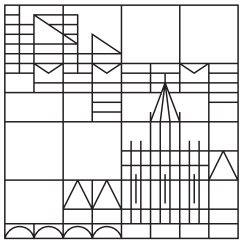
Numerische Berechnungen mit Python

Eine Einführung für das numerische Rechnen mit Python



Fachbereich Mathematik und Statistik
Arbeitsgruppe Numerische Optimierung
Prof. Dr. Stefan Volkwein

Status: 12.2021

Numerisches Rechnen in Python Notebook created by: Florian Wolf, WG Numerical Optimization, based on the lecture notes created by Christian Jäkle, WG Numerical Optimization. Questions, annotations and mistakes to: agvolkwein.oppy@uni-konstanz.de **Please report mistakes immediately.** Last revised: 13.12.2021

Inhaltsverzeichnis

Numerisches Rechnen in Python

Notebook created by: Florian Wolf, WG Numerical Optimization
based on the lecture notes created by Christian Jäkle, WG Numerical Optimization

Questions, annotations and mistakes to: agvolkwein.oppy@uni-konstanz.de

Please report mistakes immediately.

Last revised: 13.12.2021

1 Einführung

Python ist eine universelle, interpretierte höhere Programmiersprache, welche den Anspruch hat, einen gut lesbaren und knappen Programmierstil zu haben. Beispielsweise werden hier Blöcke eingerückt strukturiert, anstatt mit geschweiften Klammern zu arbeiten.

Python unterstützt mehrere Programmierparadigmen, unter anderem objektorientiertes Programmieren und funktionales Programmieren. Diese Einführung geht dabei nicht auf die Objektorientierung ein. Sie dient im Wesentlichen als Einführung für das numerische Rechnen mit Python und ist hauptsächlich an die Matlab-Einführung von Dr. Eberhard Luik, Universität Konstanz, angelehnt. Die meisten Beispiele stammen ebenfalls von dort. Gelegentlich stammen auch einige Beispiele von https://www.python-kurs.eu/index.php.

1.1 Installation

Anaconda ist eine Open-Source-Distribution für die Programmiersprachen Python und R, die alle wichtigen Pakete für das numerische Rechnen mit Python, die Entwicklungsumgebung Spyder und verschiedene Kommandozeileninterpreter enthält. Die aktuellste Version kann für Windows, Mac und Linux heruntergeladen werden und ist hier zu finden: https://www.anaconda.com/download/. Eine vollständige Installationsanleitung ist unter https://docs.anaconda.com/anaconda/install/ verfügbar.

1.2 Virtual Environments

Anaconda erlaubt es uns virtuelle Umgebungen (engl. virtual environments) zu erstellen und verwalten. Eine virtuelle Umgebung ist eine isolierte Arbeitskopie von Python, welche ihre eigenen Pfade und Dateien enthält. So kann man mit bestimmten Versionen von Paketen und Python selbst arbeiten, ohne andere Python Projekte zu beeinflussen. Dies macht es uns leicht an verschiedenen Projekten arbeiten zu können, ohne dabei auf Abhängigkeiten bei Versionen von Python oder Paketen achten zu müssen.

Die Shell-Befehle zum Erstellen, Verwalten und Löschen von virtuellen Umgebungen gibt es unter https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html, eine Kurzversion möchten wir hier vorstellen.

Folgenden Shell-Befehle laden zunächst die neueste Anaconda Version und erstellen eine virtuelle

Umgebung mit dem Name `yourenvname` mit Python-Version x.x. Mit dem letzten Befehl des Codeblocks lassen sich einzelne Pakete (z.B. numpy) mit ihrer aktuellen (oder einer spezifischen) Version in der Umgebung installieren:

```
# zeige aktuelle Version von anaconda
conda -V
# lade ggf. neueste Version herunter
conda update conda
# erstelle neue Umgebung mit dem Namen yourenvname und Pythonversion x.x
conda create -n yourenvname python=x.x anaconda
# aktiviere die Umgebung
source activate yourenvname
# Installation des Paketes [package] in der Umgebung
conda install -n yourenvname [package]
```

Mit folgenden Befehlen kann die Umgebung deaktiviert oder gelöscht werden:

```
# Deaktiviere Umgebung
source deactivate
# Lösche Umgebung
conda remove -n yourenvname -all
```

Weitere Informationen und die zugehörigen Windows-Befehle befinden sich unter Conda Cheatsheet.

2 Allgemeines

Wir wollen uns zunächst einen allgemeinen Überblick über die Funktionsweise der Programmiersprache Python verschaffen und anschließend in das numerische Rechnen einsteigen.

2.1 Help-Funktion

Mit der Funktion help() kann man die Dokumentation, also in der Regel die vollständige Beschreibung, einer Klasse, Funktion, etc. aufrufen. So liefert der Aufruf help(str) die Beschreibung der Klasse string, also den Typ, eine Kurzbeschreibung, vordefinierte Funktionen und vieles mehr.

[ ]:
```
help(str)
```

2.2 Variablen

Variablen sind in der Programmierung von essentieller Bedeutung. Sie dienen zum Speichern jeglicher Werte innerhalb eines Programmes. Hier ein kleines Beispiel:

[2]:
```
# integer
x = 5
print(type(x))
# flaots
y = 5.0
print(type(y))
# strings
```

```
z = "Test"
print(type(z))
# boolean
w = False
print(type(w))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

Die mit # beginnenden Zeilen des obigen Abschnittes sind Kommentare, also Zeilen, welche das Programm ignoriert. Diese dienen dazu das Programm bzw. einzelne Teile des Programmes zu beschreiben und zu erklären.

2.3 Datentypen und Typkonversion

An dieser Stelle möchten wir eine Auswahl primitiver Datentypen vorstellen, welche mittels Typkonversion ineinander überführt werden können. Um den Typ einer Variable zu bestimmen, benutzen wir die Funktion type().

[3]:
```
type(x)
```

[3]:
```
int
```

Wir wollen dies an einem Minimalbeispiel illustrieren:

[4]:
```
x = 5
print(type(x))
# String: str()
# Zeichenketten jeglicher Art
z = "Hello World!"
x = str(x)
print(type(x))
x = float(x)
print(type(x))
print(x)
```

```
<class 'int'>
<class 'str'>
<class 'float'>
5.0
```

Mit Listen und Dictionaries werden wir später noch zwei weitere primitive Datentypen kennenlernen.

2.4 Logische Ausdrücke

Beim Programmieren besteht häufig der Wunsch, eine bestimmte Anweisung von einer Bedingung abhängig zu machen. Zum Beispiel kann man die Quadratwurzel einer reellen Zahl a nur für $a \geq 0$ berechnen. Eine solche Bedingung nennt man einen logischen Ausdruck. Logische Ausdrücke können nur die Werte wahr (engl. true) oder falsch (engl. false) annehmen.

Logische Ausdrücke in der Mathematik sind zum Beispiel

$$x < 10$$
$$a + b \geq y + 5$$
$$x^2 + y^2 = 5$$
$$a \neq \sqrt{x - y}$$

In Python gibt es diese Vergleichsoperatoren und die Operatoren und, oder sowie nicht ebenfalls:

Python	math. Bedeutung
<	$<$
<=	$\leq$
>	$>$
>=	$\geq$
==	$=$
!=	$\neq$
and	und
or	oder
not	nicht

Damit lässt sich beispielsweise Folgendes realisieren:

[5]:
```
x = 5
y = 10

print(x == 6)
print(x != 6)
print((x==6) or (y%2 == 0))
```

```
False
True
True
```

2.5 Print-Anweisungen und Stringformatierungen

Python bietet verschiedene Möglichkeiten Strings zu formatieren und print-Anweisungen zu modifizieren. An dieser Stelle wollen wir alle der vier Möglichkeiten kurz vorgestellen und auf unsere Empfehlung etwas ausführlicher eingehen. Hier nun die Möglichkeiten mit einem Minimalbeispiel:

1. Alte Weise:

```
name = "Bob"
print("Hello, %s"%name)
```

2. Neue Weise:

```
name = "Bob"
print("Hello, {name:s}".format(name))
```

3. Literalstringinterpolation bzw. f-strings (python 3.6+)

```
name = "Bob"
print(f"Hello, {name}")
```

4. Template Strings

```
from string import Template
name = "Bob"
t = Template('Hello, $name!')
t.substitute(name=name)
```

Wir erhalten folgende exemplarische Ausgabe für die vier Möglichkeiten:

[6]:
```
print(" # ------- Alte Version ------- #")
name = "Bob"
print("Hello, %s"%name)
print(" # ------- Neue Version ------- #")
name = "Bob"
print("Hello, {0:s}".format(name))
print(" # ------- f-strings ------- #")
name = "Bob"
print(f"Hello, {name}")
print(" # ------- Template Strings ------- #")
from string import Template
name = "Bob"
t = Template('Hello, $name!')
t.substitute(name=name)
```

```
 # ------- Alte Version ------- #
Hello, Bob
 # ------- Neue Version ------- #
Hello, Bob
 # ------- f-strings ------- #
Hello, Bob
 # ------- Template Strings ------- #
```

[6]:
```
'Hello, Bob!'
```

Auch wenn Variante 4 ein mächtiges Tool darstellt, ist es für unsere Zwecke, sprich kurze Ausgaben um Wertesequenzen anzuzeigen, deutlich zu aufwändig und für den Einstieg auch sehr kompliziert. Variante 3 bietet sich für schnelle Ausgaben an, ist aber ein wenig umständlicher, sobald es um mehrere Argumente und die Formatierung derer geht.

Die am häufigsten verwendeten print-Varianten sind die Optionen 1 und 2. Da Option 2 noch ein paar kleinere Vorteile bietet, z.B. das leichtere Verändern der Reihenfolge bei mehreren Argumenten, empfehlen wir Option 2 und werden diese auch im weiteren Verlauf dieses Skriptes verwenden.

2.5.1 `{}.format()`-print

An dieser Stelle wollen wir eine Einführung in print-Anweisungen mit der `{}.format()`-Methode geben. Die grundlegende Syntax haben wir oben bereits gesehen. Nun möchten wir ein wenig mehr ins Detail gehen.

2.5.1 Mehrere Argumente

Wenn wir mehrere Argumente an verschiedenen Stellen der print-Anweisung ausgeben möchten, bietet die format-Methode, ähnlich zu Listen, einen Zugriff auf die Argumente über ihren Index. Wir betrachten folgendes Minimalbeispiel:

In der ersten Zeile sehen wir, dass die Argumente in der hinten angegebenen Reihenfolge ausgegeben werden, sofern wir nichts näher spezifizieren.

Wollen wir die Reihenfolge ändern, so genügt es, den Index des Elements an der entsprechenden Stelle anzugeben (siehe Zeile 2).

[7]:
```
print("My Name is {}. My age is {} and I live in {}".format("Bob", "19",␣
 ↪"Konstanz"))
print("My Name is {2}. My age is {1} and I live in {0}".format("Bob", "19",␣
 ↪"Konstanz"))
```

```
My Name is Bob. My age is 19 and I live in Konstanz
My Name is Konstanz. My age is 19 and I live in Bob
```

Wir halten zusätzlich fest, dass es bisher nicht nötig war, den Datentyp in der geschweiften Klammer anzugeben. Die Angabe dessen ermöglicht uns einige weitere Formatierungsoptionen. Die grundlegende Syntax der folgenden Befehle ist dabei immer gleich:

```
print("<Text> {<pos>:[opt]<type>} <Text> {<pos>:[opt]<type>}".format(<arg1>, <arg2>))
```

Dabei steht `<pos>` für den Index des gewünschten Elements, `[opt]` für einen optionalen Formatierungsparameter (vor allem bei numerischen Ausgaben wichtig) und `<type>` für den Datentyp des an dieser Stelle einzufügenden Arguments. Der Doppelpunkt trennt dabei das Element und dessen entsprechende Formatierung voneinander.

2.5.1 Strings ausgeben

Wie wir eben gesehen haben, ist die Ausgabe von Strings relativ leicht, sie erfolgt mit dem Typparameter `s`. Der optionale Parameter bietet und bswp. die Möglichkeit die Länge des auszugebenden Strings zu begrenzen. Im folgenden Befehl ist die Länge der Ausgabe auf 5 Zeichen beschränkt.

[8]:
```
print('{:.5s}'.format('xylophone'))
```

```
xylop
```

Es gibt noch eine Vielzahl weiterer Optionen, z.B. das links- oder rechtsbündige ausrichten, welche an dieser Stelle aber nicht behandelt werden sollen.

2.5.1 Zahlen ausgeben

Für Zahlen verwenden wir als Typ den Wert `d` für Integers, also ganzzahlige Werte, und `f` für Floats, also Gleitkommazahlen. Wie bei Strings auch, können wir hier eine Darstellung der Zahl festlegen. Mittels `:x.yf` wird eine Gleitkommazahl mit `x` Stellen vor und `y` Stellen nach dem Komma dargestellt. Mittels `:0x.yf` legen wir fest, dass die fehlen Stellen vor dem Komma mit Nullen von links gefüllt werden soll. Wir illustrieren dies am folgenden Minimalbeispiel:

[9]:
```
# Ohne Nullen
print('{:6.2f}'.format(3.141592653589793))
# Mit Nullen auffüllen, für einheitliche Größe (z.B. für tabellarische␣
 ↪Ausgaben)
print('{:06.2f}'.format(3.141592653589793))
```

```
  3.14
003.14
```

Wir sehen, dass Jupyter die Werte automatisch ausrichtet.

Bei ganzzahligen Werten ist die Angabe von Nachkommastellen selbstverständlich sinnfrei, die Möglichkeit, die Länge der Stellen anzugeben und diese mit Nullen von links aufzufüllen, besteht natürlich dennoch. Falls die Angabe der Stellen die Länge der Zahl unterschreitet, wird diese trotzdem in voller Länge ausgegeben.

[10]:
```
print("{:05d}".format(70000))
```

```
70000
```

Wir können auch die Ausgabe von Vorzeichen steuern und diese sogar forcieren. So werden negative Zahlen automatisch mit ihrem Signum versehen, bei positiven können wir dies mit dem nächsten Befehl erzwingen:

[11]:
```
print('{:+d}'.format(42))
```

```
+42
```

Will man für positive Werte kein Vorzeichen ausgeben, aber dennoch eine einheitliche Formatierung, so ist es sinnvoll ein Leerzeichen nach dem Doppelpunkt einzufügen. Dieses stellt entweder Platz für ein - bereit, oder richtet positive Werte bündig an negativen Werte aus. Wir sehen das im nächsten Minimalbeispiel:

[12]:
```
print('{: d}'.format((- 23)))
print('{: d}'.format((42)))
```

```
-23
 42
```

Wir können auch die Position eines möglichen Vorzeichens steuern und bspw. linksbündig ausrichten. Dies erfolgt mittels `{:=+5d}`. Dies sorgt dafür, dass positive Werte ein +erhalten, die Gesamtausgabe immer 5 Stellen lang ist (inklusive Vorzeichen, siehe größere Zahl) und das Vorzeichen immer am linken Rand steht. Wir betrachten ein kurzes Beispiele:

[13]:
```
print('{:=+5d}'.format((- 23)))
print('{:=+5d}'.format((42)))
print('{:=+5d}'.format((42000)))
```

```
-   23
+   42
+42000
```

Wir wollen nun noch einen kurzen Einblick in die Ausgabe wissenschaftlicher Daten angeben. Hierbei ist Konvention, Zahlen in der Exponentialdarstellung, also in der Form $a \cdot 10^b$, auszugeben. Hierbei

ist a die Mantisse und b der (ganzahlige) Exponent (zur Basis 10). Auch diese Möglichkeit gibt es als Ausgabeoption, wahlweise mit Parameter `E` oder `e`:

[14]:
```
print('{:.5E}'.format(0.09112346))
print('{:.5e}'.format(0.09112346))
```

```
9.11235E-02
9.11235e-02
```

Weitere Informationen befinden sich wie immer in der Python String Dokumentation.

2.5.1 Datum und Zeiten ausgeben

Bei vielen Programmen ist es interessant, zu entsprechenden Werten Zeitstempel (ggf. mit Datum) zu haben, um etwa Laufzeiten zu berechnen oder Werte in einem Graphen zur Zeit auftragen zu können. Auch hier bietet die `format`-Methode eine sehr schlichte und elegante Lösung. Mittels `from datetime import datetime` Laden wir ein Paket, welches uns Zugriff auf Datum und Uhrzeit des Computers ermöglicht (siehe `import`-Abschnitt für mehr Details) und diese Daten in ein genormtes Format überträgt. Die Ausgabe dieser Funktionen können wir sehr leicht formatieren und auf dem Bildschirm ausgeben:

[15]:
```
from datetime import datetime
# Bringe eigene Daten in passendes Format
print('{:%Y-%m-%d %H:%M}'.format(datetime(2001, 2, 3, 4, 5)))
# erhalte aktuelles Datum, sowie Uhrzeit und gebe diese aus
print('{:%Y-%m-%d %H:%M}'.format(datetime.now()))
# gebe bspw. nur Uhrzeit aus
print('{:%H:%M}'.format(datetime.now()))
```

```
2001-02-03 04:05
2021-12-10 15:50
15:50
```

Das Module `datetime` ist noch deutlich umfassender und bietet Funktionen, welche nur die Zeit, nur das Datum oder Zeitdifferenzen ausgeben können. Auch die `format`-Methode ist noch deutlich umfangreicher. Für mehr Informationen lohnt es sich einen Blick in die Dokumentation zu werfen.

2.6 Logische Verzweigungen

Verzweigungen dienen dazu, in Abhängigkeit vom Wert eines logischen Ausdruckes verschiedene Programmteile auszuführen, um so den Verlauf des Programmes zu steuern. Dabei unterscheidet man einseitige, zweiseitige und Mehrfach-Verzweigungen.

2.6.1 Einseitige Verzweigung

Diese haben in Python die Form

```
if <logischer Ausdruck>:
    Anweisungen (if-Block)
```

Wichtig ist, dass nach dem logischen Ausdruck und dem Doppelpunkt eingerückt wird. Bei der Programmausführung wird zunächst der logische Ausdruck ausgewertet. Hat dieser den Wert True,

so werden die Anweisungen des if-Blockes ausgeführt, andernfalls wird in die Zeile gesprungen, die nicht mehr eingerückt ist.

Als Beispiel berechnen wir die Quadratwurzel einer nichtnegativen reellen Zahl:

[16]:
```
x = float(input("x = "))
if x >= 0:
    print(x**(1/2))
```

```
x = 10
3.1622776601683795
```

2.6.2 Zweiseitige Verzweigung

Hier hat man zusätzlich noch die Möglichkeit, einen Programmteil auszuführen, falls der logische Ausdruck den Wert *False* hat. Die Form ist

```
if <logischer Ausdruck>:
    Anweisungen (if-Block)
else:
    Anweisungen (else-Block)
```

Hat bei der Programmausführung der logische Ausdruck den Wert True, so wird der if-Block ausgeführt. Hat er den Wert False, so wird der else-Block ausgeführt.

Wir wollen unser obiges Beispiel nun um eine Fehlermeldung ergänzen, wenn der Eingabewert negativ ist.

[17]:
```
x = float(input("x = "))
if x >= 0:
    print(x**(1/2))
else:
    print("Fehler: Eingabe muss positive reelle Zahl sein.")
```

```
x = -10
Fehler: Eingabe muss positive reelle Zahl sein.
```

2.6.3 Mehrfachverzweigung

Auch Mehrfachverzweigungen lassen sich in Python realisieren. Diese haben die Struktur

```
if <logischer Ausdruck 1>:
    Anweisungen (if-Block)
elif <logischer Ausdruck 2>:
    Anweisungen (elif- Block)
else:
    Anweisungen (else-Block)
```

Bei der Programmausführung wird zuerst der logische Ausdruck 1 ausgewertet. Hat dieser den Wert True, so wird der if-Block ausgeführt. Hat der logische Ausdruck 1 den Wert False, dann wird als nächstes der logische Ausdruck 2 berechnet; hat er den Wert True, so wird der elif-Block ausgeführt, andernfalls der else-Block. In den else-Block gelangt man nur, wenn sowohl der logische Ausdruck 1 als auch der logische Ausdruck 2 den Wert False haben.

Wir wollen nun exemplarisch die Signumsfunktion

$$\text{sign}(x) := \begin{cases} 1 & \text{wenn } x > 0 \\ 0 & \text{wenn } x = 0 \\ -1 & \text{wenn } x < 0 \end{cases}$$

implementieren.

[18]:
```
x = float(input("x = "))
print(type(x))
if x > 0:
    sign = 1
elif x == 0:
    sign = 0
else:
    sign = -1
print(sign)
```

```
x = -2
<class 'float'>
-1
```

Anmerkungen: 1. Es sind mehrere elif-Teile möglich. 2. In einem if-Block bzw. else-Block dürfen mehrere Anweisungen stehen, darunter selbst wieder if-Anweisungen.

2.7 Schleifen

Beim Programmieren besteht häufig der Wunsch, gewisse Programmteile mehrfach zu durchlaufen (mit verschiedenen Daten). Dazu dient das Konzept der Schleifenbildung. Python kennt hierfür verschieden Möglichkeiten. Auch hier muss der jeweils auszuführende Block eingerückt werden.

2.7.1 for-Schleife

Im folgenden sehen wir die allgemeine Syntax der for-Schleife in Python. Sequenz steht hierbei für ein iterierbares Objekt (z.B. ein Array).

```
for Variable in Sequenz:
    Anweisungen
else:
    Anweisungen
```

Dabei ist der optionale else-Block eine Besonderheit von Python. Semantisch funktioniert der optionale else-Block der for-Anweisung wie folgt. Er wird nur ausgeführt, wenn die Schleife nicht durch eine break-Anweisung abgebrochen wurde. Das bedeutet, dass der else-Block nur dann ausgeführt wird, wenn alle Elemente der Sequenz abgearbeitet worden sind.

Trifft der Programmablauf auf eine break-Anweisung, so wird die Schleife sofort verlassen und das Programm wird mit der Anweisung fortgesetzt, die der for-Schleife folgt, falls es überhaupt noch Anweisungen nach der for-Schleife gibt. Ein kleines Beispiel verdeutlicht die Funktionsweise des else-Blockes:

[19]:
```
essbares = ["Schinken", "Fruehstuecksfleisch", "Eier", "Nuesse"]
for essen in essbares:
```

```
    if essen == "Fruehstuecksfleisch":
        print("Bitte kein Fruehstuecksfleisch!")
        break
    print("Grossartig, {}".format(essen))
else:
    print("Super, kein Fruehstuecksfleisch!")
print("Jetzt bin ich fertig mit essen")
```

```
Grossartig, Schinken
Bitte kein Fruehstuecksfleisch!
Jetzt bin ich fertig mit essen
```

Entfernen wir das Frühstücksfleisch aus der obigen Liste, so erhalten wir die Ausgabe:

[20]:
```
essbares = ["Schinken", "Eier", "Nuesse"]
for essen in essbares:
    if essen == "Fruehstuecksfleisch":
        print("Bitte kein Fruehstuecksfleisch!")
        break
    print("Grossartig, {}".format(essen))
else:
    print("Super, kein Fruehstuecksfleisch!")
print("Jetzt bin ich fertig mit essen")
```

```
Grossartig, Schinken
Grossartig, Eier
Grossartig, Nuesse
Super, kein Fruehstuecksfleisch!
Jetzt bin ich fertig mit essen
```

Um eine Zählschleife zu simulieren (bspw. um über ein Array zu iterieren) können wir die range-Funktion verwenden:

```
range(0,10)
range(4,50,5)
range(42,-12,-7)
```

Dabei liefert die erste Funktion die Zahlen von 0 bis 9, die zweite von 4 bis 49 in fünfer-Schritten und die dritte von 42 bis -7 in siebener-Schritten. Damit lassen sich zum Beispiel die Zahlen von 1 bis 100 zusammenzählen:

[21]:
```
n = 100

s = 0
for counter in range(1,n+1):
    s += counter # bedeutet s = s + counter

print("Summe von 1 bis {0:d}: {1:d}".format(n,s))

for counter in range(10,1,-2):
    print(counter)
```

```
Summe von 1 bis 100: 5050
```

```
10
8
6
4
2
```

Hierbei ist s += counter eine Kurzschreibweise für s = s + counter.

2.7.2 while-Schleife

Hängt die Anzahl der Schleifendurchläufe von einer logischen Bedingung ab, so verwendet man die while-Schleife:

```
while Bedingung:
    Anweisungen
else:
    Anweisungen
```

Dabei ist der optionale else-Block genau so wie bei einer for-Schleife zu verwenden. Dazu ein Minimalbeispiel:

Es wird so lange eine reelle Zahl über den Bildschirm eingelesen, bis eine positive Zahl eingegeben wird. Erst dann wird mit der Berechnung von $\sqrt{x}$ fortgefahren.

[22]:
```
print("Gib eine positive Zahl ein")
x = float(input("x = "))
while x <= 0:
    print("Fehler: x<=0\nGib eine positive Zahl ein")
    x = float(input("x = "))
print("sqrt(x) = {:5.2f}".format(x**(1/2)))
```

```
Gib eine positive Zahl ein
x = -1
Fehler: x<=0
Gib eine positive Zahl ein
x = 1
sqrt(x) =  1.00
```

2.7.3 Vorzeitiges Beenden einer Schleife

Es gibt zwei Möglichkeiten eine Schleife vorzeitig zu beenden: 1. `break` springt zu der nächsten Zeile in der nicht mehr eingerückt ist. 2. `continue` der aktuelle Schleifendurchlauf wird abgebrochen und die Schleifenabarbeitung mit dem nächsten Schleifendurchlauf fortgesetzt.

2.8 Listen

Eine der grundlegendsten Datenstrukturen in Python ist die Liste. Jedes Element einer Liste ist mit einer Nummer, seinem Index, addressiert. Dabei müssen wir beachten, dass der **Index immer bei 0 beginnt**. Eine Liste wird immer mit eckigen Klammern und mit Kommata getrennten Werten `[Wert1, Wert2]` definiert. Über den Index können wir auf Element mit einer bestimmten

Nummer zugreifen oder uns bestimmmte Teile der Liste ausgeben lassen. Hier ein Minimalbeispiel:

[23]:
```
# Initialisiere Liste
myList = ["Hello", "World", "!"]
# Ausgabe des ersten Elements
print(myList[0])
# Ausgabe des vorletzten Elements
print(myList[-2])
# Ausgabe der Elemente mit Index 0 bis 1
print(myList[0:2])
# Ausgabe der Elemente ab Index 1
print(myList[1:])
```

```
Hello
World
['Hello', 'World']
['World', '!']
```

Zusätzlich gibt es mit der `len()`-Funktion die Möglichkeit die Länge einer Liste zu bestimmen:

[24]:
```
len(myList)
```

[24]:
```
3
```

Wollen wir am Ende einer Liste ein Element einfügen, so eignet sich `<Listenname>.append(<Element>)`:

[25]:
```
myList.append("!")
print(myList)
```

```
['Hello', 'World', '!', '!']
```

Die `.append()`Schreibweise bedeutet, dass für das Element myList, welches eine Instanz der Klasse `list` ist, die Funktion `.append()`definiert ist. Diese Funktion ist allgemein füt jedes Objekt der Klasse Liste definiert. Für weitere Informationen zu Klassen, Instanzen und darauf definierten Funktionen siehe im Abschnitt Klassen.

Mit der Funktion `<Listenname>.insert(Index, <Element>)` können wir ein Element an einer bestimmeten Position einfügen, also beispielsweise:

[26]:
```
myList.insert(2, "?")
print(myList)
# falls der Index "zu groß" ist, wird das Element hinten angefügt und␣
 ↪insert() entspricht append()

myList.insert(20, "?")
print(myList)
```

```
['Hello', 'World', '?', '!', '!']
['Hello', 'World', '?', '!', '!', '?']
```

Es gibt auch die Möglichkeit ein Element in einer Liste zu suchen: die Funktion `<Listenname>.index(<Element>)`" gibt, sofern `<Element>` in der List enthalten ist, den niedrigsten Index dieses Elements aus (falls dieses mehrfach vorkommt).

[27]:
```
myList.index("!")
```

[27]: 3

Falls sich das Element nicht in der Liste befindet, wird folgenden Fehler ausgegeben:

[28]:
```
# myList.index("1")
```

Unsere bisherigen Betrachtungen von Schleifen ermöglichen es uns mittels einer for-Schleife durch eine Liste zu iterieren:

[29]:
```
for x in myList:
    print(x)
```

```
Hello
World
?
!
!
?
```

Wir können dabei auch über den Index gehen, also mit einer Zählschleife:

[30]:
```
for i in range(0, len(myList)):
    print(myList[i])
```

```
Hello
World
?
!
!
?
```

Beim Iterieren durch eine Liste sollte man innerhalb der Schleife vermeiden Elemente in der Liste hinzuzufügen oder gar zu löschen. Bei Nichtbeachtung dieser Regel kann es schnell zu einem "out of range"-Fehler kommen. Welcher Fehler wurde im folgenden Minimalbeispiel begangen und wie lässt sich dieser beheben?

[31]:
```
# for i in range(0, len(myList)):
#     if i%2 == 0:
#         myList.pop(i)
```

2.9 Dictionaries (mapping)

Ein dictionary ist ein assoziatives Feld und besteht aus Schlüssel-Objekt-Paaren. Das heißt zu einem bestimmten Schlüssel gehört immer ein Objekt. Folglich kann man die Schlüssel auf die Objekte abbilden, daher der Name mapping. Als Werte dienen beliebige Objekte. Für Schlüssel gilt die Einschränkung, dass nur immutable (unveränderliche) Datentypen verwendet werden dürfen, also bspw. keine Listen und Dicts. Wir wollen uns mit dictionaries anhand eines kleinen Beispiels vertraut machen:

[32]:
```
# Initialisierung eines kleinen dicts
myDict = {"Katze": "cat", "Haus": "house", "Garten": "garden", "Papier" :␣
↪"paper"}
```

Zu einem Schlüssel, also bspw. `"Katze"` können wir uns nun den zugehörigen Wert ausgeben lassen:

[33]:
```
myDict["Katze"]
```

[33]:
```
'cat'
```

Auch hier lohnt es sich zu testen, was passiert, wenn man einen zweiten Schlüssel mit einem anderen Wert hinzufügt.

Die uns bereits bekannte `len()`-Funktion gibt hier nun die Anzahl an Paaren an:

[34]:
```
len(myDict)
```

[34]:
```
4
```

Mittel Schleifen können wir auch hier alle Schlüssel, Werte oder Paare ausgeben:

[35]:
```
print("#------------------- Schlüssel -------------------#")
print(myDict.keys())
for key in myDict:
    print(key)
print("#------------------- Werte -------------------#")
print(myDict.values())
for key in myDict:
    print(myDict[key])
print("#------------------- Paare -------------------#")
print(myDict)
for key in myDict:
    print("{0:s} : {1:s}".format(str(key), str(myDict[key])))
```

```
#------------------- Schlüssel -------------------#
dict_keys(['Katze', 'Haus', 'Garten', 'Papier'])
Katze
Haus
Garten
Papier
#------------------- Werte -------------------#
dict_values(['cat', 'house', 'garden', 'paper'])
cat
house
garden
paper
#------------------- Paare -------------------#
{'Katze': 'cat', 'Haus': 'house', 'Garten': 'garden', 'Papier': 'paper'}
Katze : cat
Haus : house
Garten : garden
Papier : paper
```

Auch hier bietet sich die `format`-Methode an, um ein Wörterbuch sauber auszugeben. Wir können hierbei in der Ausgabe direkt auf die Schlüssel zugreifen:

[36]:
```
data = {'first': 'HodorFirst', 'last': 'HodorLast!'}
# Zugriff über Schlüsselwort --> entsprechender Wert wird eingefügt
print('{first} {last}'.format(**data))
```

```
HodorFirst HodorLast!
```

Der Präfix `**` steht dafür, dass eine unbekannte Anzahl an Parametern unbekannten Typs übergeben werden (varargs). Wir können auch alle Wertepaare als Tupel zurückgeben:

[37]:
```
# Rückgabe der Wertepaare als Tupel ist auch möglich
myDict.items()
```

[37]:
```
dict_items([('Katze', 'cat'), ('Haus', 'house'), ('Garten', 'garden'),
('Papier', 'paper')])
```

Mit dem Befehl `del` können wir einzelne Einträge entfernen bzw. löschen:

[38]:
```
del myDict["Haus"]
print(myDict)
```

```
{'Katze': 'cat', 'Garten': 'garden', 'Papier': 'paper'}
```

Hierbei ist in der Klammer sowohl die Abgabe des Schlüssels, als auch des Wertes möglich. Wir können auch überprüfen, ob sich ein Wert bzw. ein Schlüssel in unserem Wörterbuch befindet. Die Abfrage `<Name> in <dict>` liefert boolesche Rückgabewerte:

[39]:
```
"house" in myDict
```

[39]:
```
False
```

Mit der Methode `clear()` können wir alle Inhalte eines Wörterbuchs löschen. Das Wörterbuch bleibt noch leer bestehen:

[40]:
```
myDict.clear()
print(myDict)
```

```
{}
```

2.10 Funktionen

2.10.1 def

Selbstdefinierte Funktionen sind Python-Programme, welche Eingabe-Parameter akzeptieren und ein Ergebnis zurückliefern. Funktionen haben folgenden Aufbau:

```
def funktionsname(Parameterliste):
    """
    Kommentar
    """
    Anweisung(en)
```

Die Funktion kann mit beliebigem Dateinamen an einem beliebigen Ort auf dem PC gespeichert werden. Es können auch mehrere Funktionen in einer Datei gespeichert sein. Möchte man diese nun in einer anderen Datei aufrufen, sollte dieses Main-File an demselben Ort gespeichert sein wie die Datei mit der Funktion (oder man gibt bei dem Import den Pfad an). Nun genügt es die Funktion zu importieren:

```
from functions import funktionsname
```

Der Befehl `help()` gibt den Kommentar der Funktion an, welcher in der Regel die Ein- und Ausgabe sowie kurz die Funktionsweise der Funktion beschreibt.

Wir können nun die Signumsfunktion als echte Funktion implementieren:

[41]:
```
def sign(x):
    """
    function to calculate sign(x)

    Paramters:
        x: real number
    Returns:
        sign of the number
    """
    if x > 0:
        signum = 1
    elif x == 0:
        signum = 0
    else:
        signum = -1
    return(signum)
```

Der Aufruf erfolgt durch:

[42]:
```
x = 10
print(sign(x))
```

```
1
```

Innerhalb der Funktion können wir auch Parameter festlegen, welche optional sind. Diese haben also einen voreingestellten Wert (default value), können vom Benutzer aber auch geändert werden:

[43]:
```
def sign(x, show=False):
    """
    function to calculate sign(x)

    Paramters:
        x: real number
        show: default=False
            show result in console
    Returns:
        sign of the number
    """
    if x > 0:
        signum = 1
```

```
    elif x == 0:
        signum = 0
    else:
        signum = -1
    if show is True:
        print("sign({:5.3f})={:d}".format(x, signum))
    return(signum)
```

In diesem Fall kann die Funktion schlicht das Vorzeichen zurückgeben, falls dies nur für eine Berechung benötigt wird, oder direkt eine Ausgabe in der Konsole anzeigen, falls der Benutzer dies möchte. Die Benutzung sieht nun wie folgt aus:

[44]:
```
x = 10
print("#------- Ohne Ausgabe -------#")
sgn = sign(x)
print(sgn)
print("#------- Mit Ausgabe -------#")
sgn = sign(x, show=True)
```

```
#------- Ohne Ausgabe -------#
1
#------- Mit Ausgabe -------#
sign(10.000)=1
```

Bei Funktionen mit mehreren Parametern ist es wichtig, dass nicht default Variablen in der richtigen Reihenfolge übergeben werde. Bei default-Werten ist dies wiederum egal. Folgende sinnfreie Funktion soll dies illustrieren:

[45]:
```
def test(x, y, showX=False, showY=False):
    if showY:
        print("x = {}".format(x))
    if showX:
        print("y = {}".format(y))
    result = x - y
    print(result)
```

Wir beachten, dass `test(2,1)` $\neq$ `test(1,2)` ist, aber `test(2,1, showX=True, showY=True)`=`test(2,1, showY=True, showX=True)`:

[46]:
```
print("Beobachtung 1:")
# Vertauschen der nicht default Werte liefert unterschiedliche Ergebnisse
test(1,2)
test(2,1)
print("Beobachtung 2:")
# Während das Tauschen des defaults irrelevant ist
test(1,2, showX=True, showY=True)
test(1,2, showY=True, showX=True)
```

```
Beobachtung 1:
-1
1
Beobachtung 2:
```

```
x = 1
y = 2
-1
x = 1
y = 2
-1
```

Wir können dieses Problem umgehen, indem wir nicht-default Variablen explizit parametrisieren. So liefern die beiden unterschiedlichen Aufrufe dann schlussendlich doch das gleiche Ergebnis:

[47]:
```
test(x=1,y=2)
test(y=2,x=1)
```

```
-1
-1
```

Diese Option bietet sich oft an, wenn man Funktionen aufruft, deren Parameternamen bekannt, deren Reihenfolge unbekannt ist.

2.10.2 lambda-Funktionen

Für kürzere Funktionen eignen sich sogenannte `lambda`-Funktionen. Dies sind anonyme Funktionen, welche eine beliebige Anzahl an Parametern akzeptieren, aber nur einen Ausdruck zulassen. Die allgemeine Syntax sieht wie folgt aus:

```
lambda arguments : expression
```

Die Stärke von `lambda`-Funktionen liegt in ihrer kurzen Form, sodass sich einfache Funktionen schnell definieren lassen oder auch Funktionen als Rückgabe einer Funktion nur eine sehr kurze Syntax benötigen. Wir wollen zwei Minimalbeispiele implementieren:

(1)
$$f : \mathbb{R} \to \mathbb{R}, x \mapsto x^2$$

(2)
$$\text{power} : \mathbb{N} \to \mathbb{R}[x], n \mapsto \begin{bmatrix} \text{f} : \mathbb{R} \to \mathbb{R} \\ x \mapsto x^n \end{bmatrix}$$

[50]:
```
# (1) Example
f = lambda x: x**2
# (2) Example
def power(n):
    return(lambda x:x**n)
```

Die Verwendung erfolgt nun exakt analog:

[51]:
```
print(f(2))
myDouble = power(2)
print(myDouble(3))
```

```
4
9
```

2.11 Module und Funktionen importieren

Die `import`-Funktion bietet die Möglichkeit, Module (also bspw. ganze Bibliotheken) oder einzelne Funktionen aus externen eigenen Dateien oder installierten Bibliotheken zu laden. Der Übersicht halber stehen das Einbinden von Modulen und Funktionen immmer zu Beginn des Programms. Die allgemeine Syntax dazu lautet:

Laden einer Bibliothek, welche später mit `<Shortcut>.` zur Verfügung steht.

```
import <Name> as <Shortcut>
```

Laden einer einzelnen Funktion aus der Datei `<Name>`, welche global, also ohne Voranstellen eines Punktes zur Verfügung steht.

```
from <Name> import <Funktion>
```

Wir wollen uns nun zwei konkrete Beispiele ansehen:

[52]:
```
# Laden des Numpy-Moduls, welches mit np.<Ziel> zur Verfügung steht
import numpy as np

# Laden der spezifischen Funktion fabs() zur Verwendung des Absolutbetrages
from math import fabs
```

In ihrer Verwendung unterscheiden sich die beiden Möglichkeiten wie folgt:

[53]:
```
# Zugriff auf eine bestimmte Funktion des Moduls durch einen Punkt
print(np.abs(-10))
# globale Verfügbarkeit
print(fabs(-10))
```

```
10
10.0
```

3 Numpy

Viele numerische Berechnungen beruhen auf Matrixberechnungen. Das Paket NumPy (Akronym für "Numeric Python", oder "Numerical Python" bietet hier als Open Source Erweiterung, die Möglichkeit, schnelle numerische Berechnungen mit Python auszuführen. Wir geben hier Grundlagen im Umgang mit diesem. Um das Paket nutzen zu können, müssen wir es zuerst laden. Wie wir oben bereits gesehen haben, gilt Folgendes als Konvention bei ihrer Verwendung:

[54]:
```
import numpy as np
```

3.1 Arrays erzeugen und belegen

Wir behandeln ein d-dimensionales Numpy-Array im folgenden wie Matrizen. So ist zum Beispiel ein eindimensionales Numpy-Array ein Vektor und ein zweidimensionales Numpy-Array eine Matrix.

Wir erhalten zum Beispiel durch den folgenden Befehl die Matrix:

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

[55]:
```
A = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]])
A
```

[55]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

Dabei sind die runden, wie eckigen Klammern zu beachten. Durch

[56]:
```
x = np.array([1, 2, 3, 4])
print(x)
y = np.array([[10],[20],[30]])
print(y)
```

```
[1 2 3 4]
[[10]
 [20]
 [30]]
```

werden die Vektoren

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix} \quad \text{und} \quad y = \begin{pmatrix} 10 \\ 20 \\ 30 \end{pmatrix}$$

erzeugt.

Hierbei ist wichtig, dass der erste Vektor als Element aus $\mathbb{R}^4$ erzeugt wird, also das Format `(4,)` hat, wohingegen der zweite Vektor ein Element aus $\mathbb{R}^{3\times 1}$ ist, also als Matrix mit Format `(3,1)` gepeichert wird. Für eine Matrixmultiplikation ist dies irrelevant, da diese in beiden Fällen durch entsprechendes Transponieren durchgeführt werden kann. Will man bei einem der beiden Formate eine Zeile und/oder Spalte hinzufügen, muss man auf die Dimensionen acht geben. Wir gehen alle Fälle einzeln durch:

[57]:
```
print("Zeilenvektor")
print("Dimension: {}".format(np.shape(x)))
print("\nSpalte hinzufügen")
newcolumn = np.array([1])
x_new = np.hstack((x, newcolumn))
print(x_new)
print("Neue Dimension: {}".format(np.shape(x_new)))
print("\nZeile hinzufügen")
newcolumn = np.array([1, 2, 3, 4])
x_new = np.vstack((x, newcolumn))
```

```
print(x_new)
print("Neue Dimension: {}".format(np.shape(x_new)))


print("\n\nSpaltenvektor")
print("Dimension: {}".format(np.shape(y)))
print("\nSpalte hinzufügen")
newcolumn = np.array([1])
newcolumn = np.array([[1], [2], [3]])
y_new = np.hstack((y, newcolumn))
print(y_new)
print("Neue Dimension: {}".format(np.shape(y_new)))
print("\nZeile hinzufügen")
newcolumn = np.array([[1]])
y_new = np.vstack((y, newcolumn))
print(y_new)
print("Neue Dimension: {}".format(np.shape(y_new)))
```

```
Zeilenvektor
Dimension: (4,)

Spalte hinzufügen
[1 2 3 4 1]
Neue Dimension: (5,)

Zeile hinzufügen
[[1 2 3 4]
 [1 2 3 4]]
Neue Dimension: (2, 4)


Spaltenvektor
Dimension: (3, 1)

Spalte hinzufügen
[[10  1]
 [20  2]
 [30  3]]
Neue Dimension: (3, 2)

Zeile hinzufügen
[[10]
 [20]
 [30]
 [ 1]]
Neue Dimension: (4, 1)
```

Auf ein Element eines Arrays können wir zugreifen, indem wir seine Position angeben. Dabei beginnen **Zeilenindex, Spaltenindex und Indizierung bei Null**. So liefert die Eingabe

[58]:
```
print(A)
A[3,2] = 100
print(A)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
[[  1   2   3   4]
 [  5   6   7   8]
 [  9  10  11  12]
 [ 13  14 100  16]]
```

Wie auch in Matlab gibt es bereits vorhandene Funktionen um Arrays bestimmter Größen mit Nullen oder Einsen zu befüllen. Die beiden folgenden Befehle leisten dies:

[59]:
```
B = np.zeros((4,3))
C = np.ones((2,5))
```

Wir erhalten so

[60]:
```
print(B)
print(C)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

Die $n \times n$ Einheitsmatrix wird in Python mittels Numpy durch das Kommando `np.eye(n,n)` erzeugt. Dabei muss n schon mit einem Wert belegt sein. So liefert

[61]:
```
n = 3
I3 = np.eye(n)
print(I3)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

die 3×3-Einheitsmatrix.

Dabei benötigt man, im Gegensatz zu `np.zeros` und `np.ones` , für den Befehl nur eine Klammer. Dies liegt daran, dass `np.eye` ein anderes Input (ein Tupel) erwartet. Für den interessierten Leser sei hier auf die Dokumentation von numpy verwiesen. Das Kommando

[62]:
```
x = np.arange(0,2.2,0.2)
print(x)
```

```
[0.  0.2 0.4 0.6 0.8 1.  1.2 1.4 1.6 1.8 2. ]
```

erzeugt das eindimensionale Array

$$x = \begin{pmatrix} 0 & 0.2 & 0.4 & \dots & 1.8 & 2.0 \end{pmatrix}.$$

Dabei werden die Werte des halboffenen Intervalls $[0, 2.2)$ in $0.2-$Schritten erzeugt. Dieses eignet sich besonders gut, um später Funktionen in äquidistanten Stellen auszuwerten und zu plotten.

3.2 Teilbereiche eines Arrays

Von einem bereits definierten Array kann man Teilbereiche (Zeilen, Spalten, Teilarrays) ansprechen. Für das Array A

[63]:
```
print(A)
```

```
[[  1   2   3   4]
 [  5   6   7   8]
 [  9  10  11  12]
 [ 13  14 100  16]]
```

liefern die folgenden Befehle die Ergebnisse:

[64]:
```
print(A)
print("Zweite Zeile der Matrix")
u = A[1,:]
print(u)
print("# --------------- #")
print("Dritte Spalte der Matrix")
v = A[:,2]
print(v)
print("# --------------- #")
print("Untermatrix Zeilen 1-3 und Spalten 1-2")
B = A[0:3,0:2]
print(B)
print("# --------------- #")
print("Untermatrix Zeilen 3-4 und Spalten 3-4")
C = A[2:4,2:4]
print(C)
```

```
[[  1   2   3   4]
 [  5   6   7   8]
 [  9  10  11  12]
 [ 13  14 100  16]]
Zweite Zeile der Matrix
[5 6 7 8]
# --------------- #
Dritte Spalte der Matrix
[  3   7  11 100]
# --------------- #
Untermatrix Zeilen 1-3 und Spalten 1-2
[[ 1  2]
 [ 5  6]
 [ 9 10]]
```

```
# --------------- #
Untermatrix Zeilen 3-4 und Spalten 3-4
[[ 11  12]
 [100  16]]
```

Ein Array kann auch mit Hilfe von Teilarrays definiert werden. So erzeugt zum Beispiel die Sequenz eine mit Blockmatrizen belegte Matrix D:

[65]:
```
I=np.eye(2,2)
Z=np.zeros((2,2))
E=np.array([[2, -1], [-1, 2]])
D=np.hstack((np.vstack((E,Z,I)),np.vstack((Z,E,Z)),np.vstack((I,Z,E))))
print(D)
```

```
[[ 2. -1.  0.  0.  1.  0.]
 [-1.  2.  0.  0.  0.  1.]
 [ 0.  0.  2. -1.  0.  0.]
 [ 0.  0. -1.  2.  0.  0.]
 [ 1.  0.  0.  0.  2. -1.]
 [ 0.  1.  0.  0. -1.  2.]]
```

Aus einem Array können Teilbereiche gestrichen werden. Dadurch ändert sich die Größe des Arrays. Betrachten wir das obige Array A und geben dann die folgenden Befehle ein:

[66]:
```
print("Streichen der ersten Zeile:")
G = np.delete(A,(1), axis = 0)
print("G = {}".format(G))
print("Streichen der ersten Spalte:")
H = np.delete(A,(1), axis = 1)
print("H = {}".format(H))
```

```
Streichen der ersten Zeile:
G = [[  1   2   3   4]
 [  9  10  11  12]
 [ 13  14 100  16]]
Streichen der ersten Spalte:
H = [[  1   3   4]
 [  5   7   8]
 [  9  11  12]
 [ 13 100  16]]
```

Ebenso können Zeilen bzw. Spalten in einem Array eingefügt werden. Beispielsweise liefert

[67]:
```
D = np.vstack((A[0:2,:],np.array([21,22,23,24]),A[3,:]))
print(D)
```

```
[[  1   2   3   4]
 [  5   6   7   8]
 [ 21  22  23  24]
 [ 13  14 100  16]]
```

3.3 Rechnen mit Arrays

3.3.1 Addition, Matrixprodukt, Elementw. Mult.

Zwei n-dimensionale Arrays A und B derselben Größe werden addiert bzw. subtrahiert durch

[68]:
```
print(A+D)
print(A-D)
```

```
[[  2   4   6   8]
 [ 10  12  14  16]
 [ 30  32  34  36]
 [ 26  28 200  32]]
[[  0   0   0   0]
 [  0   0   0   0]
 [-12 -12 -12 -12]
 [  0   0   0   0]]
```

Diese Operationen werden (wie aus der Mathematik bekannt) elementweise durchgeführt. Vorsicht ist dagegen bei der Multiplikation geboten, denn hier kennt die Mathematik zwei Möglichkeiten: elementweise Multiplikation und das Matrixprodukt. Dementsprechend gibt es auch in Python die beiden Möglichkeiten

[69]:
```
print("Matrixprodukt:")
print(np.dot(A,D))
print("\nElementweise Multiplikation:")
print(A*D)
```

```
Matrixprodukt:
[[ 126  136  486  156]
 [ 286  312 1018  364]
 [ 446  488 1550  572]
 [2391 2534 4037 2820]]

Elementweise Multiplikation:
[[    1     4     9    16]
 [   25    36    49    64]
 [  189   220   253   288]
 [  169   196 10000   256]]
```

Das Matrixprodukt ist nur definiert, falls A ein $m \times n$-Array und B ein $n \times r$-Array ist und ergibt dann ein $m \times r$-Array. Die elementweise Multiplikation ist dagegen nur definiert wenn beide Matrizen dieselbe Dimension haben. Entsprechend gilt für die Potenz:

[70]:
```
print("Berechne A^3:")
print(np.linalg.matrix_power(A, 3))
print("\nElementweise Potenzierung: ")
print(A**3)
```

```
Berechne A^3:
[[  6200   6960  17920   8480]
 [ 13388  15032  40476  18320]
 [ 20576  23104  63032  28160]
```

```
 [ 54454  61436 206118  75400]]

Elementweise Potenzierung:
[[      1       8      27      64]
 [    125     216     343     512]
 [    729    1000    1331    1728]
 [   2197    2744 1000000    4096]]
```

Die Matrixpotenz A^3 ist nur für ein quadratisches Array (d.h. $n \times n$-Arrays) erklärt. Die elementweise Division zweier Arrays erfolgt mittels

[71]:
```
print("Elementweise Division:")
print(A/D)
```

```
Elementweise Division:
[[1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [0.42857143 0.45454545 0.47826087 0.5       ]
 [1.         1.         1.         1.        ]]
```

was a_{ij}/b_{ij} für $1 \leq i,j \leq n$ bedeutet.

3.3.2 Lösen von LGS

Zum Lösen von linearen Gleichungssystem können wir folgenden Befehl verwenden

[72]:
```
b = np.array([1,2,3,4])
x = np.linalg.solve(A,b)
print("Lösungsvektor: ")
print("x = {}".format(x))
```

```
Lösungsvektor:
x = [ 0.    0.   -0.    0.25]
```

3.3.3 Funktionen auf Arrays (Dim., Norm, . . .)

Numpy liefert einige Funktionen zum Umgang mit Arrays, um bspw. die Größe einer Matrix zu ermitteln oder die Norm eines Vektors zu berechnen. Eine kleine Auswahl soll hier vorgestellt werden. Zur Bestimmung der Dimension eines Arrays eigenet sich Folgendes:

[73]:
```
# Dimension einer Matrix bzw. eines Arrays bestimmen
m,n = np.shape(A)
print(m,n)
```

```
4 4
```

Von einer Matrix lässt sich mit dem folgenden Befehl der Vektor der Diagonale extrahieren sowie aus einem gegebenen Vektor eine Diagonalmatrix machen, welche den Vektor als Diagonale besitzt:

[74]:
```
# Diagonale einer Matrix als Vektor extrahieren
print("Vektor der Diagonalen:")
v = np.diag(A)
```

```
print(v)
# Diagonalmatrix mit v auf der Diagonalen
print("\nDiagonalmatrix:")
H = np.diag(v)
print(H)
```

```
Vektor der Diagonalen:
[ 1  6 11 16]

Diagonalmatrix:
[[ 1  0  0  0]
 [ 0  6  0  0]
 [ 0  0 11  0]
 [ 0  0  0 16]]
```

Oft möchte man eine Matrix transponieren. Dafür gibt es zwei Möglichkeiten, welche das gleiche Ergebnis liefern

[75]:
```
# Mglkeit (1)
AT = np.transpose(A)
print(AT)
# Mglkeit (2)
AT2 = A.T
print(AT2)
```

```
[[  1   5   9  13]
 [  2   6  10  14]
 [  3   7  11 100]
 [  4   8  12  16]]
[[  1   5   9  13]
 [  2   6  10  14]
 [  3   7  11 100]
 [  4   8  12  16]]
```

Um die Norm eines Vektors oder einer Matrix zu bestimmen betrachten wir exemplarisch folgenden Befehle:

[76]:
```
print("---- Vektornormen ----")
x = np.array([1,2,3,4,5])
print("Euklidische Norm:")
print(np.linalg.norm(x, ord=2))
print("1-Norm:")
print(np.linalg.norm(x, ord=1))
print("Inf-Norm:")
print(np.linalg.norm(x, ord=np.inf))
print("---- Matrixnormen ----")
print("Frobeniusnorm:")
print(np.linalg.norm(A, ord='fro'))
print("Zeilensummenorm:")
print(np.linalg.norm(A, ord=np.inf))
```

```
---- Vektornormen ----
```

```
Euklidische Norm:
7.416198487095663
1-Norm:
15.0
Inf-Norm:
5.0
---- Matrixnormen ----
Frobeniusnorm:
106.16496597277278
Zeilensummenorm:
143.0
```

3.4 Konstanten und elementare Funktionen

Folgende Funktionen und Konstanten werden für numerische Berechnungen in Python immer wieder benötigt: - Konstante pi: `np.pi` - Euler'sche Zahl: `np.e` - Exponentialfunktion: `np.exp()` - Sinus/Cosinus/Tangens: `np.sin()`, `np.cos()`, `np.tan()` - Logarithmen zur Basis 2, 10 und e: `np.log10()`, `np.log2()`, `np.log()` - Wurzel und n-te Wurzel: `np.sqrt()`, `<Var>**(1/n)` - Auf- und abrunden: `np.ceil()`, `np.floor()` - Runden auf eine ganze Zahl: `np.round()`

Manchmal auch ganz praktisch: - Zufälliges Element aus $[0,1)^{n\times m}$ für $n,m \in \mathbb{N}$: `np.random.rand(n,m)`

3.5 Speichern und Laden von txt-Dateien

Das Speichern und Laden von Dateien eures Programms ist in vielen verschiedenen Dateiformaten möglich. Wir wollen uns hier auf `.txt`-Dateien beschränken. Das Speichern von Dateien bietet sich immer dann an, wenn große Datenmengen einer Berechnung (z.B. riesige Martizen, deren Einträge berechnet wurden) auch zukünftig noch interessant sind, aber eine Neuberechnung zu aufwändig wäre. Der Befehl `np.savetxt("fname.txt", <Variable>)`, speichert die `<Variable>` in die Datei `fname.txt`. Wir können so beispielsweise unsere obige Matrix A speichern

[77]:
```
np.savetxt("matrixA.txt", A)
```

Im Ordner, in dem sich dieses Notebook befindet, sollte nun eine Datei mit dem Namen `matrixA.txt` existieren, welche die Matrix A enthält. Um diese Werte zu laden gibt es den folgenden Befehl:

[78]:
```
loaded_A = np.loadtxt("matrixA.txt")
print(loaded_A)
```

```
[[  1.   2.   3.   4.]
 [  5.   6.   7.   8.]
 [  9.  10.  11.  12.]
 [ 13.  14. 100.  16.]]
```

Der Befehl `np.savetxt("fname.txt", <Variable>)` akzeptiert noch einige weitere Parameter, bspw. das Format, in dem Zahlen gespeichert werden sollen. Die vollständige Syntax lautet:

```
np.savetxt(fname, A, fmt="%.18e", delimiter=" ",
           newline="\n" ,header=" " , footer=' ',
           comments='# ')
```

4 Plots

4.1 Funktionen plotten

Zum zeichnen von Graphen benötigen wir in der Regel die folgenden beiden Pakete:

[79]:
```
import numpy as np
import matplotlib.pyplot as plt
```

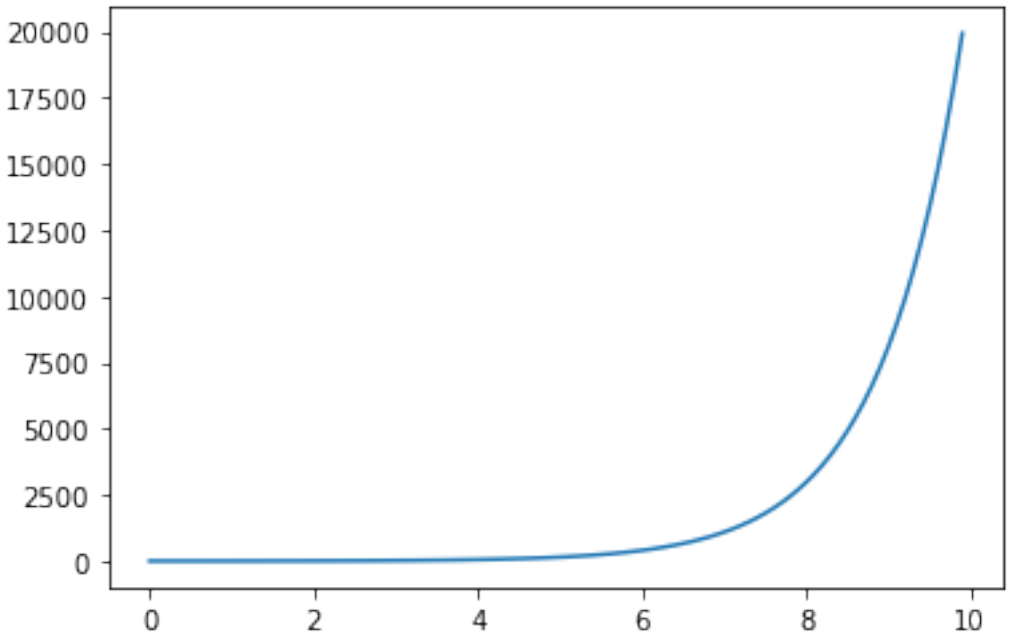
In den folgenden Beispielen werden wir diese Pakete nicht noch einmal explizit laden, sondern nur noch Pakete laden, welche für einzelne Beispiele zusätzlich benötigt werden.

Matplotlib ist ein umfangreiches Graphik-Paket. Wir geben hier nur einen kleinen Einblick über die unzähligen Möglichkeiten, die dieses Paket liefert. Für einen Gesamtüberblick sei auf die Dokumentation unter http://matplotlib.org/index.html verwiesen. Durch die Eingabe eines entsprechenden Graphik Befehls wird ein neues Fenster mit der Überschrift Figure geöffnet. Für Spyder-User muss dies erst aktiviert werden (Tools -> Preferences -> IPython console -> Graphics. Hier im Feld Graphics backend Automatic auswählen). Eine erstellte Graphik kann in verschiedenen Formaten (z.B. Postskript, JPEG, PDF) abgespeichert und so in andere Texte (z.B. Latex-Dokumente) eingebunden werden. Der Befehl `plt.close()` schließt geöffnete Figure-Fenster, was generell vor Beenden des Programmes zu empfehlen ist.

4.1.1 2d-Plots

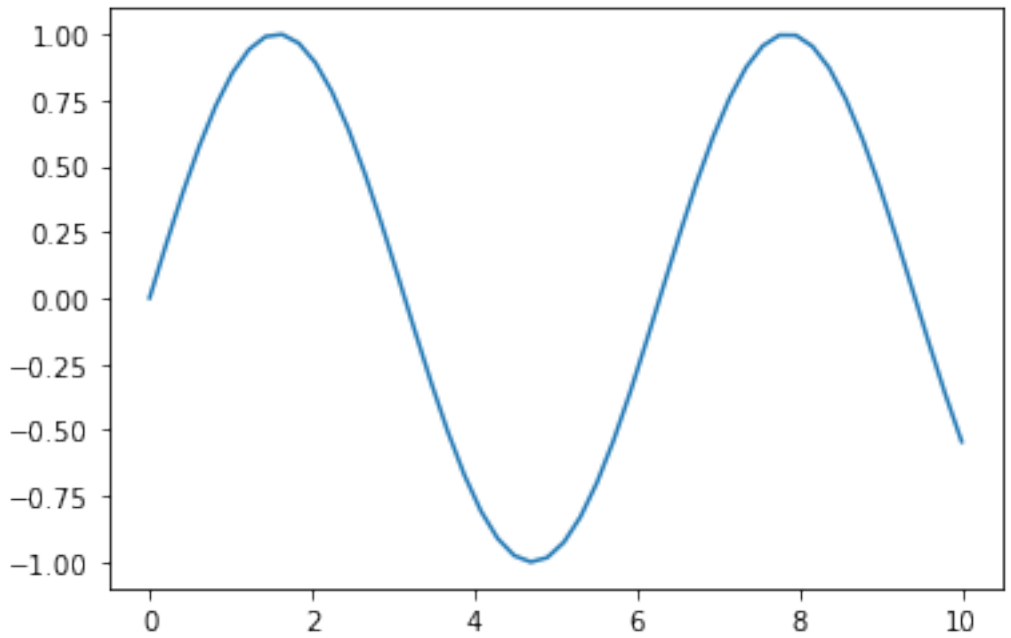
Zum Zeichnen von reellen Funktionen (oder Messergebnissen) in der Ebenen verwendet man das Matplotlib-Kommando plot. Dabei ist zunächst eine Wertetabelle von der Funktion zu erstellen, welche als Parameter an das plot - Kommando übergeben wird. Das folgende Beispiel zeichnet die Funktion $\sin(x)$ im Intervall [0,10]:

[80]:
```
x = np.arange(0,10,0.1)
y = np.exp(x)
plt.plot(x,y)
plt.show()
```



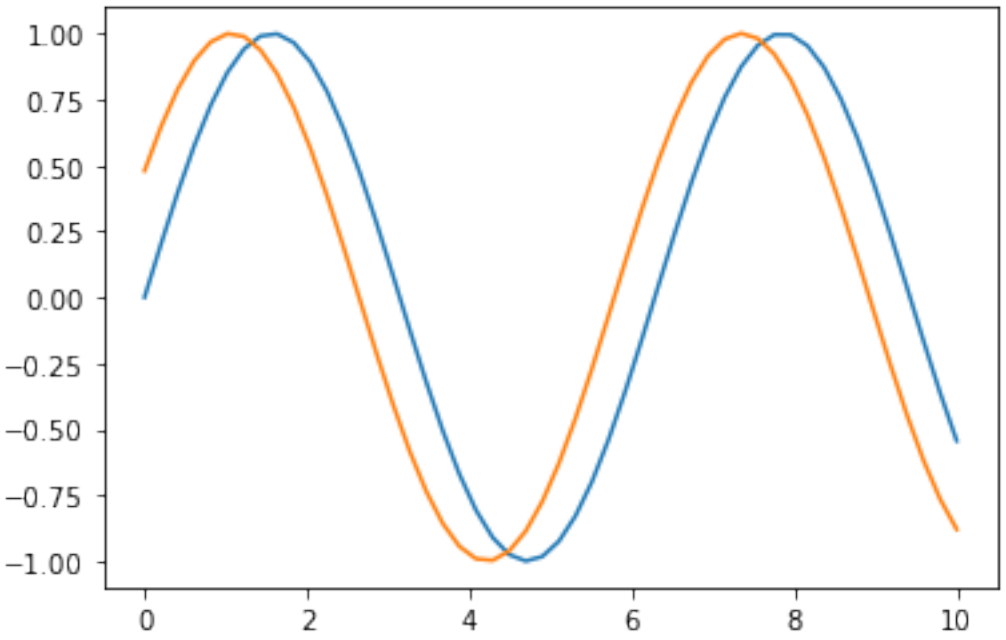
Alternativ zu dem Befehl `np.arange(Start, Stop, schrittlänge)` kann auch der Befehl `np.linspace(Start, Stop, Anzahl Punkte, endpoint=True oder False)` verwendet werden. Das gleiche Beispiel sieht dann so aus:

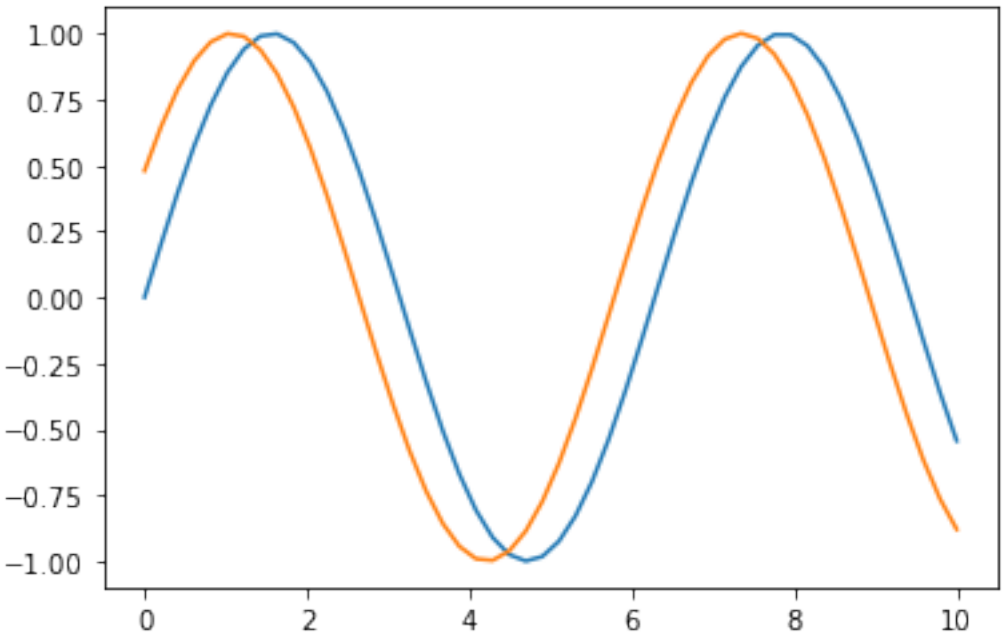
[81]:
```
x = np.linspace(0, 10, 50, endpoint=True)
y = np.sin(x)
plt.plot(x,y)
plt.show()
```



Möchte man mehrere Kurven in einem Plot haben, genügt es den Befehl `plt.show()` an das Ende zu setzen. Mit einem Plot-Befehl können auch mehrere Kurven gleichzeitig gezeichnet werden. Wir erhalten damit zwei Möglichkeiten, welche völlig gleichwertig sind. Bei mehr als zwei Datensätzen ist der Übersicht halber die zweite Variante deutlich besser. Wir erhalten folgende Kurven:

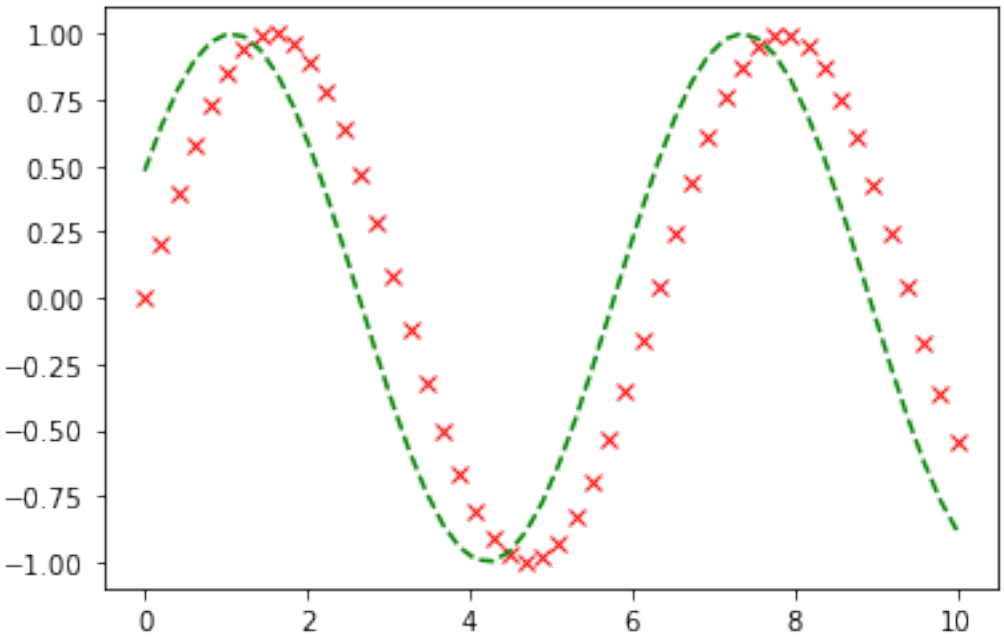
[82]:
```
x = np.linspace(0, 10, 50, endpoint=True)
y = np.sin(x)
z = np.sin(x+0.5)
plt.plot(x,y,x,z)
plt.show()

x = np.linspace(0, 10, 50, endpoint=True)
y = np.sin(x)
z = np.sin(x+0.5)
plt.plot(x,y)
plt.plot(x,z)
plt.show()
```





Durch einen weiteren Parameter werden in der Zeichnung Linientyp, Farbe und Markierungstyp bestimmt. Das folgende Minimalbeispiel soll dies illustrieren, ehe eine ausführliche Tabelle folgt. Hier bietet es sich an kurz zu überlegen welchen Ausgang man vom folgenden Code erwartet und den Plot mit der Erwartung zu vergleichen.

[83]:
```
x = np.linspace(0, 10, 50, endpoint=True)
y = np.sin(x)
z = np.sin(x+0.5)
plt.plot(x,y, "rx")
plt.plot(x,z, "g--")
plt.show()
```



Linientyp: Für den Linientyp gibt es die folgenden Möglichkeiten: * `-`: durchgezogene Linie * `--`: gestrichelte Linie * `-.`: Strichpunkt-Linie * `:`: punktierte Linie

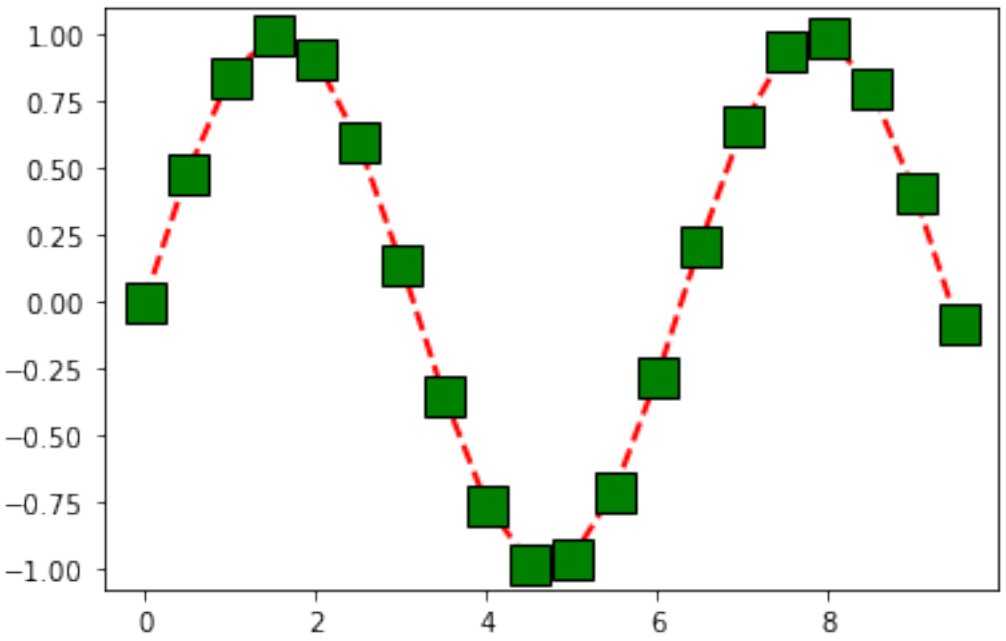
Färbungen: * `b`: blau * `g` : grün * `r` : rot * `c`: cyan * `m`: magenta * `y`: gelb * `k`: schwarz * `w`: weiß

Weitere Optionen sowie Möglichkeiten für verschiedene Marker sind unter https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.plot.html verfügbar. Die Graphik wird in einem neuen Fenster dargestellt, in dem sie weiter bearbeitet werden kann (all diese Optionen kann man auch direkt im Code festlegen, wie wir später sehen werden).

Ferner kann man die Dicke der Linien wählen. Dazu werden weitere Parameter in das plot-Kommando aufgenommen. Wir testen das folgende Python-Programm:

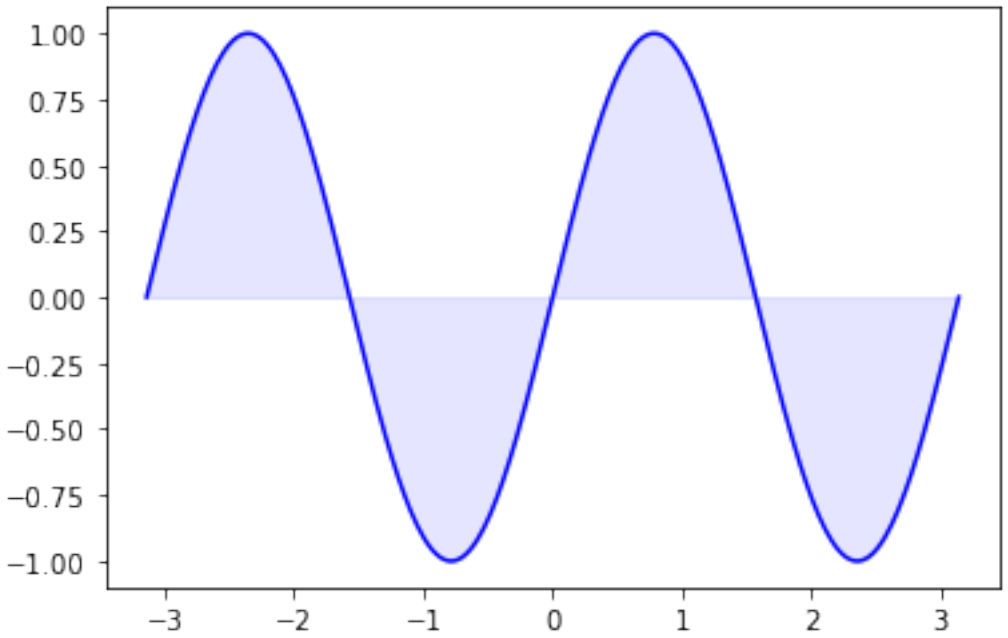
[84]:
```
x = np.arange(0,10,0.5)
y = np.sin(x)
plt.plot(x,y,'rs--',MarkerEdgeColor='k', MarkerFaceColor='g', MarkerSize=15,␣
 ↪LineWidth=2)
plt.show()
```

```
/var/folders/z4/vv8gb7j93tn006mx_gs44wdm0000gp/T/ipykernel_69304/139605796.py:
 ↪3:
MatplotlibDeprecationWarning: Case-insensitive properties were deprecated in␣
 ↪3.3
and support will be removed two minor releases later
  plt.plot(x,y,'rs--',MarkerEdgeColor='k', MarkerFaceColor='g', MarkerSize=15,
LineWidth=2)
```



Schließlich können noch einige Regionen in der Graphik eingefärbt werden. Dazu betrachten wir folgendes Beispiel:

[85]:
```
n = 256
X = np.linspace(-np.pi,np.pi,n,endpoint=True)
Y = np.sin(2*X)

plt.plot (X, Y, color='blue')
# Alpha ist der Filter bzw. die Deckkraft der Farbe
plt.fill_between(X, 0, Y, color='blue', alpha=.1)
plt.show()
```



Auch hier bietet es sich an mit einigen Parametern zu experimentieren und die verschiedenen Ausgänge zu betrachten.

4.1.2 3D-Plots

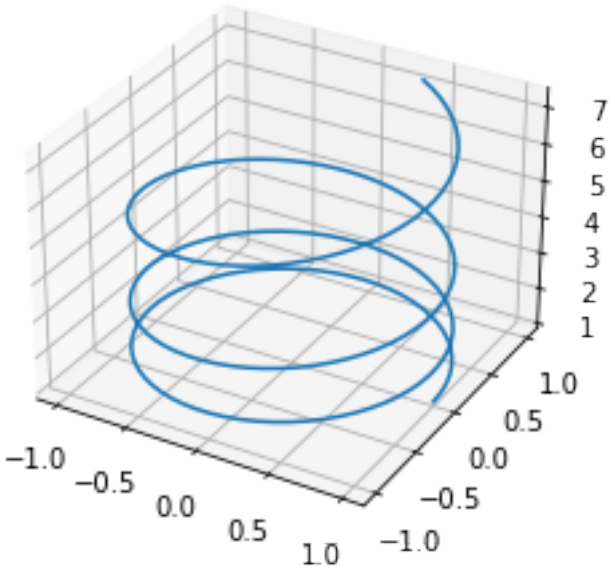
Wir starten mit dem Zeichnen einer Raumkurve. Diese sind in der Regel durch eine Parametrisierung gegeben:

$$(x(t), y(t), z(t)) \quad \text{mit } t \in [a, b].$$

Dabei sind $x(t)$, $y(t)$ und $z(t)$ reelle und differenzierbare Funktionen. Wir betrachten das folgende Beispiel:

[86]:
```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
t = np.linspace(0, 2, 200, endpoint=True)
x = np.cos(10*t)
y = np.sin(10*t)
z = np.exp(t)
```

```
ax.plot(x, y, z)
plt.show()
```



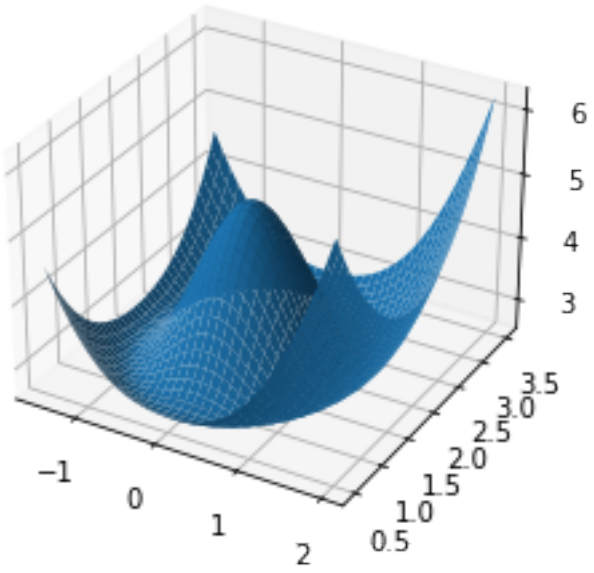
Als nächstes betrachten wir den Graphen einer Funktion $f : D \subset \mathbb{R}^2 \to \mathbb{R}$. Dieser kann als "Gebirge" über dem Definitionsbereich oder als Höhenkarte dargestellt werden. Für die Darstellung als Gebirge gibt es mehrere Varianten. Wir veranschaulichen dies exemplarisch für die Funktion

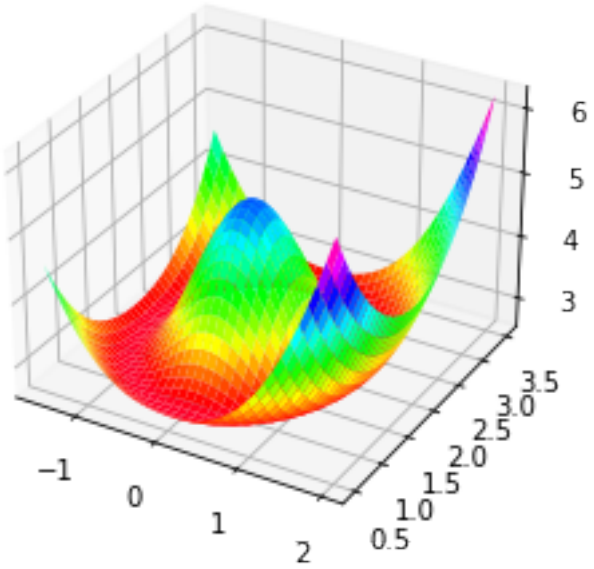
$$f(x,y) := 5\exp(-x^2-(y-2)^2)+x^2+(y-2)^2$$

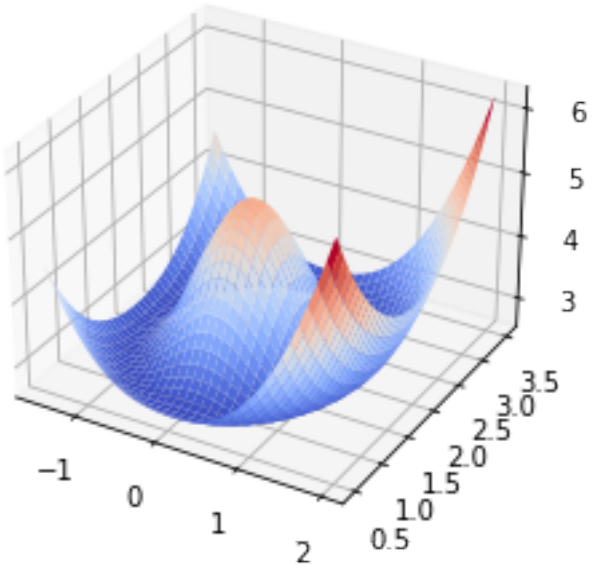
mit dem Definitionsbereich $D = [-1.5,2] \times [0.5,3.5]$. Zunächst benötigen wir einen meshgrid, welcher aus dem Definitionsbereich die Gitterpunkte auswählt, in der die Funktion ausgewertet werden soll. Für die graphische Darstellung gibt es unzählige verschiedene Möglichkeiten. Wir zeigen hier einige Beispiele und verweisen für alles weitere auf die Dokumentation von Matplotlib. Wir betrachten folgendes Minimalbeispiel:

[87]:
```
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

xlist = np.linspace(-1.5, 2.0, 35)
ylist = np.linspace(0.5, 3.5, 40)
X, Y = np.meshgrid(xlist, ylist)
Z = 5*np.exp(-X**2-(Y-2)**2)+X**2+(Y-2)**2

#surface plot, normale Färbung
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, Z)
# Regenbogenfärbung
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, Z, cmap=cm.gist_rainbow)
```

```
# Regenbogenfärbung mit kühleren Farben
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm)

plt.show()
```

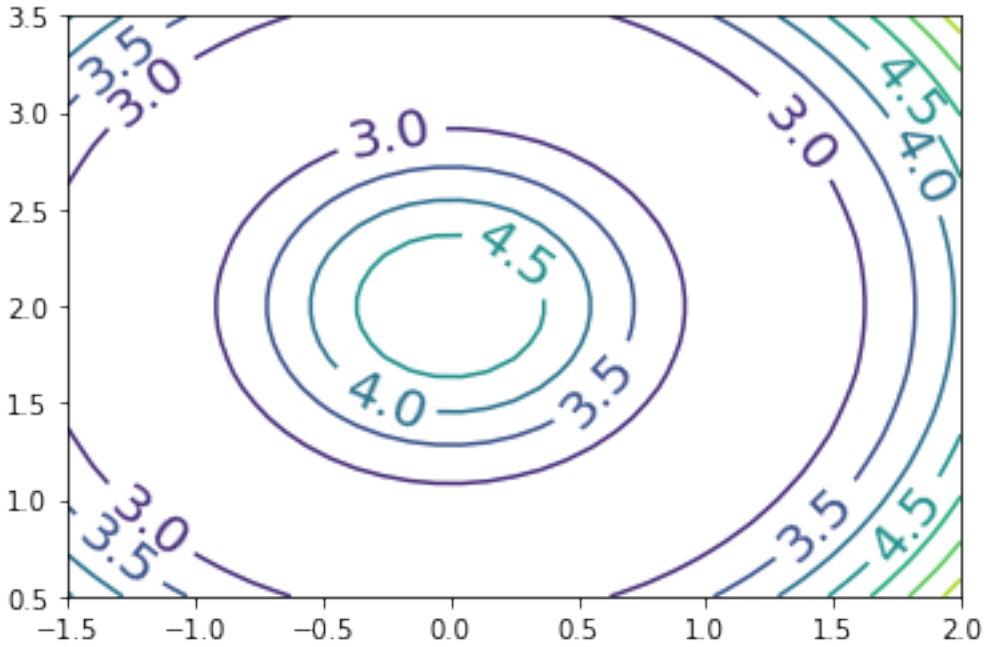


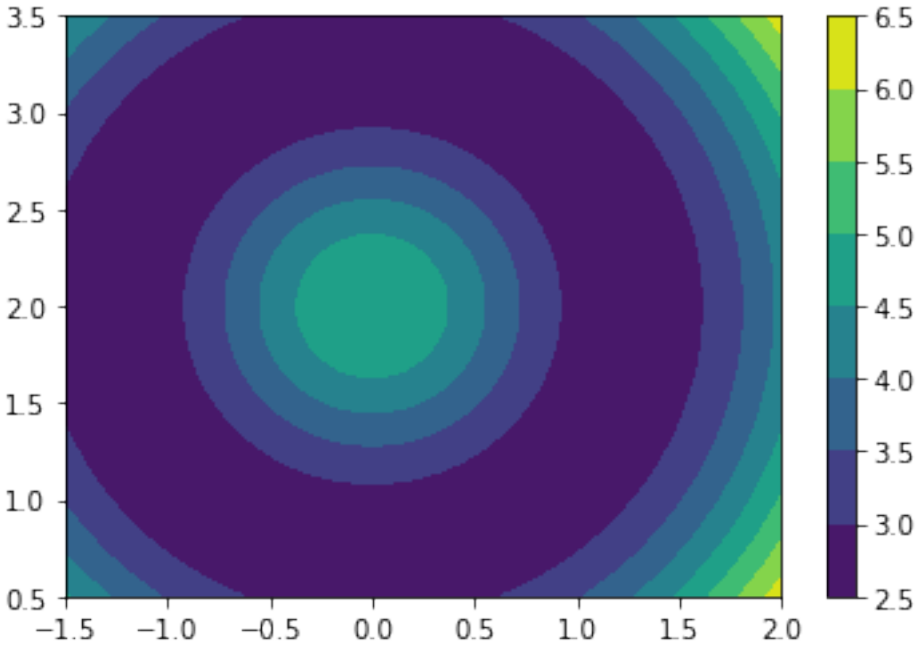




Weitere Auswahlmöglichkeiten für die Einfärbung der Graphik sind unter https://matplotlib.org/devdocs/gallery/color/colormap_reference.html zu finden. Eine weitere graphische Darstellung von $f : D \subset \mathbb{R}^2 \to \mathbb{R}$ ist die Höhenkarte. Auch dafür bietet Matplotlib wieder viele Möglichkeiten. Wir geben hier wieder zwei kleine Beispiele an. Erneut möchten wir zum eigenen Verändern der Parameter anregen.

[88]:
```
xlist = np.linspace(-1.5, 2.0, 35)
ylist = np.linspace(0.5, 3.5, 40)
X, Y = np.meshgrid(xlist, ylist)
Z = 5*np.exp(-X**2-(Y-2)**2)+X**2+(Y-2)**2

#contour plot
plt.figure()
cp = plt.contour(X, Y, Z)
plt.clabel(cp, inline=True, fontsize=20)

plt.figure()
cp = plt.contourf(X, Y, Z)
plt.colorbar(cp)

plt.show()
```





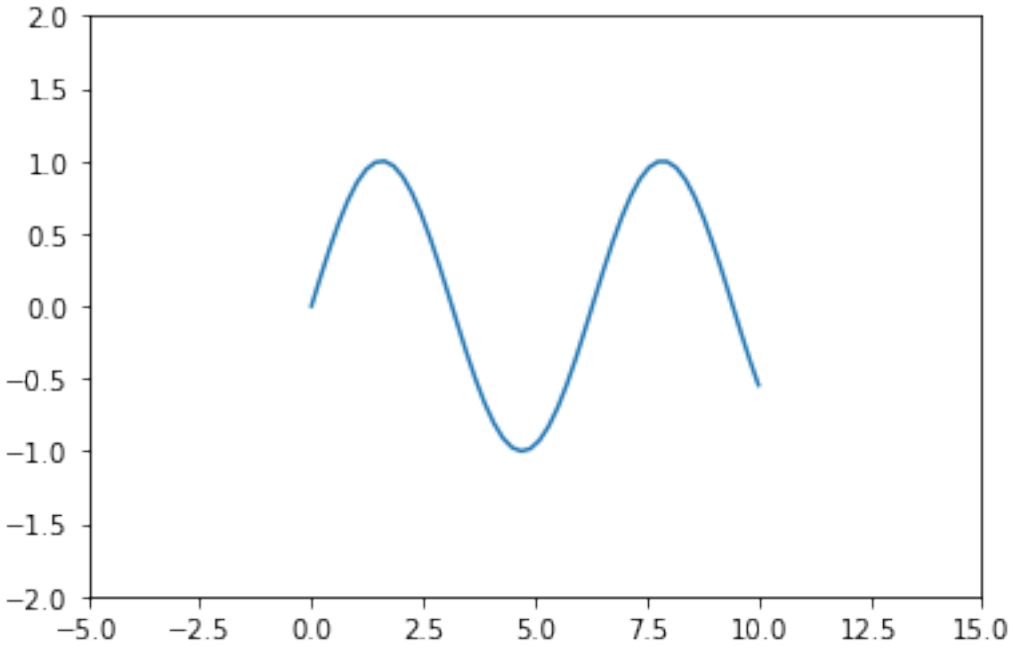
Alle weiteren Optionen für Höhenkarten gibt es unter https://matplotlib.org/3.2.0/api/_as_gen/matplotlib.pyplot.contour.html.

4.2 Graphiken beschriften

Es gibt eine ganze Reihe von Möglichkeiten, um eine erstellte Graphik zu beschriften.

Achsenhandling: Normalerweise wählt Matplotlib die Achsen und deren Skalierung automatisch so, dass die Graphik das Figure-Fenster gerade ausfüllt. Mit der Funktion `axis` können wir aber auch den Wertebereich einer Achse beeinflussen. So zeichnet z.B.

[89]:
```
x = np.linspace(0, 10, 50, endpoint=True)
y = np.sin(x)

plt.plot(x,y)

xmin, xmax, ymin, ymax = -5, 15, -2, 2
plt.axis([xmin, xmax, ymin, ymax])

plt.show()
```



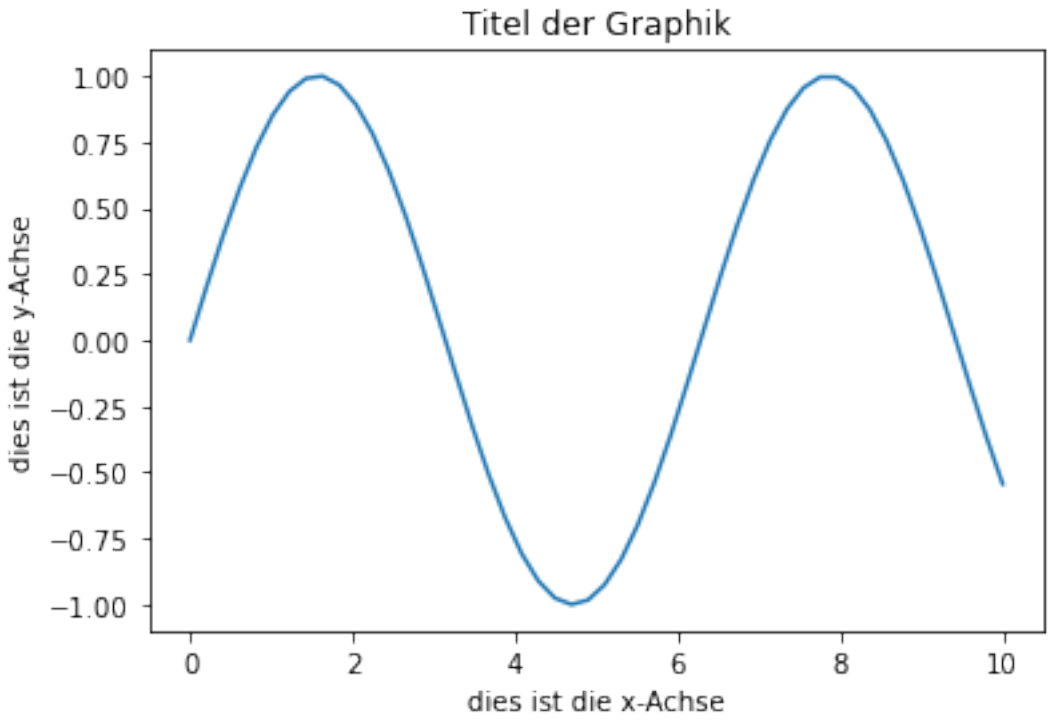
die X-Achse von `xmin` bis `xmax` und die Y-Achse von `ymin` bis `ymax`.

Achsenbeschriftung: Die Achsen werden mit den Kommandos `xlabel`, `ylabel` bzw. `zlabel` beschriftet.

Überschrift: Durch den Matplotlib-Befehl `title` wird das Bild mit einer Überschrift versehen. Insgesamt also:

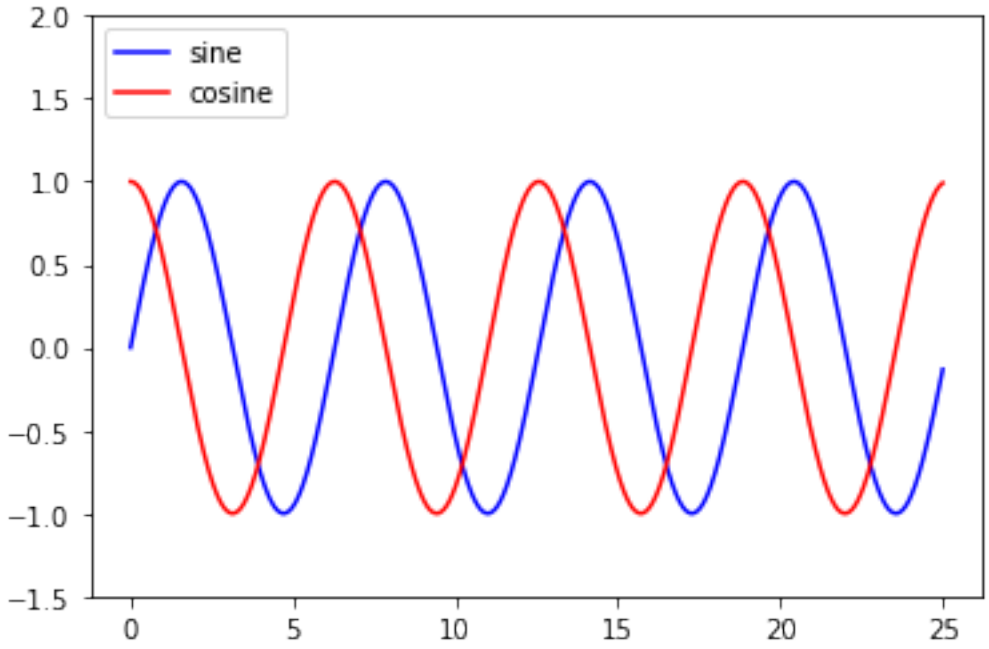
[90]:
```
x = np.linspace(0, 10, 50, endpoint=True)
y = np.sin(x)

plt.plot(x,y)

# Achsenbeschriftung
plt.xlabel('dies ist die x-Achse')
plt.ylabel('dies ist die y-Achse')
# Titel
```

```
plt.title("Titel der Graphik")

plt.show()
```



Legenden: Wir betrachten hierfür wieder ein Beispiel:

[91]:
```
x = np.linspace(0, 25, 1000)
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x, y1, '-b', label='sine')
plt.plot(x, y2, '-r', label='cosine')
plt.legend(loc='upper left')
plt.ylim(-1.5, 2.0)

plt.show()
```



Dabei liefert `plt.ylim(min,max)` einen neuen Bereich in dem die Y-Wert dargestellt werden sollen. Mit `plt.legend` können wir eine Legende einfügen. Dabei bestimmt `loc='string'` die Position. Wenn man sich noch nicht sicher ist, so kann man `loc='best'` setzen. Matplotlib versucht dann, die günstigste Position für die Legende zu finden.

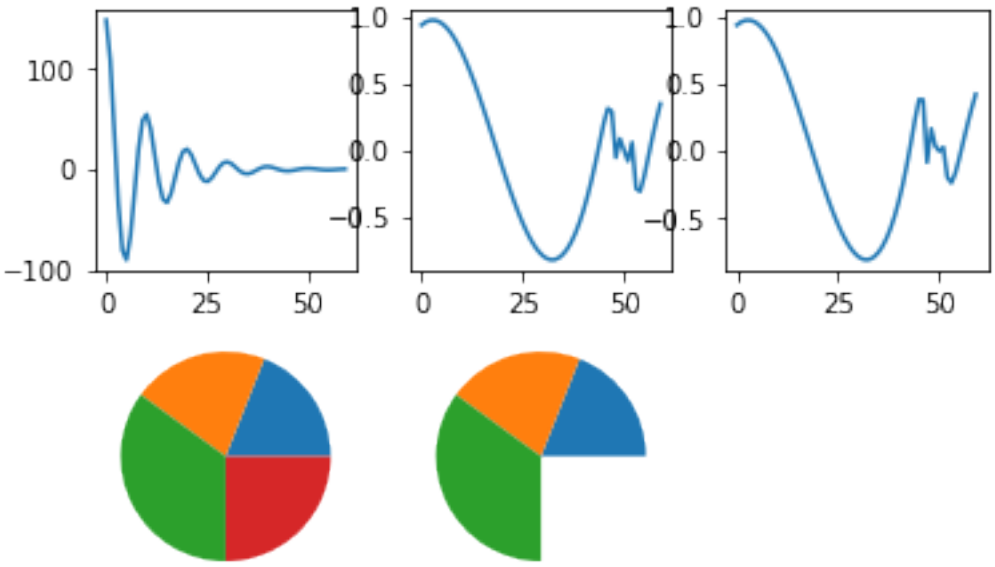
4.3 Unterbilder

Man kann mehrere Bilder in einem Graphik-Fenster darstellen. Dazu wird das Matplotlib-Kommando `subplot(nrows, ncols, plot_number)` verwendet. Dies bedeutet, dass das Graphik-Fenster in insgesamt `nrows·ncols` Unterbilder zerlegt wird und aktuell das `plot_number` Bild angesprochen wird. Hier ein Beispiel:

[92]:
```
t = np.arange(-5.0, 1.0, 0.1)

f = np.exp(-t)*np.cos(2*np.pi*t)
g = np.sin(t)*np.cos(1/t)
h = np.sin(t)*np.cos(1/(t+0.1))

x = np.array([0.19, 0.21, 0.35, 0.25])
u = np.array([0.19, 0.21, 0.35])

plt.subplot(2,3,1)
plt.plot(f)
plt.subplot(2,3,2)
plt.plot(g)
plt.subplot(2,3,3)
plt.plot(h)
plt.subplot(2,3,4)
```

```
plt.pie(x)
plt.subplot(2,3,5)
plt.pie(u)

plt.show()
```

```
/var/folders/z4/vv8gb7j93tn006mx_gs44wdm0000gp/T/ipykernel_69304/3030106703.
 ↪py:1
9: MatplotlibDeprecationWarning: normalize=None does not normalize if the sum␣
 ↪is
less than 1 but this behavior is deprecated since 3.3 until two minor releases
later. After the deprecation period the default value will be normalize=True.␣
 ↪To
prevent normalization pass normalize=False
  plt.pie(u)
```

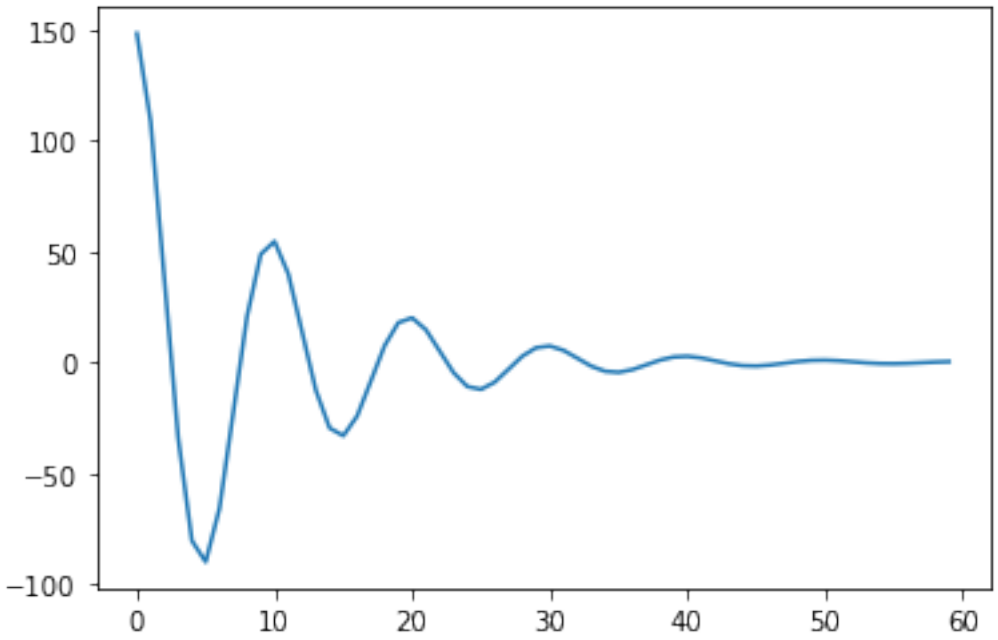


4.4 Graphiken abspeichern

Wir stellen hier zwei Möglichkeiten vor, eine erstellte Graphik zu speichern. Bei der ersten Variante wählt man einfach in dem sich öffnenden Fester das Speichern Symbol aus und kann die Graphik im gewünschten Format, sowie an dem gewünschten Speicherpfad abspeichern. Für die zweite Variante betrachten wir das folgende Beispiel:

[93]:
```
plt.close()


t = np.arange(-5.0, 1.0, 0.1)
f = np.exp(-t)*np.cos(2*np.pi*t)
```

```
plt.plot(f)
plt.savefig('imgs/Beispiel.png')

plt.show()
```



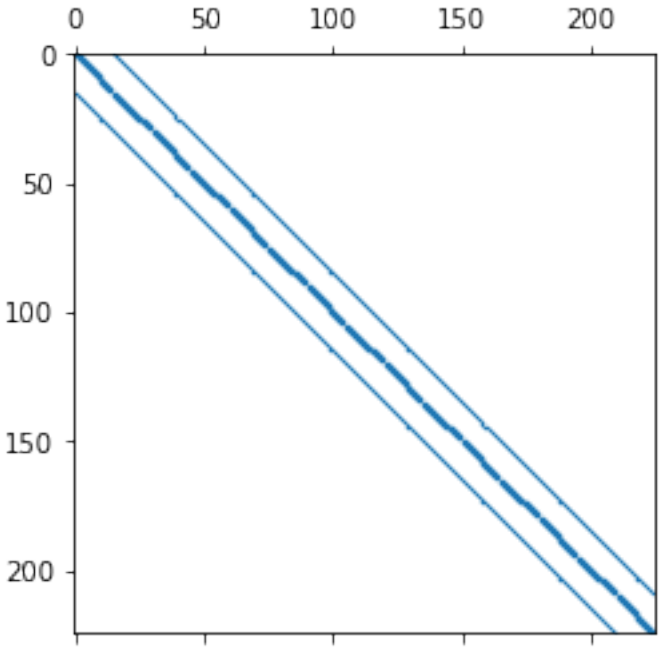
Hier wird die erstellte Graphik automatisch als PNG-Datei in das Verzeichnis dieses Notebooks abgespeichert. Mit `'string.*`, wobei * z.B. jpg, png, pdf usw. sein kann, wird die Graphik in das entsprechende Format gespeichert.

4.5 Struktur einer Matrix

Manchmal kann es sinnvoll sein, insbesondere bei großen Matrizen die Struktur einer Matrix zunächst über eine Graphik zu untersuchen. Wir betrachten folgendes Beispiel:

[94]:
```
m = 15
d = 2

T_1 = -2*d*np.eye(m**d, m**d)

t_2 = np.ones(m-1)
t_2 = np.append(t_2, 0)
t_2 = np.tile(t_2, m**(d-1))
t_2 = np.delete(t_2, -1)
T_2 = np.diag(t_2, 1)

T_3 = np.diag(t_2, -1)

I_1 = np.eye(m**d, m**d, m**(d-1))
I_2 = np.eye(m**d, m**d, -m**(d-1))
```

```
A = T_1 + T_2 + T_3 + I_1 + I_2


plt.spy(A, markersize=0.5)
plt.show()
```



5 Weiterführendes

5.1 Klassen

Klassen bieten die Grundstruktur der objektorientierten Programmierung. Python-Klassen bieten alle Standardeigenschaften der objektorientierten Programmierung: Die Vererbung bietet mehrere Basisklassen, in einer abgeleiteten Klassen kann jede Methode der Basisklasse überschrieben werden und jede Methode der Basisklasse(n) kann in der abgeleiteten Klasse mit demselben Namen aufgerufen werden. Ein Objekt kann beliebig viele Mengen und Arten von Daten haben. Da dieses Thema sehr umfangreich und komplex ist, werden wir uns hier auf die grundlegende Syntax beschränken. Nach dem Klassenname muss der folgenden Block eingerückt werden und Klassennamen beginnen in der Regel mit einem Großbuchstaben. Grundlegende Syntax:

```
class <Classname>:
    <Anweisungen>
```

Dabei können wir auch innerhalb einer Klassendefinition eine Funktion definieren, mit welcher die Klasse ausgestattet ist. Jeder Instanz der Klasse steht diese Funktion zur Verfügung und sie kann mittels `<Classname>.<function>()` aufgerufen werden. In unserem Beispiel wird das die Funktion `abs()` sein, welche den Absolutbetrag einer komplexen Zahl ausgibt. Ein anderes Beispiel war die `list.append()`-Funktion aus dem Listenabschnitt.

Wir wollen uns nun ein einfaches Beispiel ansehen und die `__init__`-Methode kennenlernen. Mit dieser Methode muss jede Klasse ausgestattet sein, von der wir eine Instanz erzeugen wollen.

[95]:
```
class Complex():
    def __init__(self, realpart, impart):
        self.r = realpart
        self.i = impart
    def abs(self):
        return(np.sqrt(self.r**2 + self.i**2))
```

Benutzen können wir die Klasse, welche komplexe Zahlen repräsentiert, nun wie folgt:

[96]:
```
x = Complex(1,2)
# Realteil ausgeben
print(x.r)
# Imagintärteil ausgeben
print(x.i)
# Ausgabe des Betrages
print(x.abs())
```

```
1
2
2.23606797749979
```

Alles weitere zu Klassen unter: https://docs.python.org/3/tutorial/classes.html.

5.2 Anmerkungen, Fragen, Sonstiges

Anmerkungen, Fragen oder Fehler gerne an: agvolkwein.oppy@uni-konstanz.de.