temperaturprofile zu unterschiedlichen Zeiten für $b = 4.5$ und $b = 10.5$ dargestellt. Das folgende Programm erstellt ein Sequenz von Graphen mit Temperaturprofilen zu unterschiedlichen Zeiten. Die Animation kann unter http://www.youtube.com/watch?v=HE_d1RJrDf0 betrachtet werden.


```

1 from matplotlib.pyplot import *
2 from math import *
3 import numpy as np
4
5 def animate(tmax, dt, x, function, ymin, ymax, t0=0,\
6             xlab='x', ylab='y', hardcopy_stem='tmp_'):
7     t = t0
8     counter = 0
9     while t <= tmax:
10        figure()
11        y = function(x, t)
12        plot(x,y)
13        axis([x[0], x[-1], ymin, ymax])
14        title('time=%.1f s' % t)
15        ylabel(ylab)
16        xlabel(xlab)
17        savefig(hardcopy_stem + '%04d.pdf' % counter)
18        counter = counter + 1
19        t += dt
20
21 def T(z, t):
22     return np.exp(-b*z)*np.cos(t - b*z) # b is global
23
24 b = float(2)
25 n = 401
26 z = np.linspace(0, 1, n)
27 animate(3*2*pi, 0.05*2*pi, z, T, -1.2, 1.2, 0, 'z', 'T')

```

4.7 Übungen zu Kapitel 4

Übung 4.1: Graph einer Funktion

Stellen Sie die Funktion $y(t) = v_0 t - 0.5gt^2$ für $v_0 = 10$, $g = 9.81$ und $t \in [0, 2v_0/g]$ graphisch als Diagramm dar. Die Anschrift der x-Achse soll 'Zeit (s)' sein, die Anschrift der y-Achse soll 'Höhe (m)' sein. Nennen Sie das Programm *plot_ball.py*.

Übung 4.2: Kurvenschar

Für alle Werte v_0 der folgenden Liste $v_0 = [0.2, 0.5, 1, 2, 5, 10, 20, 40]$ sollen alle Kurven der Funktion $y(t) = v_0 t - 0.5gt^2$ mit $g = 9.81$ dargestellt werden. Sei $t \in [0, 2v_0/g]$ der Zeitbereich für jede Kurve. Es ist zu beachten, dass jede Kurve einen eigenen t-Vektor benötigt. Nennen Sie das Programm *plot_ball2.py*.

Übung 4.3: Flugbahn eines Balls (schiefer Wurf)

Die Funktionsgleichung (1.10) der Trajektorie eines Balls wurde bereits in Kapitel 1 vorgestellt.

$$f(x) = x \tan(\Theta) - \frac{1}{2v_0^2} \frac{gx^2}{\cos^2(\Theta)} + y_0$$

Verändern Sie jeweils einen Parameter (y_0, v_0, Θ) und stellen Sie die so erhaltene Kurvenschar in einem Graphen dar. Wählen Sie die Parameter für die drei Fälle wie folgt: Fall 1: $y_0 = 0, v_0 =$

10 $\theta = 1.0/18 \cdot \pi \cdot [1, 2, 3, 4, 5, 6, 7, 8, 9]$, Fall 2: $y_0 = 0, \Theta = \pi/6, v_0 = [2, 5, 10, 20, 30, 50]$ und Fall 3: $v_0 = 10, \Theta = \pi/6, y_0 = [0, 2, 5, 10]$. Nennen Sie das Programm *plot_schiefer_wurf.py*.

Übung 4.4: Gedämpfte Schwingung

$$f(t; A, a, \omega) = A e^{-at} \sin(\omega t) \quad (4.7)$$

Stellen Sie die obige Funktion für verschiedene Parameter dar und beschreiben Sie in Worten, was die Parameter bewirken.

```
1 def f(t, A=1, a=1, omega=2*pi):
2     return A*exp(-a*t)*sin(omega*t)
```

Nennen Sie ihr Programm *gedämpfte_schwingung.py*

Übung 4.5: Wellenpaket

Die folgenden Funktionsgleichung definiert ein Wellenpaket.

$$f(x, t) = e^{-(x-3t)^2} \sin(3\pi(x-t)) \quad (4.8)$$

Stellen Sie diese Funktion in einem Bereich von $x = -4$ bis $x = 4$ zu den Zeiten $t = [0, 0.5, 1, 2, 4]$ graphisch dar. Nennen Sie das Programm *wellenpaket.py*.

Übung 4.6: Wieviele x-y-Punkte müssen verwendet werden?

```
1 import numpy as np
2 x = np.linspace(0, 2, 20)
3 y = x*(2 - x)
4 import matplotlib.pyplot as plt
5 plt.plot(x, y)
6 plt.show()
```

Führen Sie das obige Programm aus und betrachten Sie die graphische Ausgabe. Als Nächstes ersetzen Sie die Funktion in Zeile 3 durch die folgende trigonometrische Funktion:

$y = \cos(18 \cdot \pi \cdot x)$. Was beobachten Sie? Stellen Sie diese Funktion nochmals mit 1000 Punkten (Zeile 2) dar.

Übung 4.7: Viskosität von Wasser

Die Viskosität μ von Wasser ändert sich mit der Temperatur T (in Kelvin) entsprechend der Formel:

$$\mu(T) = A \cdot 10^{B/(T-C)} \quad (4.9)$$

Die Konstante A hat den Wert $A = 2.414 \cdot 10^{-5} \text{ Pa s}$, die Konstante B hat den Wert $B = 247.8 \text{ K}$ und C hat den Wert $C = 140 \text{ K}$. Stellen Sie diese Funktion graphisch für einen Bereich von 0°C bis 100°C dar. Beschriften Sie die x-Achse mit 'Temperatur (C)' und die y-Achse mit 'Viskosität (Pa s)'. Nennen Sie ihr Programm *visko.py*.

Übung 4.8: Taylor Approximation

Jede periodische Funktion kann mit einer Taylor Summe approximiert werden. Die folgende Summe ist eine Näherung für $\sin(x)$:

$$\sin(x) \approx S(x; n) = \sum_{j=0}^n (-1)^j \frac{x^{2j+1}}{(2j+1)!} \quad (4.10)$$

Je mehr Glieder dieser Summe berechnet werden, desto genauer wird die Näherung. Im Fall des Grenzübergangs gilt: $\lim_{n \rightarrow \infty} S(x; n) = \sin(x)$. Es soll im Folgenden die Qualität der Näherung $S(x; n)$ visualisiert werden. Als Erstes wird eine Python-Funktion benötigt, welche $S(x; n)$ berechnet. Als Zweites sollen die folgenden Graphen $\sin(x)$, $S(x; 1)$, $S(x; 2)$, $S(x; 3)$, $S(x; 6)$ und $S(x; 12)$ für den Bereich von $x = 0$ bis $x = 4\pi$ dargestellt werden. Nennen Sie die Funktion `plot_taylor.py`

Übung 4.9: Animierte Heaviside Funktion

Die *Heaviside*-Stufen-Funktion wird in vielen mathematischen Modellen verwendet. Häufig ist jedoch die Unstetigkeit der Funktion ein Problem. Um diese Problematik zu vermeiden, kann eine geglättete *Heaviside*-Stufen-Funktion wie folgt definiert werden:

$$H_\epsilon(x) = \begin{cases} 0, & x < -\epsilon \\ \frac{1}{2} + \frac{x}{2\epsilon} + \frac{1}{2\pi} \sin\left(\frac{\pi x}{\epsilon}\right), & -\epsilon \leq x \leq \epsilon \\ 1, & x > \epsilon \end{cases} \quad (4.11)$$

Erstellen Sie eine Animation dieser Funktion, indem Sie den Parameter ϵ von 0.5 bis 0.0001 gleichmässig variieren. Nennen Sie ihr Programm `heaviside_animation.py`.

Übung 4.10: Näherung von Pi animieren

Das folgende Programm zeichnet ein regelmässiges 12-Eck mit dem Radius 0.5:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 n=12
4 R=0.5
5 t=np.array(range(n+1))
6 x=R*np.cos(2*np.pi*t/n);
7 y=R*np.sin(2*np.pi*t/n);
8 plt.figure()
9 plt.gca().set_aspect('equal')
10 plt.plot(x, y)
11 plt.show()
```

Erweitern Sie das Programm so, dass eine Sequenz von Bildern entsteht, beginnend mit einem regelmässigen Dreieck und endend mit einem 73-Eck. Geben Sie in der Titelzeile zur Grafik die Abweichung des Umfangs zur Zahl π aus. Der Umfang kann mit Hilfe der Funktion `pfadlaenge()` aus der Übung 3.4 berechnet werden. Nennen Sie das Programm `pi.polygon.py`.

Übung 4.11: Physikalisches Konzept der Energie

Die Höhe $y(t)$ eines vertikal nach oben geworfenen Balls kann mit der Funktionsgleichung $y(t) = v_0 t - 0.5gt^2$ beschrieben werden ($g = 9.81 \text{ m/s}^2$ und v_0 als Anfangsgeschwindigkeit zum Zeitpunkt $t = 0$). Die Potentielle Energie ($P = mgy$) und die Kinetische Energie (Bewegungsenergie) sind wichtige physikalische Grössen. Die Kinetische Energie ist wie folgt

definiert: $K = 1/2mv^2$, dabei ist v die Momentangeschwindigkeit des Balls. Es gilt $v(t) = y'(t)$. Erstellen Sie einen Graphen von den Funktionen $P(t)$, $K(t)$ und $P(t) + K(t)$ für den Bereich von $t = 0$ bis $t = 2v_0/g$. Wählen Sie $m = 1$ und $v_0 = 10$. Nennen Sie das Programm *energie.py*

Übung 4.12: Numerische Genauigkeit

Stellen Sie die folgende Funktion $v(x)$ graphisch für $\mu = 1, 0.01, 0.001$ für den Bereich $0 \geq x \geq 1$ graphisch dar.

$$v(x) = \frac{1 - e^{x/\mu}}{1 - e^{1/\mu}} \quad (4.12)$$

Verbessern Sie die numerische Rechengenauigkeit durch die Verwendung von `numpy.float96()`.

```
import numpy
x = numpy.float96(x); mu = numpy.float96(e)
```

Nennen Sie ihr Programm *genauigkeit.py*.

Anhang

A Installationsanleitung

Ein Grund für den Erfolg von Python ist die freie Verfügbarkeit (open source) und die Unterstützung aller Plattformen (Win, OSX und Linux). Es gibt unzählige Softwarepakete, welche Python für spezifische Aufgaben erweitern. Eine Übersicht der verschiedenen Pakete liefert die Seite <http://pypi.python.org/pypi>. Die Vielzahl an unterschiedlichen Möglichkeiten einer Installation der Pakete erschweren das Leben eines Einsteigers. Aus diesem Grund empfehle ich, eine vollständige Lösung der Firma Enthought zu installieren. Diese ermöglicht eine automatisierte Installation für alle Plattformen mit einer grossen Anzahl wissenschaftlicher Pakete. Eine Anleitung dazu finden Sie unter A.1. Eine andere Möglichkeit ist, Python 2.7 und die Pakete NumPy und Matplotlib manuell zu installieren.

A.1 ENTHOUGHT Python Distribution 7.2

Die Firma Enthought (<http://www.enthought.com>) bietet eine Sammlung der wichtigsten wissenschaftlichen Python Pakete an. Für Projekte an Schulen und an Universitäten kann die Software gratis verwendet werden. Auf allen Plattformen lässt sich die Sammlung per Tastendruck installieren (http://download.enthought.com/epd_7.2/). Der Umfang beträgt je nach Plattform zwischen 200 und 300 MB. In der aktuellen Version 7.2 sind die folgenden Pakete enthalten:

Tabelle 2: *Installierte Projekte der ENTHOUGHT Python Distribution 7.2*

Project	Version	Summary
appinst	2.1.0	OS abstraction for installing application menus, links and icons
apptools	4.0.1	application building block technologies
basemap	1.0.1	plots data on map projections with matplotlib
biopython	1.58	tools for biological computation
bitarray	0.3.5	efficient representation of arrays of booleans
blockcanvas	4.0.1	visual environment for creating simulation experiments
bsdiff4	1.0.1	binary diff and patch using the BSDIFF4-format
chaco	4.1.0	interactive 2D plotting
cloud	2.3.9	access to PiCloud's cloud-computing platform
codetools	4.0.0	code analysis and execution tools
configobj	4.7.2	simple but powerful configuration file reader and writer
coverage	3.5.1	Code coverage measurement for Python
Cython	0.15.1	language for writing C extensions for Python
distribute	0.6.24	download, build, install, upgrade, and uninstall Python packages
Fortsetzung auf der nächsten Seite		

Tabelle 2 – Fortsetzung

Project	Version	Summary
docutils	0.8.1	Documentation Utilities
enable	4.1.0	low-level drawing and interaction
enstaller	4.4.1	managing and install tool for egg distributions
envisage	4.1.0	Extensible Application Framework
EPD	7.2	kitchen-sink-included Python distribution provided by Enthought
epydoc	3.0.1	generates API documentation for Python modules from docstrings
ets	4.1.0	components to construct custom scientific applications
etsdevtools	4.0.0	tools to support Python development
etsproxy	0.1.1	proxy modules for backwards compatibility
foolscap	0.6.2	new version of Twisted's native RPC protocol
freetype	2.4.4	high-quality portable font engine
fwrap	0.1.1	Tool to wrap Fortran 77/90/95 code in C, Cython and Python
GDAL	1.8.1	Geospatial Data Abstraction Library
graphcanvas	4.0.0	Interactive Graph (network) Visualization
grin	1.2.1	searches directories of source code better than grep or find
h5py	2.0.0	Python interface to the HDF library
hdf5	1.8.5.1	general purpose file format library for scientific data
html5lib	0.90	HTML parser designed to follow the WHATWG HTML5 specification
idle	2.7.2	interactive Python shell
ipython	0.12	advanced shell for interactive and exploratory computing
Jinja2	2.6	template engine
libnetcdf4	4.1.1	NetCDF (network Common Data Form) library for array-oriented data
libgdal	1.8.1	Geospatial Data Abstraction Library
libjpeg	7.0	JPEG library
libpng	1.2.40	reference library for Portable Network Graphics
libpython	1.2	mingw import library
libxml2	2.7.3	XML parser and toolkit
libxslt	1.1.24	XSLT library with XPath support
libYAML	0.1.4	YAML 1.1 parser and emitter (C library)
lxml	2.3.2	XML/XSLT library with bindings to libxml2/libxslt
matplotlib	1.1.0	interactive 2D plotting library
mayavi	4.1.0	interactive 3D visualization
MDP	3.2	Modular toolkit for Data Processing (MDP)
mingw	4.5.2	native Windows port of GCC
MKL	10.3	Intel Math Kernel Library (runtime)
netCDF4	0.9.5	interact with in both the new netCDF 4 and 3 formats
networkx	1.6	creating and manipulating graphs and networks
nose	1.1.2	extends the test loading and running features of unittest
numexpr	2.0	fast evaluation of array expressions
numpy	1.6.1	general-purpose array-processing and math library
pandas	0.6.1	data analysis library
paramiko	1.7.7.1	SSH2 protocol for secure connections to remote machines
pep8	0.6.1	Python style guide checker
PIL	1.1.7	image processing library
ply	3.4	Python implementation of lex and yacc
pyaudio	0.2.4	bindings for PortAudio

Fortsetzung auf der nächsten Seite

Tabelle 2 – Fortsetzung

Project	Version	Summary
Pycluster	1.50	clustering software for gene expression data analysis
pycrypto	2.4.1	collection of cryptographic algorithms and protocols
pydot	1.0.25	interface to Graphviz's Dot language
pyface	4.1.0	GUI abstraction layer
pyfits	3.0.3	interface to FITS formatted files
pyflakes	0.5.0	analyze Python programs and detect errors
pygarrayimage	0.0.7	allows NumPy arrays as source of texture data for pygame
pygame	1.1.4	interface for developing visually-rich applications
Pygments	1.4	code syntax highlighting package written in Python
pyhdf	0.8.3	interface to the NCSA HDF4 library
PyOpenGL	3.0.1	python binding to OpenGL and related APIs
pyOpenSSL	0.12	wrapper around the OpenSSL library
pyparsing	1.5.6	module for creating parsers with a simple grammar
pyproj	1.8.9	cartographic transformations and geodetic computations
pyserial	2.6	encapsulation the access the serial port
PySide	1.1.0	bindings for Qt
pytables	2.3.1	hierarchical datasets for extremely large data
Python	2.7.2	general-purpose, high-level programming language
pythondatetimeutil	1.5	extensions to the standard datetime module
pytz	2011n	world timezone definitions, modern and historical
pywin32	214	Python extensions for Windows
PyYAML	3.10	YAML parser and emitter
pyzmq	2.1.11	binding to zeromq (fast messaging)
Qt	4.7.3	cross-platform application and UI framework
Reportlab	2.5	generator for PDF documents from dynamic data
scikitlearn	0.9	set of python modules for machine learning and data mining
scikits.image	0.4.2	image processing routines for SciPy
scikits.statsmodels	0.3.1	statistical computations and models for use with SciPy
scikits.timeseries	0.91.3	manipulating, reporting, and plotting time series
scimath	4.0.1	support for scientific and mathematical calculations
scipy	0.10.0	libraries for mathematics, science, and engineering
scite	1.74	Text editor based on Scintilla
scons	2.0.1	Pythonic substitute for Make
SimPy	2.2	object-oriented, process-based discrete-event simulation language
Sphinx	1.1.2	creates intelligent and beautiful project documentation
SQLAlchemy	0.7.1	SQL toolkit and Object Relational Mapper
swig	1.3.40	C and C++ wrapper and interface generator
sympy	0.7.1	library for symbolic mathematics
tornado	2.1.1	non-blocking web server
traits	4.1.0	object attributes with some additional characteristics
traitsui	4.1.0	traits-capable windowing framework
Twisted	11.1.0	event-driven networking engine
VTK	5.6.0	3-D computer graphics, image processing, and visualization
wxPython	2.8.10.1	wrapper around the wxWidgets C++ GUI library
xlrd	0.7.1	extract data from Microsoft Excel (tm) spreadsheet files
xlwt	0.7.2	create spreadsheet files compatible with MS Excel

Fortsetzung auf der nächsten Seite

Tabelle 2 – Fortsetzung

Project	Version	Summary
zope.interface	3.8.0	implementation of Zope object interfaces

A.2 Installation von Python 2.7

Eine aktuelle Version von Python mit verschiedenen Konfigurationen für alle Plattformen finden Sie unter <http://python.org/getit/>.

Installation für Windows

Laden Sie Python 2.7.2 Windows Installer oder Python 2.7.2 Windows X86-64 Installer herunter. Die Datei ist etwa 10 MB gross und, verglichen mit anderen Programmiersprachen, sehr kompakt. Die Installation läuft genauso wie bei jeder anderen Windows-Software.

Achtung: Wenn Sie während der Installation gefragt werden, ob Sie optionale Komponenten abwählen möchten, so tun Sie dies nicht! Einige dieser Komponenten können für Sie sehr nützlich sein, insbesondere IDLE. Wenn Sie in der Lage sein wollen, Python von der Windows-Eingabeaufforderung aus laufen zu lassen, dann müssen Sie die Umgebungsvariable PATH entsprechend setzen. Klicken Sie hierzu auf Systemsteuerung ⇒ System ⇒ Erweitert ⇒ Umgebungsvariablen. Klicken Sie auf die Variable namens Path im unteren Bereich für 'Systemvariablen', dann wählen Sie Bearbeiten und fügen Sie ;C:\\Python27 am Ende der bereits vorhandenen Zeile an.

Installation für MAC

Laden Sie Python 2.7.2 Mac OS X 64-bit/32-bit x86-64/i386 Installer (for Mac OS X 10.6 and 10.7) oder Python 2.7.2 Mac OS X 32-bit i386/PPC Installer herunter. Installieren Sie ihre Version.

B Installation von zusätzlichen Softwarepaketen

Auf der Seite *Python Package Index* <http://pypi.python.org/pypi> sind alle zusätzlichen Pakete von Python aufgelistet.

B.1 Matplotlib

Auf der Projektseite zu matplotlib finden Sie Beispiele und eine Anleitung für die Installation des Pakets <http://matplotlib.sourceforge.net/>.

B.2 NumPy und SciPy

NumPy und SciPy sind OpenSource Erweiterungs-Module für Python, die schnelle Funktionen für mathematische und numerische Routinen bereitstellen. NumPy (Numeric Python) dient zur Bearbeitung von großen Arrays und Matrizen mit numerischen Daten.

SciPy (Scientific Python) erweitert die Funktionalität von NumPy mit nützlichen Funktionen wie Minimierung, Regression, Fourier-Transformation und vielen anderen.

Sowohl NumPy als auch SciPy sind nicht standardmässig installiert. Bei der Installation sollte man beachten, dass man NumPy vor SciPy installieren muss. SciPy findet man unter <http://www.scipy.org/download>.

B.3 SciTools

Das Paket *SciTools* finden Sie unter <http://code.google.com/p/scitools>. Für Windows existiert ein Installer. Mac/Linux Nutzer können das Paket aus dem Source Code erzeugen. (Auf einem Mac Rechner müssen die Entwicklerwerkzeuge installiert sein). Nachdem der Source Code entpackt und an einer wiederauffindbaren Stelle im Dateisystem gespeichert ist, können Sie ein Terminal Fenster öffnen. Mit den folgenden Befehlen lässt sich das Paket installieren.

```
> cd PFAD/scitools-0.8  
> sudo python setup.py install
```

B.4 Weitere Informationen

Ein Python-Kurs (Tutorial) finden Sie unter <http://www.python-kurs.eu/index.php>.

Ein reiches Angebot an Programmieraufgaben finden Sie unter <http://www.programmieraufgaben.ch/>.

Das Projekt Euler behandelt zahlreiche Mathematikaufgaben <http://projecteuler.net/> (englisch)