Grundkurs Python 3

Release 0.1.2d

Aktualisiert am 02.12.2018

Bernhard Grotz

http://www.grund-wissen.de

Die Quelldateien dieses Buchs wurden unter Linux mittels Vim und Sphinx, die enthaltenen Graphiken mittels Inkscape erstellt. Der Quellcode sowie die Original-Graphiken können über die Projektseite heruntergeladen werden:

http://www.grund-wissen.de

Bei Fragen, Anmerkungen und Verbesserungsvorschlägen bittet der Autor um eine kurze Email an folgende Adresse:

info@grund-wissen.de

Augsburg, den 2. Dezember 2018.

Bernhard Grotz

Inhaltsverzeichnis

Installation

Installation von Python3

Bei neuen Linux-Versionen ist Python in den Versionen 2.7 und 3.5 bereits vorinstalliert. Auf älteren Systemen kann es hingegen notwendig sein, die aktuelle (und sehr empfehlenswerte) Version 3 von Python nachträglich zu installieren. Hierzu sollten folgende Pakete mittels `apt` installiert werden:

```
sudo aptitude install python3 python3-doc python3-pip
```

Das zuletzt genannte Programm `pip3` erlaubt es, zusätzliche Erweiterungen (sofern diese nicht auch über `apt` installierbar sind) mittels folgender Syntax zu installieren:

```
sudo pip3 paketname
```

Dabei werden automatisch alle bekannten Python-Repositories durchsucht und die aktuelle Version installiert. Mit der Option `-U` („update“) wird eine eventuell bereits vorhandene, nicht mehr aktuelle Version eines Pakets durch die neueste Version ersetzt. Beispielsweise kann so mittels `pip3 -U Sphinx` die neueste Version des Python-Dokumentationssystems Sphinx installiert werden. Alternativ kann auch in den gewöhnlichen Linux-Paketquellen mittels `apt` nach einem entsprechenden Python-Paket gesucht beziehungsweise dieses installiert werden.

Installation von Ipython3

Anstelle des „normalen“ Python-Interpreters, der sich durch Aufruf von `python3` ohne weitere Argumente starten lässt, sollte bevorzugt `ipython3` verwendet werden. Neben einer automatischen Vervollständigung von Modul-, Klassen- und Funktionsnamen bei Drücken der `Tab`-Taste bietet Ipython eine interaktive Syntax-Hilfe und weitere hilfreiche Funktionen.

Folgende Pakete sollten für Ipython3 installiert werden:

```
sudo aptitude install ipython3 ipython3-qtconsole ipython3-notebook python3-tk
```

Ipython kann als Shell-Version anschließend mittels `ipython3`, die graphische Oberfläche mittels `ipython3 qtconsole` gestartet werden.

Virtuelle Umgebungen

Python ermöglicht es mittels einer so genannten virtuellen Umgebung, die Entwicklung eines Python-Programms gezielt auf eine bestimmte Python-Version und eine bestimmte Auswahl an installierten Paketen abzustimmen.

Zunächst muss hierzu das Paket `virtualenv` installiert werden:

```
sudo aptitude install python3-pip python3-dev build-essential
pip3 install --upgrade virtualenv
```

Anschließend kann im Basis-Verzeichnis eines Projekts folgendermaßen eine neue virtuelle Arbeitsumgebung erstellt werden:

```
# Virtuelle Umgebung im Unterverzeichnis "env" erstellen:
virtualenv -p python3 env

# Oder:

virtualenv -p python3 --no-site-packages env
```

Der Unterschied zwischen diesen beiden Varianten liegt darin, dass die erste Symlinks auf bereits installierte Python-Pakete setzt (platzsparend, aufbauend auf dem bestehenden System), die zweite hingegen eine vollständig neue Umgebung schafft (nützlich, wenn ein installiertes Paket für ein konkretes Projekt modifiziert werden soll, beispielsweise Sphinx).

Die virtuelle Umgebung kann dann aus dem Projektverzeichnis heraus folgendermaßen aktiviert werden:

```
# Virtuelle Umgebung aktivieren:
source env/bin/activate
```

Alle Paket-Installationen, die bei einer aktiven virtuellen Umgebung vorgenommen werden, haben nur Auswirkung auf diese Umgebung; zunächst ist überhaupt kein Zusatzpaket installiert. Mittels `pip3 install paketname` können wie gewohnt Pakete installiert werden:

```
# Python-Paket in der virtuellen Umgebung installieren:
pip3 install Sphinx
```

Gegebenenfalls muss, beispielsweise bei der lokalen Installation von Sphinx, anschließend `hash -r` eingegeben werden, damit der „Suchpfad“ aktualisiert und die Python-Programme beim Aufruf auch lokal gefunden werden.

Durch Eingabe von `deactivate` in dem Shell-Fenster wird die virtuelle Umgebung wieder beendet:

```
# Virtuelle Umgebung beenden:
deactivate
```

Links

Virtualenv-Tutorials:

- https://realpython.com/python-virtual-environments-a-primer/
- https://www.simononsoftware.com/virtualenv-tutorial-part-2/

Einführung

Python ist eine Programmiersprache, die einfach zu erlernen ist und ein schnelles Entwickeln von Programmen ermöglicht. Aufgrund der verhältnismäßig hohen Lesbarkeit und einiger hilfreicher Mathematik-Module wird Python auch im akademischen und schulischen Bereich häufig verwendet.

Interaktiver Modus

Um Python im interaktiven Modus zu starten, ruft man den Interpreter in einer Shell ohne weitere Parameter auf:

```
python3
```

Nach dem Eingabe-Prompt >>> kann beliebiger Python-Code eingegeben werden. Drückt man die Enter-Taste, so wird dieser unmittelbar ausgeführt. So lässt sich der Python-Interpreter beispielsweise als besserer Taschenrechner benutzen:[1]

```
>>> 5 + 3        # Addition
8
>>> 7 * 21       # Multiplikation
147
>>> 15 ** 2      # Potenz
225
>>> 14 / 80      # Division
0.175
```

Möchte man ausführlicher im interaktiven Modus arbeiten, so lohnt es sich, *Ipython zu installatieren* und `ipython3` als Shell-Version beziehungsweise `ipython3 qtconsole` als GUI-Version aufzurufen. Beide Varianten von Ipython bieten Vervollständigungen der Eingabe durch Drücken der Tab-Taste, die GUI-Version bietet zudem Syntax-Highlighting und ein automatisches Einblenden von Funktionsbeschreibungen („*Docstrings*“).

[1] Neben der gewöhnlichen Division mit `/` kann auch mit `//` eine Ganzzahl-Division durchgeführt werden. Bei einer solchen Division wird der Divisionsrest weggelassen und stattdessen die nächst kleinere ganze Zahl als Ergebnis zurückkgegeben; beispielsweise ergibt `17 // 5` den Wert `3`. Der Divisionsrest kann mit dem Modulo-Operator `%` bestimmt werden; beispielsweise ergibt `17 % 5` den Wert `2`. Beide Werte können auf einmal durch die Funktion `divmod()` ausgegeben werden; beispielsweise ergibt `divmod(17,5)` das Zahlenpaar (3,2).

Wurzeln können entweder als Potenzen mit einer rationalen Zahl als Exponent oder mittels der Funktion `math.sqrt()` aus dem `math`-Modul berechnet werden.

Interne Hilfe

Jeder Python-Interpreter bietet dem Nutzer im interaktiven Modus die Möglichkeit, weitere Informationen oder Dokumentationen zu Python-Objekten (Funktionen, Operatoren usw.) anzuzeigen. Um Hilfe zu einem beliebigen Python-Objekt zu erhalten, kann die Funktion `help()` genutzt werden:

```
# Hilfe zur print()-Funktion anzeigen:
help(print)
```

Ebenso können Variablen an die Help-Funktion übergeben werden, um Hilfe zum jeweiligen Objekttyp zu erhalten. Mit `type(object_name)` kann der Typ eines Objekts, mit `id(variable_name)` zusätzlich die aktuelle Speicheradresse des Objekts angezeigt werden.

Python-Skripte

Python-Code, der bei der interaktiven Benutzung des Interpreters eingegeben werden kann, kann in gleicher Form ebenso in Textdateien („Skripte“), üblicherweise mit der Endung `.py`, geschrieben werden. Derartige Dateien können entweder in einer interaktiven Python-Sitzung mittels `execfile("skriptname.py")` oder folgendermaßen in einem Shellfenster aufgerufen werden:

```
python3 skriptname.py
```

Beim Aufruf eines Skripts wandelt Python den Quellcode in so genannten „Bytecode“ um und führt diesen aus. Wird eine Quellcode-Datei anderen Dateien importiert, so legt Python automatisch eine zusätzliche Datei `skriptname.pyc` im gleichen Verzeichnis an.

Es ist auch möglich, eine Skriptdatei direkt als ausführbare Datei aufzurufen. Dazu fügt man zunächst folgende Zeilen am Anfang der Skriptdatei ein:

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

Die erste Zeile wird „Shebang“ genannt und gibt den Pfad zum Python-Interpreter an, der beim Aufruf des Skripts geladen werden soll; die zweite Zeile gibt an, welcher Zeichensatz in der Datei verwendet wird (`utf-8` ist Standard unter Linux).

Mit der obigen Ergänzung kann die Skriptdatei dann mittels chmod ausführbar gemacht werden:

```
chmod +x skriptname.py
```

Das Skript kann damit mittels `./skriptname.py` aus dem aktuellen Pfad heraus oder allgemein mittels `pfad-zum-skript/skriptname.py` aufgerufen werden. Soll es benutzerweit aufrufbar sein, so empfiehlt es sich, einen Symlink zum Skript im Verzeichnis

`~/bin` zu erstellen und dieses durch folgenden Eintrag in der `~/.bashrc` zum Systempfad hinzuzufügen:[2]

```
if [ -d "$HOME/bin" ] ; then
    PATH="$HOME/bin:$PATH"
    export PATH;
fi
```

Das Schreiben von Code-Dateien ist in Python gegenüber der interaktiven Benutzung des Interpreters unter anderem deshalb von Vorteil, da Python beispielsweise bei der Definition von *Funktionen* und *Kontrollstrukturen* Einrückungen statt Klammern zur Gliederung des Quellcodes verwendet.

Gute Texteditoren machen den Umgang mit Einrückungen einfach und bieten obendrein Vorteile wie Syntax-Highlighting, Eingabe-Vervollständigungen, Snippets, usw. Bei Verwendung von Vim und des Vicle-Plugins ist es zudem auch während des Schreibens der Textdateien möglich, einzelne Code-Zeilen oder auch ganze Code-Blöcke an eine laufende Interpreter-Sitzung zu senden; so können die Vorteile des Interpreters (beispielsweise Ausgabe von Variablenwerten und Zwischenergebnissen) und des Texteditors kombiniert werden.

Umfangreichere Skripte sollten zur besseren Lesbarkeit mit Kommentaren versehen werden. Kommentare werden durch das Raute-Symbol `#` eingeleitet und gehen bis zum Ende der Zeile.

Variablen

Eine wichtige Eigenschaft von Computer-Programmen ist, dass sie nahezu beliebig viele Werte und Zeichen in entsprechenden Platzhaltern („Variablen“) speichern und verarbeiten können. Auf so gespeicherte Werte kann man im Verlauf des Programms wieder zugreifen und/oder den Variablen neue Werte zuweisen.

In Python können Variablennamen aus Groß- und Kleinbuchstaben, Ziffern und dem Unterstrich-Zeichen bestehen, wobei sie nicht mit einer Ziffer beginnen dürfen. Bei der Benennung von Variablen ist außerdem auf Groß- und Kleinschreibung zu achten, beispielsweise bezeichnen `var_1` und `Var_1` zwei unterschiedliche Variablen. Zudem dürfen keine von der Programmiersprache reservierten Wörter als Variablennamen verwendet werden (beispielsweise `and, or, is, type, key` usw.)

Ein Wert kann in einer Variablen mittels des Zuweisungs-Operators `=` gespeichert werden:

```
var_1 = "Hallo!"
var_2 = 42
```

In Python dient das =-Zeichen somit ausschließlich der Zuweisung von Werten; für einen Werte-Vergleich muss hingegen das doppelte Istgleich-Zeichen == verwendet werden.

[2] Dieser Trick ist im Shell-Skripting-Tutorial näher beschrieben.

Im interaktiven Modus wird der Wert einer Variablen angezeigt, indem man deren Namen in einer neuen Zeile eingibt und `Enter` drückt. In Skripten werden die Werte von Variablen oftmals mittels der Funktion `print()` angezeigt:

```
print(var_1)
print(var_2)
```

Bei der Verwendung von `print()` werden dabei die Variable als *Zeichenkette* ausgegeben.

Variablen werden in Python dynamisch typisiert, das heißt in der gleichen Variablen können im Verlauf des Programms verschiedene *Datentypen* zugewiesen werden.

Operatoren

Bei der Auswertung einzelner mathematischer Ausdrücke gilt wie üblich „Punkt vor Strich“. Um eine andere Auswertungsreihenfolge zu bewirken, können einzelne Ausdrücke, wie in der Mathematik üblich, durch runde Klammern zusammengefasst werden.

Für die in Python üblichen Operatoren ist eine allgemein gültige „Rangfolge“ für die Auswertungsreihenfolge festgelegt. In der folgenden Tabelle sind die Operatoren mit der höchsten Priorität stehen oben, gleichberechtigte Operatoren (die von links nach rechts ausgewertet werden) stehen in der gleichen Zeile.[3]

Operator	Bedeutung
`()`	Gruppierung
`x[]`, `x.attribute`	Listenzugriff (siehe *Listen*), Objekteigenschaft
`**`	Potenz
`+` und `-`	Positives beziehungsweise negatives Vorzeichen einer Zahl
`*`, `/`, `//`, `%`	Multiplikation, Division, Ganzzahl-Division, Rest (Modulo)
`==`, `<=`, `<`, `!=`, `>=`, `>`, `is`, `is not`, `in`, `not in`	Wertevergleich (gleich, kleiner als oder gleich, kleiner als, ungleich, größer als oder gleich, größer als), Identitätsvergleich, Test auf Mengenzugehörigkeit
`not`	Logisches Nicht
`and`	Logisches Und
`or`	Logisches Oder

In Python muss zwischen dem Wertevergleich `==` und der Objekt-Identität `is` unterschieden werden. Beispielsweise liefert `3/2 == 1.5` das Ergebnis `True`, da die numerischen Werte übereinstimmen; hingegen liefert `3/2 is 1.5` das Ergebnis `False`, da es sich einmal um einen mathematischen Ausdruck und einmal um eine Zahl handelt.

[3] Eine Ausnahme bildet der Potenz-Operator `**`: Werden mehrere Potenzen in einem Ausdruck kombiniert, so werden diese von rechts nach links ausgewertet. Somit gilt `2 ** 2 ** 3 == 2 ** 8 == 256`. Für eine andere Auswertungsreihenfolge muss `(2 ** 2) ** 3 == 4 ** 3 == 64` geschrieben werden.

Kombinierte Zuweisungsoperatoren

Neben dem gewöhnlichen Zuweisungsoperator `=` gibt es in Python weitere, kombinierte Zuweisungsoperatoren. Mit diesen wird ein mathematischer Operator mit einer Zuweisung verbunden; die Variable wird dabei also um den jeweiligen Wert verändert.

`+=`	Erhöhung der links stehenden Variable um Wert auf der rechten Seite
`-=`	Erniedrigung der links stehenden Variable um Wert auf der rechten Seite
`*=`	Multiplikation der links stehenden Variable mit Wert auf der rechten Seite
`/=`	Division der links stehenden Variable durch Wert auf der rechten Seite
`//=`	Ganzzahlige Division der links stehenden Variable durch Wert auf der rechten Seite
`//=`	Rest bei ganzzahliger Division der links stehenden Variable durch Wert auf der rechten Seite
`**=`	Potenzieren einer Variable mit Wert auf der rechten Seite

Beispielsweise kann auf diese Weise mit `x **= 2` der aktuelle Wert der Variablen `x` quadriert werden. Für Zeichenketten existieren nach dem gleichen Prinzip die Operatoren `+=` und `*=`, die zum String auf der linken Seite einen weiteren String anhängen bzw. den String auf der linken Seite mehrfach wiederholt aneinander reihen.

Kombinierte Vergleichsoperatoren

Eine weitere Besonderheit in Python liegt darin, dass mehrere Vergleichsoperatoren unmittelbar miteinander kombiniert werden können; beispielsweise kann wie in der Mathematik `1 < 2 < 3` geschrieben werden. Die einzelnen Teilausdrücke muss man sich dabei mit einem `and`-Operator verbunden denken, denn der Ausdruck ist genau dann wahr, wenn `1 < 2 and 2 < 3` gilt.

Die „Verkettungsregel“ gilt für alle Vergleichsoperatoren, auch wenn das Ergebnis nicht immer mit der eigenen Erwartung übereinstimmen muss. Beispielsweise könnte man im Fall `1 == 2 < 3` das Ergebnis `True` erwarten, wenn man sich die gleichwertigen Operatoren von links nach rechts ausgewertet denkt, denn `1 == 2` ist `False` und zudem ist `False < 3`. Die Aussage liefert in Python jedoch `False` als Ergebnis, denn sie wird als `1 == 2 and 2 < 3` interpretiert, und zwei mit `and` verknüpfte Aussagen ergeben nur dann ein wahres Ergebnis, wenn beide Aussagen wahr sind.

Im Zweifelsfall können die einzelnen Teilaussagen jederzeit mit Hilfe von runden Klammern gruppiert werden, um eine ungewollte Auswertungsreihenfolge zu vermeiden.

Bedingte Wertzuweisung

In Programmiersprachen wie `C` gibt es ein Sprachkonstrukt, dass einem „ternären“ Operator entspricht, also ein Operator mit drei notwendigen Argumenten. In `C` lautet dieser

Operator etwa `x = condition ? a : b`, was bedeutet, dass der Variablen x der Wert `a` zugewiesen wird, wenn die Bedingung `condition` wahr ist; andernfalls wird der Variablen `x` der Wert `b` zugewiesen.

In Python lautet das entsprechende Sprachkonstrukt folgendermaßen:

```
x = a if condition else b
```

Auch hier wird zunächst die Bedingung `condition` geprüft. Wenn diese den Wert `True` ergibt, so wird der Variablen `x` der Wert `a` zugewiesen, andernfalls der Wert `b`.

Datentypen

Im folgenden Abschnitt werden die zum Python-Standard gehörenden Datentypen kurz vorgestellt. In Skripten oder im interaktiven Modus kann der Datentyp eines Objekts oder einer Variable jederzeit mittels `type(variable)` angezeigt werden.

`None` – Der Nichts-Typ

Durch den Datentyp `None` wird in Python beispielsweise symbolisiert, dass eine Variable keinen Wert beinhaltet. Dies ist beispielsweise sinnvoll, wenn man eine Variable definieren, ihr aber erst späteren einen konkreten Wert zuweisen will; ein anderer Anwendungsfall wäre die Rückgabe eines Ergebniswerts bei einer erfolglosen Suche.

Um einer Variablen den Wert `None` zuzuweisen, gibt man folgendes ein:

```
var_1 = None

# Test:

print(var_1)
# Ergebnis: None
```

`None` ist ein *Singleton*, es gibt also stets nur eine Instanz dieses Typs; `None` kann somit stets wie eine Konstante verwendet werden. Ebenso kann mittels des `None`-Typs abgefragt werden, ob eine Variable einen Wert beinhaltet oder nicht. Eine solche Abfrage kann prinzipiell so aussehen:

```
if var_1 is None:
    print('var_1 has no value.')
else:
    print('the value of var_1 is ' var_1)
```

Mittels des Schlüsselworts `is` wird im obigen Beispiel überprüft, ob `var_1` eine Instanz des Typs `None` ist. Durch `if var_1 is not None` kann sichergestellt werden, dass ein Code-Teil nur dann ausgeführt wird, wenn der Variablen `var_1` bereits ein Wert zugewiesen wurde.

Numerische Datentypen

`True` und `False` – Boolesche Werte

Eine boolesche Variable kann nur `True` (wahr) oder `False` (falsch) als Werte annehmen. Von Python wird `True` als `1` beziehungsweise `False` als `0` interpretiert, so dass sich theoretisch auch mit Variablen des Datentyps `bool` rechnen lässt (beispielsweise ergibt `True + True` den Wert `2`).

Der boolesche Wahrheitswert eines beliebigen Ausdrucks kann mittels der Standard-Funktion *bool()* ermittelt werden, beispielsweise liefert `bool(1)` den Wert `True`.

`int` – Ganze Zahlen

Ganzzahlige Werte werden in Python durch den Datentyp `int` repräsentiert.

Um die Anzahl an Ziffern einer `int`-Zahl zu bestimmen, kann diese mittels `str()` in eine Zeichenkette umgewandelt werden; anschließend kann die Länge dieser Zeichenkette mittels `len()` bestimmt werden:

```
num_1 = 58316

# Zahl in Zeichenkette umwandeln:

str(num_1)
# Ergebnis: '58316'

# Anzahl der Ziffern der Zahl ausgeben:

len(str(num_1))
# Ergebnis: 5
```

Wird im umgekehrten Fall eine Zahl beispielsweise mittels der Funktion `input()` eingelesen, so liegt sie als Zeichenkette vor; mittels `int()` ist dann eine Konvertierung in eine gewöhnliche Zahl möglich.

Bisweilen werden Zahlen auch in einer binären, oktalen oder hexadezimalen Darstellung verwendet. Um eine dezimale `int`-Zahl mit einer anderen Zahlenbasis (`2`, `8` oder `16`) darzustellen, gibt es folgende Funktion:

```
num_1 = 78829

bin(num_1)
# Ergebnis: '0b10011001111101101'

oct(num_1)
# Ergebnis: '0o231755'

```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
hex(num_1)
# Ergebnis: '0x133ed'
```

Das Ergebnis sind jeweils Zeichenketten, die mit `0b` (binär), `0o` (oktal) oder `0x` (hexadezimal) beginnen. Um eine derartige Zeichenkette wieder in eine gewöhnliche `int`-Zahl zu konvertieren, kann man die `int()`-Funktion nutzen, wobei die ursprüngliche Zahlenbasis als zweites Argument angegeben werden muss:

```
# Binärzahl in Dezimalzahl umwandeln:

int('0b10011001111101101', base=2)
# Ergebnis: 78829
```

Um die größte beziehungsweise kleinste mindestens zweier Zahlen (`int` oder `float`) zu bestimmen, können die Funktionen `min()` oder `max()` genutzt werden:

```
min(-5, 17842, 30911, -428)
# Ergebnis: -428

max(-5, 17842, 30911, -428)
# Ergebnis: 30911
```

Der Absolutwert einer `int` oder `float`-Zahl kann mittels der Standardfunktion *abs(number)* ausgegeben werden.

`float` – Gleitkommazahlen

Zahlen mit Nachkommastellen werden in Python durch den Datentyp `float` repräsentiert. Die Nachkommastellen werden dabei – wie im englischen Sprachraum üblich – nicht durch ein Komma, sondern durch einen Punkt `.` von dem ganzzahligen Anteil getrennt. Zudem ist es möglich, sehr große oder sehr kleine `float`-Zahlen mittels `e` oder `E` in Exponential-Schreibweise anzugeben. Die Zahl hinter dem `e` gibt dabei an, um wie viele Stellen der Dezimalpunkt innerhalb der Zahl verschoben wird.

```
4.3e5 == 430000
# Ergebnis: True

7.92e-5 == 0.0000792
# Ergebnis: True
```

Um eine `float`-Zahl auf n Nachkomma-Stellen zu runden, kann die Funktion `round(float_num, n)` genutzt werden. Wird das Argument `n` weggelassen, wird auf die nächste ganze Zahl gerundet. Eine Gleitkommazahl `float_1` kann ebenso mittels `int(float_1)` in eine ganze Zahl umgewandelt werden; dabei werden jedoch eventuell vorhandene Nachkommastellen abgeschnitten, es also stets die betragsmäßig nächst kleinere ganze Zahl als Ergebnis zurück gegeben.

`complex` – Komplexe Zahlen

Komplexe Zahlen bestehen aus einem Realteil und einem Imaginärteil. Der Imaginärteil besteht aus einer reellen Zahl, die mit der imaginären Einheit `j` multipliziert wird.[1]

Um eine komplexe Zahl zu definieren, gibt man in Python etwa folgendes ein:

```
z_1 = 4 + 3j
z_2 = 5.8 + 1.5j
```

Für das Arbeiten mit komplexen Zahlen kann das `cmath`-Modul aus der Standardbibliothek genutzt werden.

`str` – Zeichenketten

Zeichenketten („Strings“) sind eine Folge von Zeichen, die wahlweise in einfachen oder doppelten Anführungszeichen geschrieben werden.[2] Nahezu jedes Python-Objekt kann mittels `str(object)` in eine Zeichenkette umgewandelt werden, um beispielsweise eine druckbare Darstellung mittels `print()` zu ermöglichen.

```
string_1 = 'Hallo'
string_2 = 'Welt!'
string_3 = str(539)     # Ergebnis: '539'
```

Zeichenketten können mittels `+` miteinander kombiniert werden. Möchte man eine Zeichenkette beliebiger Länge in mehrfacher Wiederholung, so kann diese mittels `*` und einer ganzzahligen Zahl vervielfacht werden. Da Zeichenketten in Python (wie *Tupel*) unveränderbar sind, wird bei den obigen Beispielen stets eine neue Zeichenkette erzeugt. Die ursprüngliche Zeichenkette bleibt jeweils unverändert:

```
string_1 + ' ' + string_2
# Ergebnis: 'Hallo Welt!'

string_1 * 3
# Ergebnis: 'HalloHalloHallo'
```

Die Länge einer Zeichenkette kann mittels `len()` bestimmt werden:

```
len('Hallo Welt')
# Ergebnis: 10
```

[1] In der Mathematik wird die imaginäre Einheit meist mit i bezeichnet, in der Elektrotechnik wird hingegen oft j verwendet. In Python kann sowohl `j` als auch `J` als Symbol für die imaginäre Einheit geschrieben werden.

[2] Python behandelt einfache und doppelte Anführungszeichen gleichwertig, anders als beispielsweise die Linux-Shell. Innerhalb eines Strings, der in einfache Anführungszeichen gesetzt wird, können doppelte Anführungszeichen vorkommen und umgekehrt.

Sollen einfache Anführungszeichen in einem String vorkommen, der ebenfalls durch einfache Anführungszeichen begrenzt ist, so muss vor die inneren Anführungszeichen jeweils ein Backslash (\) als Escape-Sequenz gesetzt werden.

Zur besseren Lesbarkeit sollten Code-Zeilen allgemein nicht mehr als 80 Zeichen lang sein. Lange Zeichenketten können allerdings in der nächsten Zeile fortgesetzt werden, wenn die vorangehende Zeile mit einem einzelnen Backslash \ als Zeile-Fortsetzungs-Zeichen abgeschlossen wird:

```
long_string = 'Das ist eine lange Zeichenkette, die für eine bessere \
              Lesbarkeit über zwei Zeilen verteilt geschrieben wird.'
```

Durch den Backslash werden also beide Zeilen zu einer logischen Einheit verbunden; hinter dem Backslash darf allerdings kein Kommentarzeichen stehen.

Mehrzeilige Zeichenketten können ebenso in dreifache Anführungszeichen gesetzt werden. Solche „Docstrings“ werden beispielsweise verwendet, um längere Code-Abschnitte, Funktionen, Klassen oder Module zu dokumentieren, denn sie bleiben vom Interpreter unbeachtet. Beim Schreiben von Docstrings sollten die offiziellen Empfehlungen beachtet werden.

Zeichenketten können allgemein folgende Sonderzeichen beinhalten:

Zeichen	Bedeutung
`\t`	Tabulator
`\n`	Newline (Zeilenumbruch)
`\r`	Carriage Return
`\\`	Backslash
`\'`	Einfaches Anführungszeichen
`\"`	Doppeltes Anführungszeichen
`\xnn`	Sonderzeichen (*ASCII*), repräsentiert durch eine zweistellige Hexadezimalzahl, beispielsweise `\xe4`
`\unnnn`	Sonderzeichen (16-bit-Unicode), repräsentiert durch eine vierstellige Hexadezimalzahl, beispielsweise `\u7fe2`

Möchte man das Interpretieren der obigen Sonderzeichen unterbinden, kann dies durch ein vorangestelltes `r` („raw“) geschehen; beispielsweise wird in `r'a\tb'` das `\t` nicht als Tabulator-Zeichen interpretiert.

Indizierung von Zeichenketten

Auf die einzelnen Zeichen einer Zeichenkette kann mittels des Index-Operators `[ ]` zugegriffen werden. Dabei wird das erste Zeichen, wie in der Programmiersprache `C` üblich, mit `0` indiziert. Auf das letzte Element eines n Zeichen langen Strings kann entsprechend mit dem Index `n-1`, oder in Kurzschreibweise mit dem Index `-1` zugegriffen werden. Ein größerer Index als `n-1` löst einen Fehler (`IndexError`) aus.

```
example = 'Hallo Welt'

# Erstes und zweites Zeichen:

example[0]
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
# Ergebnis: 'H'

example[1]
# Ergebnis: 'a'

# Vorletztes und letztes Zeichen:

example[-2]
# Ergebnis: 'l'

example[-1]
# Ergebnis: 't'
```

Der Index-Operator kann ebenso genutzt werden, um Bereiche („Slices“) einer Zeichenkette auszugeben. Hierzu werden in den eckigen Klammern zwei Index-Zahlen `n_1` und `n_2` durch einen Doppelpunkt getrennt angegeben. Es muss dabei allerdings beachtet werden, dass in Python bei Bereichsangaben die obere Grenze *nicht* im Bereich eingeschlossen ist:

```
example[0:5]
# Ergebnis: 'Hallo'

example[6:-1]
# Ergebnis: 'Wel'

example[6:]
# Ergebnis: 'Welt'
```

Lässt man von der Bereichsangabe die Zahl vor oder nach dem Doppelpunkt weg, so wird die Zeichenkette von Beginn an beziehungsweise bis zum Ende ausgegeben.

Bei der Verwendung von Slices kann optional noch ein dritter Parameter angegeben werden, der die „Schrittweite“ festlegt, also angibt, jedes wie vielte Zeichen ausgewählt werden soll:

```
example[::2]
# Ergebnis: 'HloWl'
```

Wird für die Schrittweite ein negativer Wert angegeben, so wird der String von hinten nach vorne abgearbeitet.

String-Funktionen

Für Zeichenketten gibt es in Python einige Funktionen, die in der Form `zeichenkette.funktionsname()` angewendet werden können.

Suchen von Teilstrings

Mittels des `in`-Operators kann geprüft werden, ob ein Teilstring in einer anderen Zeichenkette enthalten ist:

```
'all' in 'Hallo'
# Ergebnis: True
```

Möchte man eine Zeichenkette nicht nur hinsichtlich der Existenz eines Teilstrings prüfen, sondern auch wissen, wie oft dieser darin enthalten ist, kann die Funktion `count()` verwendet werden:

```
# Anzahl des Buchstabens 'l' im Wort 'Hallo':

'Hallo'.count('l')
# Ergebnis: 2
```

Der Funktion `count()` können als weitere Argumente eine Start- und eine Stopp-Position übergeben werden, wenn nicht die ganze Zeichenkette, sondern nur ein bestimmter Abschnitt durchsucht werden soll:

```
# Anzahl des Buchstabens 'l' in den ersten drei Buchstaben von 'Hallo':

'Hallo'.count('l', 0, 3)
# Ergebnis: 1
```

Wie allgemein in Python üblich, wird bei Bereichsangaben die untere Grenze ins Intervall aufgenommen, die obere nicht; im obigen Beispiel werden daher nur die drei Indexnummern `0`, `1` und `2` geprüft. Mit der gleichen Sytanx kann mittels der Funktion `find()` die Index-Position des ersten Vorkommens eines Teilstrings innerhalb der Zeichenkette angezeigt werden:

```
'Hallo'.find('l')
# Ergebnis: 2

'Hallo'.find('l', 2, 4)
# Ergebnis: 3
```

Soll die Suche nicht „von links“, sondern „von rechts“ aus erfolgen, kann die Funktion `rfind()` genutzt werden; sie gibt als Ergebnis den Index des *letzten* Vorkommens des angegebenen Teilstrings zurück. Wird der gesuchte String im Zielstring nicht gefunden, liefern die Funktionen `find()` und `rfind()` den Wert `-1` als Ergebnis zurück.

Soll nur die Endung einer Zeichenkette untersucht werden, beispielsweise bei der Prüfung eines Dateityps, so kann anstelle der Funktion `rfind()` noch besser die Funktion `endswith()` genutzt werden. Diese liefert genau dann als Ergebnis `True` zurück, wenn die Zeichenkette mit dem angegebenen Teilstring endet.

In gleicher Weise wie `endswith()` gibt die Funktion `startswith()` als Ergebnis `True` zurück, wenn die Zeichenkette mit dem angegebenen Teilstring beginnt. Beide Funktionen liefern andererseits `False` als Ergebnis.

Komplexere Suchmuster sind mit Hilfe von regulären Ausdrücken und den zugehörigen Funktionen aus dem `re`-Modul möglich.

Ersetzen von Teilstrings

Zeichenketten sind unveränderbar. Daher kann der Index-Operator nicht auf der linken Seite des Zuweisungsoperators `=` stehen; beispielsweise würde die Eingabe von `'Hallo Welt'[0:5] = 'Salut'` einen `TypeError` erzeugen. Um eine solche Veränderung vorzunehmen, kann jedoch beispielsweise die speziell für Zeichenketten definierte `replace()`-Funktion genutzt werden, und der daraus resultierende String wieder der ursprünglichen Variable zugewiesen werden:

```
# 'Hallo' durch 'Salut' ersetzen:
example = 'Hallo Welt!'.replace('Hallo', 'Salut')
```

Komplexere Ersetzungen sind mit Hilfe von regulären Ausdrücken und den zugehörigen Funktionen aus dem `re`-Modul möglich.

Groß- und Kleinschreibung ändern

Python achtet bei der Behandlung von Zeichenketten auf die Groß- und Kleinschreibung. Sollen also beispielsweise zwei Wörter hinsichtlich nur ihres Inhalts, nicht jedoch hinsichtlich der Groß- und Kleinschreibung verglichen werden, so werden üblicherweise beide zunächst in Kleinbuchstaben umgewandelt. Hierfür kann die Funktion `lower()` verwendet werden:

```
'Hallo'.lower() == 'hallo'
# Ergebnis: True
```

Die Funktion `upper()`, wandelt in umgekehrter Weise alle Buchstaben einer Zeichenkette in Großbuchstaben um. Zwei ähnliche Funktionen sind `capitalize()`, bei einer Zeichenkette nur den ersten Buchstaben als Großbuchstaben und die restlichen als Kleinbuchstaben ausgibt sowie `title()`, die bei jedem Wort einer Zeichenkette den ersten Buchstaben als Großbuchstaben und die übrigen als Kleinbuchstaben ausgibt. Mit `swapcase()` können zudem alle Großbuchstaben einer Zeichenkette in Kleinbuchstaben und umgekehrt umgewandelt werden.

Leerzeichen entfernen, Text zentrieren

Mittels der Funktionen `lstrip()` oder `rstrip()` können Leerzeichen am Anfang oder am Ende einer Zeichenkette entfernt werden; mittels `strip()` werden Leerzeichen sowohl am Anfang wie auch am Ende einer Zeichenkette entfernt.

Die Funktion `rstrip()` wird häufig eingesetzt, um beim Einlesen einer Textdatei alle Leerzeichen am Ende der einzelnen Zeilen zu entfernen.

Mittels der Funktion `center()` kann eine Zeichenkette umgekehrt über eine bestimmte Textbreite zentriert ausgegeben werden. Beispielsweise gibt `'Hallo'.center(20)` als Ergebnis die Zeichenkette `'  Hallo  '` zurück. Eine derartige Formatierung kann beispielsweise für eine tabellarische Ausgabe von Daten nützlich sein.

Aufteilen und Zusammenfügen von Zeichenketten

Mittels der Funktion `split()` kann eine Zeichenkette in eine Liste von Teilstrings aufgeteilt werden. Als Argument wird dabei üein Zeichen oder eine Zeichenfolge angegeben, gemäß der die Aufteilung erfolgen soll. Verwendet man beispielsweise als Trennzeichen das Leerzeichen `' '`, so wird die Zeichenkette in einzelne Wörter aufgeteilt.

```
# Zeichenkette aufteilen:

'Hallo Welt'.split(' ')
# Ergebnis: ['Hallo', 'Welt']
```

Wird die Funktion `split()` ohne Trennzeichen als Argument aufgerufen, so wird die Zeichenkette an allen Whitespace-Zeichen aufgetrennt. So kann beispielsweise mit `len( text.split() )` die Anzahl der Wörter in einer Zeichenkette gezählt werden.

```
# Zeichenkette zusammenfügen:

' '.join(['Hallo', 'Welt'])
# Ergebnis: 'Hallo Welt'
```

Umgekehrt kann mittels der Funktion `join()` eine Liste von Teilstrings zu einer einzigen Zeichenkette zusammengesetzt werden. Dabei wird zunächst in der Zeichenkette ein Verbindungszeichen (oder eine Folge von Verbindungszeichen) angegeben, als Argument der `join()`-Funktion wird dann die Liste der zu verknüpfenden Teilstrings übergeben.

Formatierung von Zeichenketten

Bisweilen mag man beispielsweise mit `print()` den Wert einer Variablen als Teil einer Zeichenkette ausgeben. Zu diesem Zweck können in die Zeichenkette Platzhalter eingebaut werden, die dann durch die gewünschten Werte ersetzt werden. Dies funktioniert der „klassischen“ Methode nach (wie etwa in C) so:

```
var = 5829

'Der Wert von var ist %s.' % var
# Ergebnis: 'Der Wert von var ist 5829.'
```

Sollen an mehreren Stellen Ersetzungen vorgenommen werden, werden die Platzhalter in der gleichen Reihenfolge durch die Elemente eines gleich langen Variablen-Tupels ersetzt:

```
var_1 = 8913
var_2 = 7824

print('Der Wert von var_1 ist %s, \
       der Wert von var_2 ist %s' % (var_1, var_2) )

# Ergebnis: 'Der Wert von var_1 ist 8913, der Wert von var_2 ist 7824.'
```

Nach der neueren, mehr pyton-artigen Variante können Ersetzungen in Zeichenketten auch mittels der Funktion `format()` vorgenommen werden:

```
var   = 5829
var_1 = 8913
var_2 = 7824

print( 'Der Wert von var ist {}.\n'.format(var) )
# Ergebnis: 'Der Wert von var ist 5829.'

print( 'Der Wert von var_1 ist {}, \
        der Wert von var_2 ist {}.\n'.format(var_1, var_2) )
```

In diesem Fall werden die geschweiften Klammern innerhalb der Zeichenkette als Platzhalter angesehen und durch die als Argumente der Funktion `format()` angegebenen Variablen ersetzt. Als einzige Besonderheit müssen bei dieser Methode „echte“ geschweifte Klammern, die als Textsymbole in der Zeichenkette vorkommen sollen, durch `{{` bzw. `}}` dargestellt werden.

Eine komplette Liste an möglichen Format-Angaben findet sich in der offiziellen Python-Dokumentation.

Sonstiges:

Eine vollständige Liste an String-Funktionen erhält man, indem man die Funktion `dir()` auf einen beliebigen String anwendet, beispielsweise `dir(string_1)`. Nähere Informationen können dann beispielsweise mittels `help(string_1.replace)` aufgerufen werden.

`list` und `tuple` – Listen und Tupel

Listen und Tupel dienen der Sequenzierung von Objekten beliebigen Datentyps. Sie können unterschiedliche Datentypen in beliebiger Reihenfolge beinhalten. Listen werden in Python durch eckige Klammern, Tupel durch runde Klammern gekennzeichnet; die einzelnen Elemente werden durch jeweils ein Komma-Zeichen voneinander getrennt.

```
liste_1 = ['a', 'b', 'c', 1, 2, 3]  # oder:  list( 'a', 'b', 'c', 1, 2, 3 )
tupel_1 = ('a', 'b', 'c', 1, 2, 3)  # oder: tuple( ['a', 'b', 'c', 1, 2, 3] )
```

Der einzige Unterschied zwischen Listen, die mit `[` und `]` gekennzeichnet sind, und Tupeln, deren Elemente zwischen `(` und `)` stehen, liegt darin, dass die Inhalte von Listen verändert werden können, während die Inhalte von Tupeln unveränderbar sind.[3] Tupel können genutzt werden, um die Datensicherheit bestimmter Variablen, die an verschiedenen Stellen eines Programms genutzt werden, zu gewährleisten; im allgemeinen werden jedoch bevorzugt Listen genutzt.

Mittels der Schlüsselwörter `in` beziehungsweise `not in` kann geprüft werden, ob ein Objekt in einer Liste enthalten ist oder nicht:

```
'a' in liste_1
# Ergebnis: True
```

Indizierung von Listen und Tupeln

Auf die einzelnen Elemente einer Liste kann mit Hilf des mit Hilfe des Index-Operators `[ ]` zugegriffen werden. Die Syntax stimmt mit der *Indizierung von Zeichenketten* überein, da es sich bei diesen letztlich ebenfalls um Listen einzelner Buchstaben handelt:

```
# Erstes und zweites Element der Liste:

liste_1[0]
# Ergebnis: 'a'

liste_1[1]
# Ergebnis: 'b'

# Vorletztes und letztes Element der Liste:

liste_1[-2]
# Ergebnis: '2'

liste_1[-1]
# Ergebnis: '3'
```

Bereiche („Slices“) einer Liste können mit Hilfe des Index-Operators ausgewählt werden, indem man zwei mit einem Doppelpunkt getrennte Index-Zahlen `n1` und `n2` angibt; dabei muss beachtet werden, dass in Python bei Bereichsangaben die obere Grenze *nicht* im Bereich eingeschlossen ist:

```
liste_1[3:5]
# Ergebnis: [1, 2]
```

Wird bei der Verwendung von Slices die obere und/oder die untere Bereichsangabe weggelassen, so werden die Elemente vom Anfang an beziehungsweise bis zum Ende hin

[3] Genau genommen sind bei einem Tupel (oder auch einem `frozenset`) nur die Referenzen auf die enthaltenen Objekte unveränderlich. Enthält ein Tupel beispielsweise als erstes Argument eine Liste namens `l`, so kann dieser mittels `l.insert(0, 'Hallo!')` ein neues Element hinzugefügt werden. Das Tupel ändert sich dabei nicht, da die ID der Liste `l` unverändert bleibt.

ausgewählt. Ebenso wie bei Zeichenketten kann zudem ein dritter Parameter angegeben werden, der festlegt, jedes wie vielte Element ausgewählt werden soll:

```
liste_1[1:5:2]
# Ergebnis: ['a', 'c', 2]
```

Ist der Wert des dritten Parameters negativ, so wird die Liste von hinten nach vorne abgearbeitet.

Mehrdimensionale Listen

Listen können „verschachtelt“ sein, eine Liste kann also weitere Listen als Elemente beinhalten. Durch eine derartige Struktur könnten beispielsweise die Werte einer Tabelle gespeichert werden, die aus mehreren Zeilen besteht, wobei jede Zeile wiederum mehrere Spalten enthält.

Bei der Indizierung von verschachtelten Listen kann der Index-Operator mehrmals hintereinander angewendet werden:

```
liste_3 = [ ['a','b','c'], ['d','e','f'], ['g','h','i'] ]

# Zweites Listenelement ('Zeile') auswählen:

liste_3[1]
# Ergebnis: ['d','e','f']

# Drittes Element ('Spalte') dieser Zeile auswählen:
liste_3[1][2]
# Ergebnis: 'f'
```

Durch die erste Indizierung wird im obigen Beispiel eine Teilliste ausgewählt; auf diese Liste als Ergebnis der ersten Indizierung kann erneut der Indexoperator angewendet werden, um ein darin enthaltenes Element (oder auch einen Bereich) auszuwählen.

Listen-Funktionen

Für Listen existieren Funktionen, die in der Form `liste.funktionsname()` angewendet werden können.

Listen verknüpfen

Durch `liste_1.extend(liste2)` können zwei Listen miteinander kombiniert werden:

```
liste_1 = [1, 2, 3]
liste_2 = [4, 5, 6]

liste_1.extend(liste_2)
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
liste_1
# Ergebnis: [1, 2, 3, 4, 5, 6]
```

Durch die Funktion `extend()` wird die erste Liste um die Elemente der zweiten Liste erweitert; die Funktion `extend()` selbst liefert den Wert `None` als Ergebnis zurück. Möchte man eine derartige neue Liste erstellen, ohne dabei die ursprünglichen Listen zu verändern, kann der +-Operator verwendet werden:

```
liste_1 = [1, 2, 3]
liste_2 = [4, 5, 6]

liste3 = liste_1 + liste_2

liste_1
# Ergebnis: [1, 2, 3]

liste_2
# Ergebnis: [4, 5, 6]

liste_3
# Ergebnis: [1, 2, 3, 4, 5, 6]
```

Ebenso kann beispielsweise mittels `3 * liste_1` eine Liste erzeugt werden, die aus einer dreifachen Wiederholung der `liste_1` besteht:

```
liste_4 = 3 * liste_1 ; liste_4
# Ergebnis: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Möchte man eine zweite Liste als eigenes Element zu einer Liste hinzufügen, so kann die Funktion `append()` verwendet werden:

```
liste_1.append(liste_2)
# Ergebnis: [1, 2, 3, [4, 5, 6]]
```

Listen sortieren

Mittels der Funktion `sort()` können die Elemente einer Liste in aufsteigender Reihenfolge, mittels `reverse()` in absteigender Reihenfolge sortiert werden:

```
liste_5 = [3, 1, -5, 8, 2]

# Liste sortieren:

liste_5.sort()

liste_5
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
# Ergebnis: [-5, 1, 2, 3, 8]

liste_5.reverse()

liste_5
# Ergebnis: [8, 3, 2, 1, -5]
```

Elemente indizieren, einfügen und entfernen

Um zu prüfen, wie häufig ein bestimmtes Element in einer Liste enthalten ist, kann die Funktion `count()` angewendet werden:

```
liste_1.count(3)
# Ergebnis: 1
```

Soll nur geprüft werden, *ob* ein Element in einer Liste enthalten ist, so kann auch das Schlüsselwort `in` verwendet werden; im obigen Beispiel könnte somit `3 in liste_1` geschrieben werden, um zu prüfen, ob das Element `3` in der Liste vorkommt.

Die Position (des ersten Auftretens) eines Elements innerhalb einer Liste kann mittels der Funktion `index()` bestimmt werden:[4]

```
liste_1.index(3)
# Ergebnis: 2
```

Demnach kann der Wert `3` im obigen Beispiel mittels `liste_1[2]` aufgerufen werden und ist somit das dritte Element der Liste. Vor der Verwendung von `index()` sollte allerdings mittels `if element in liste:` stets geprüft werden, ob sich das Element in der Liste befindet, da ansonsten ein `ValueError` auftritt.

Mittels `insert(indexnummer, element)` kann ein neues Element *vor* der angegebenen Indexnummer eingefügt werden:

```
# Neues Element vor dem dritten Element (Index 2) einfügen:
liste_1.insert(2, 'Hallo')

liste_1
# Ergebnis: [1, 2, 'Hallo', 3]
```

Mittels `remove(element)` oder `pop(indexnummer)` können Elemente wieder aus der Liste entfernt werden:

```
# Element 'Hallo' aus Liste entfernen:
liste_1.remove('Hallo')
```

(continues on next page)

[4] Die Funktionen `count()` und `index()` sind die einzigen beiden Listenfunktionen, die auch für die unveränderlichen Tupel definiert sind.

(Fortsetzung der vorherigen Seite)

```
# Drittes Element entfernen:
liste_1.pop(2)

liste_1
# Ergebnis: [1, 2]
```

Beim Aufruf der `pop()`-Funktion wird das aus der Liste entfernte Objekt als Ergebnis zurückgegeben, was beispielsweise in „Stacks“ durchaus erwünscht ist. Das Löschen eines Listenbereichs zwischen zwei Indexnummern ist mittels `del liste[n1:n2]` möglich; auch bei dieser Bereichsangabe wird die obere Grenze nicht mit eingeschlossen.

Zu beachten ist wiederum, dass `remove()` einen `ValueError` auslöst, wenn sich das zu entfernende Element nicht in der Liste befindet, und `pop()` einen `IndexError` auslöst, wenn die Liste kürzer als die angegebene Indexnummer oder leer ist.

Listen kopieren

Listen können nicht einfach mittels des Zuweisungsoperators kopiert werden. Versucht man dies, so wird lediglich eine neue Referenz erstellt, die auf die gleiche Liste zeigt und die Inhalte der ursprünglichen Liste verändern kann:

```
# Liste erstellen:
liste_1 = [0,1,2,3,4,5]

# Neue Referenz auf die Liste:
liste_2 = liste_1

# Liste mittels der Referenz ändern:
liste_2[0] = 1

# Test:

liste_1
# Ergebnis: [1, 1, 2, 3, 4, 5]
```

Der Grund für dieses scheinbar seltsame Verhalten des Python-Interpreters liegt darin, dass auf diese Weise Listen direkt verändert werden können, wenn sie als *Argumente an Funktionen* übergeben werden. Da dies wesentlich häufiger vorkommt als das „echte“ Kopieren einer Liste, ist es in Python der Standard.

Um eine echte Kopie einer Liste zu erstellen, muss die Funktion `copy()` auf die ursprüngliche Liste angewendet werden:

```
# Liste erstellen:
liste_1 = [0,1,2,3,4,5]

# Kopie der Liste erstellen:
liste_2 = liste_1.copy()
```

Werden jetzt die Inhalte der zweiten Liste geändert, so bleiben die Inhalte der ersten Liste bestehen.

List-Comprehensions

Mit Hilfe so genannter List-Comprehensions können aus bestehenden Listen neue Listen erzeugt werden; dabei können beispielsweise Filter auf die die Elemente der bestehenden Liste angewendet werden; ebenso ist es möglich, Funktionen auf alle Elemente der bestehenden Liste anzuwenden und die jeweiligen Ergebnisse in der neuen Liste zu speichern.

Beispiele:

- Alle Elemente einer bestehenden Liste sollen quadriert werden:

```
# Ausgangsliste erstellen:
alte_liste = [1, 2, 3, 4, 5]

# List Comprehension anwenden:
neue_liste = [i**2 for i in alte_liste]

neue_liste
# Ergebnis: [1, 4, 9, 16, 25]
```

  In diesem Beispiel wird für jedes Element der bestehenden Liste, das jeweils mit einer temporären Variablen `i` bezeichnet wird, der Quadratwert `i**2` berechnet und das Ergebnis als neue Liste gespeichert.

- Aus einer bestehenden Liste sollen alle geradzahligen Werte ausgewählt werden:

```
# Ausgangsliste erstellen:
alte_liste = [1, 2, 3, 4, 5]

# List Comprehension anwenden:
neue_liste = [i for i in alte_liste if i % 2 == 0]

neue_liste
# Ergebnis: [2, 4]
```

  In diesem Beispiel werden die Elemente der bestehenden Liste, wiederum kurz mit `i` bezeichnet, unverändert in die neue Liste aufgenommen, sofern sie die angegebene Bedingung `i % 2 == 0` erfüllen.

- Aus zwei bestehenden Listen sollen alle Elemente ausgewählt werden, die in beiden Listen enthalten sind:

```
# Ausgangslisten erstellen:
liste_1 = [1, 2, 3, 4, 5]
liste_1 = [2, 3, 4, 5, 6]

# List Comprehension anwenden:
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
gemeinsame_elemente = [i for i in liste_1 if i in liste_2]

gemeinsame_elemente
# Ergebnis: [2, 3, 4, 5]
```

In diesem Beispil wird mittels der temporären Variablen `i` schrittweise geprüft, ob die Elemente der ersten Liste auch in der zweiten Liste enthalten sind und gegebenenfalls in die neue Liste aufgenommen.

- Die Werte zweier Listen sollen elementweise miteinander multipliziert werden:

```
# Ausgangslisten erstellen:
liste_1 = [1, 2, 3, 4, 5]
liste_2 = [2, 3, 4, 5, 6]

# List Comprehension anwenden:
produkte = [liste_1[i] * liste_2[i] for i in range(5)]

produkte
# Ergebnis: [2, 6, 12, 20, 30]
```

In diesem Beispiel wurde mit Hilfe der Funktion *range()* ein Bereich an ganzen Zahlen festgelegt, den die Variable `i` durchlaufen soll. Die Variable `i` bezeichnet in diesem Fall also nicht ein konkretes Element einer Liste, sondern vielmehr eine Indexnummer; mittels dieser Indexnummer kann dann auf die Elemente der Ausgangslisten zugegriffen werden.

Auch eine zusätzliche *if*-Bedingung, beispielsweise `if liste_1[i] > 2` wäre in diesem Fall möglich, würde logischerweise aber zu einem anderen Ergebnis führen.

List Comprehensions ermöglichen es allgemein, neue Listen schnell und gut lesbar zu erzeugen.

`set` und `frozenset` – Mengen

Ein Set bezeichnet in Python eine Menge an Objekten beliebigen Datentyps, wobei jedes Objekt nur ein einziges Mal in der Menge enthalten sein darf. Sets werden in Python in geschweifte Klammern gesetzt: Durch Anwendung von Operatoren auf paarweise je zwei Sets können – entsprechend den Regeln der Mengenlehre – neue Sets gebildet werden.

```
set_1 = {'a', 'b', 'c', 1, 2, 3}    # oder: set( ['a', 'b', 'c', 1, 2, 3] )
set_2 = {'b', 'c', 'd', 2, 3, 4}

# Schnittmenge:

set_1 & set_2
# Ergebnis: {'b', 'c', 2, 3}
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
# Vereinigungsmenge:

set_1 | set_2
# Ergebnis: {'a', 'b', 'c', 'd', 1, 2, 3, 4}

# Differenzmenge:

set_1 \ set_2
# Ergebnis: {'a', 1}

# Symmetrische Differenz (Entweder-Oder):

set_1 ^ set_2
# Ergebnis: {'a', 1, 4, 'd'}


# Test, ob set_1 eine Obermenge von set_2 ist:

set_1 > set_2
# Ergebnis: False
```

Mengen können unter anderem dazu genutzt werden, um aus einer Liste alle mehrfach vorkommenden Elemente heraus zu filtern. Hierzu wird etwa folgende Syntax genutzt:

```
any_list = ['a', 'a', 'b', 'c', 1, 2, 2, 3]

list_with_unique_elements = list(set(any_list))
# Ergebnis: ['a', 'b', 'c', 1, 2, 3]
```

Zum Arbeiten mit Mengen sind zusätzlich folgende Funktionen nützlich:

- Mit der `add()`-Funktion lassen sich Elemente zu einer Menge hinzufügen, beispielsweise `my_set.add('x')`.
- Mit der `discard()`-Funktion lassen sich Elemente aus einer Menge entfernen, beispielsweise `my_set.discard('x')`.
- Mit der `copy()`-Funktion kann eine Kopie einer Menge erstellt werden, beispielsweise `my_set2 = my_set_.copy()`.
- Mit der `clear()`-Funktion können alle Elemente einer Menge gelöscht werden, beispielsweise `my_set2.clear()`.

Neben üblichen Sets können mittels der Funktion `frozenset()` auch unveränderliche Listen erzeugt werden.

`dict` – Wörterbücher

In Python existiert ein weiterer Datentyp für Schlüssel-Wert-Paare. Ein solches `dict` ist somit aufgebaut wie ein Wörterbuch, das zu jedem Eintrag (Schlüssel) eine passende Erläuterung (Wert) liefert.

Zur Darstellung von `dicts` werden in Python geschweifte Klammern verwendet. Als Schlüssel werden meist `strings` genutzt, die zugehörigen Werte werden durch einen Doppelpunkt getrennt angegeben. Die einzelnen Schlüssel-Wert-Paare werden – wie die Elemente einer Liste – mit je einem Komma voneinander getrennt aufgelistet.

```
# Beispiel:

address-book = {
    name_1 : adresse_1,
    name_2 : adresse_2,
    name_3 : adresse_3,
    ...
    }
```

Anstelle von Zeichenketten können auch Zahlen oder andere unveränderbare Objekte genutzt werden – beispielsweise Tupel. Veränderbare Objekttypen wie beispielsweise Listen sind hingegen als Schlüssel nicht erlaubt.

Auf die einzelnen Werte eines `dicts` kann mittels des Index-Operators zugegriffen werden, wobei jedoch nicht ein numerischer Wert, sondern der Name eines Schlüssels in den eckigen Klammern angegeben wird:

```
address-book[name_1]
# Ergebnis: adresse_1
```

Mittels der `dict`-Funktionen `keys()`, `values()` und `items()` lassen sich so genannte „Views“ eines Wörterbuchs erzeugen. Bei einem View handelt es sich um eine Listen-Variable, die automatisch aktualisiert wird, wenn das zugehörige `dict` geändert wird.

Funktion	Ergebnis	Beschreibung
`anydict.keys()`	`[key_1, key_2, ...]`	Liste mit allen Schlüsseln
`anydict.values()`	`[value_1, value_2, ...]`	Liste mit allen Werten
`anydict.items()`	`[(key_1, value_1), (key_2, value_2), ...]`	Liste von Schlüssel-Wert-Tupeln

Mit `key_1 in anydict` kann geprüft werden, ob der Schlüssel `key_1` im Wörterbuch `anydict` vorhanden ist (Ergebnis: `True` oder `False`).

Um den zum Schlüssel `key_1` gehörigen Wert von `anydict` auszugeben, kann der Index-Operator `[ ]` genutzt werden:

```
anydict[key_1]
# Ergebnis: value_1 oder Error
```

Ist der Schlüssel nicht vorhanden, wird ein `KeyError` ausgelöst. Möchte man dies verhindern, so kann man folgenden Code nutzen:

```
anydict.get(key_1, default=None)
# Ergebnis: value_1 oder None
```

Mittels der `dict`-Funktion `get()` können Dictionaries auch als „Zähler“ genutzt werden: Soll beispielsweise festgehalten werden, wie häufig einzelne Worte in einem Text vorkommen, so legt man zunächst mittels `wortanzahl = dict()` ein neues `dict` an; anschließend kann wortweise geprüft werden, ob dieses Wort bereits als Schlüssel im `dict` enthalten ist. Ist dies der Fall, so wird der zugehörige „Counter“-Wert um `1` erhöht, andernfalls wird ein neuer Schlüssel mit dem Wert `1` angelegt. Ist das zu prüfende Wort in der Variable `wort` gespeichert, so genügt für die genannte Aufgabe bereits ein Einzeiler:

```
# Prüfen, ob Wort bereits im Dict enthalten ist;
# Counter um 1 erhöhen
wortanzahl[wort] = wortanzahl.get(wort, 0) + 1
```

`file` – Dateien

Datei-Objekte stellen in Python die Hauptschnittstelle zu externen Dateien auf dem Computer dar. Sie können genutzt werden, um Dateien beliebigen Typs zu lesen oder zu schreiben.

Datei-Objekte werden erzeugt, indem man die Funktion `open()` aufruft, und dabei den Namen der Datei sowie ein Kürzel für den gewünschten Bearbeitungsmodus angibt:

```
myfile = open('file.txt', 'r')
```

Als Bearbeitungsmodus kann `'r'` (lesen), `'w'` (schreiben) oder `'rw'` (lesen und schreiben) gewählt werden. Sollen binäre Dateien gelesen beziehungsweise geschrieben werden, muss an das jeweilige Kürzel ein `b` angehängt werden, beispielsweise bezeichnet `'rb'` den Lesemodus einer binären Datei.

Einlesen von Dateien

Wird eine Datei im Lesemodus geöffnet, so kann sie beispielsweise mittels der Funktion `read()` im Ganzen als ein einziger String eingelesen werden:

```
# Datei als einzelnen String einlesen:
long_string = myfile.read()
```

Diese Methode ist für größere Dateien nicht empfehlenswert. Besser ist es, mittels der Funktion `readline()` eine Datei Zeile für Zeile einzulesen. Bei jedem solchen Aufruf wird

die jeweils eingelesene Zeile als Ergebnis zurückgegeben und der „Cursor“ für die aktuelle Position in der Datei auf die nächste Zeile gesetzt. Mit der Funktion `readlines()` wird die Datei zeilenweise in eine Liste von Zeichenketten eingelesen.

Noch einfacher ist ein zeilenweises Einlesen, indem die Datei-Variable selbst als iterierbares Objekt an eine *for*-Schleife übergeben wird:

```
# Schleife über alle Zeilen der Datei:
for line in myfile:

    # Gib die aktuelle Zeile aus:
    print(line)
```

Am Ende eines Lesezugriffs sollte die Datei mittels `close(myfile)` wieder geschlossen werden.

Schreiben in Dateien

Um Text in eine Datei zu schreiben, wird diese zunächst im Schreibmodus geöffnet:

```
myfile = open('file.txt', 'w')
```

Anschließend kann mittels der Funktion `write()` eine (gegebenenfalls auch mehrzeilige) Zeichenkette in die Datei geschrieben werden:

```
myfile.write('Hallo Welt!\n')
```

Zudem kann mittels `myfile.writelines(stringliste)` kann eine Liste von einzelnen Zeichenketten in eine Datei geschrieben werden. Die einzelnen Zeichenketten werden bei `write()` als auch bei `writelines()` geschrieben „wie sie sind“, es muss also bei Bedarf ein Zeilenende-Zeichen `\n` am Ende der zu schreibenden Zeichenkette(n) sein, damit auch in der geschriebenen Datei an der jeweiligen Stelle ein Zeilenumbruch erfolgt.

Am Ende eines Schreibzugriffs *muss* die Datei mittels `close(myfile)` wieder geschlossen werden, da nur dann das Datei-Attribut `mtime` („Modifikationszeit“) korrekt gesetzt wird.

Da während des Einlesens oder Schreibens von Dateien prinzipiell auch mit Fehlern gerechnet werden muss, müsste stets mittels `try..except..finally` ein Ausnahme-Handling eingebaut werden. Eine (bessere) Alternative hierzu beziehungsweise zum manuellen Schließen eines `file`-Objekts besteht darin, ein `with`-Statement mit folgender Syntax zu verwenden:

```
with open("file.txt", 'r') as myfile:
    for line in myfile:
        print(line)
```

Mit dieser Variante wird die Datei automatisch geschlossen, auch wenn beim Lesen oder Schreiben ein Fehler aufgetreten ist.

Kontrollstrukturen

Die folgenden Kontrollstrukturen können zur Steuerung eines Programms verwendet werden, wenn einzelne Code-Blöcke nur unter bestimmten Bedingungen oder auch mehrfach ausgeführt werden sollen. In Python werden dabei keine Klammern zur Markierung von Code-Blöcken verwendet; stattdessen werden Einrückungen zur Gruppierung und Kenntlichmachung einzelner Code-Blöcke genutzt. Üblicherweise wird für jede Einrückungstiefe ein Tabulator-Zeichen (entspricht vier Leerzeichen) verwendet.

`if`, `elif` und `else` – Fallunterscheidungen

Mit Hilfe von `if`-Abfragen ist es möglich, Code-Teile nur unter bestimmten Bedingungen ablaufen zu lassen. Ist die `if`-Bedingung wahr, so wird der anschließende, mit einer Einrückung hervorgehobene Code ausgeführt.

```
if bedingung:

    ... Anweisungen ...
```

Optional können nach einem `if`-Block mittels `elif` eine oder mehrere zusätzliche Bedingungen formuliert werden, die jedoch nur dann untersucht werden, wenn die erste `if`-Bedingung falsch ist. Schließlich kann auch eine `else`-Bedingung angegeben werden, die genau dann ausgeführt wird, wenn die vorherige Bedingung (beziehungsweise alle vorherigen Bedingungen bei Verwendung eines `elif`-Blocks) nicht zutreffen.

Insgesamt kann eine Fallunterscheidung beispielsweise folgenden Aufbau haben:

```
if bedingung_1:
    ...

elif bedingung_2:
    ...

else:
    ...
```

Bei der Untersuchung der einzelnen Bedingungen werden die Werte von Variablen häufig mittels *Vergleichsoperatoren* überprüft. Mehrere Teilbedingungen können zudem mittels logischer Operatoren zu einer Gesamtbedingung verknüpft werden:

- Werden zwei Bedingungen mit `and` verknüpft, so ist das Ergebnis genau dann wahr, wenn beide Bedingungen erfüllt sind.
- Werden zwei Bedingungen mit `or` verknüpft, so ist das Ergebnis dann wahr, wenn mindestens eine der beiden Bedingungen (oder beide zugleich) erfüllt sind.
- Der Operator `not` kehrt den Wahrheitswert einer Bedingung um, eine wahre Bedingung liefert also `False` als Ergebnis, eine falsche Bedingung `True`.

Da die logischen Operatoren eine geringer Priorität haben als die Vergleichsoperatoren, können mehrere Vergleiche auch ohne Klammern mittels `and` beziehungsweise `or` verbunden werden.

```
# Beispiel:

var_1 = (578 + 94) / 12
var_2 = (1728) / 144

if var_1 > var_2:
    print("var_1 is larger als var_2.")
elif var_1 < var_2:
    print("var_1 is smaller als var_2.")
else:
    print("var_1 is equal var_2.")

# Ergebnis: var_1 is equal var_2.
```

Beinhaltet eine Variable *var* einen Listentyp, beispielsweise ein *Tupel*, einen *String* , ein *Dict* oder ein Set, so ergibt `if var` den Wert `False`, falls die Länge der Liste gleich Null ist, und `True`, falls die jeweilige Liste mindestens ein Element beinhaltet.

`while` und `for` – Schleifen

Mittels Schleifen kann ein Code-Abschnitt wiederholt ausgeführt werden. Python bietet hierfür zweierlei Möglichkeiten: Mittels einer `while`-Schleife wird Code so lange ausgeführt, solange eine angegebene Bedingung wahr ist; mit einer `for`-Schleife lassen sich auch komplexere Schleifentypen erzeugen.

`while`-Schleifen

Eine `while`-Schleife hat allgemein folgende Syntax:

```
while bedingung:

    ... Anweisungen ...
```

Ist die Bedingung wahr, so werden die im unmittelbar folgenden Block stehenden Anweisungen ausgeführt. Vor jedem weiteren Durchlauf wird wieder geprüft, ob die angegebene Bedingung erfüllt ist; sobald dies nicht der Fall ist, wird die Schleife abgebrochen.

Unmittelbar an den `while`-Block kann optional auch noch ein `else`-Block angefügt werden, der genau einmal ausgeführt wird, sobald die `while`-Bedingung das erste mal den Wahrheitswert `False` annimmt. Damit kann beispielsweise eine Programmstruktur folgender Art geschaffen werden:

```
while eingabe != passwort:

    ... weitere Texteingabe ...

else:

    ... richtiges Passwort ...
```

`break` und `continue`

Der Ablauf einer `while` kann durch folgende beide Schlüsselwörter im Inneren des Anweisungsblocks beeinflusst werden:

- Mittels `break` wird die Schleife unmittelbar an der jeweiligen Stelle beendet.
- Mittels `continue` kann der Rest des aktuellen Schleifendurchlaufs übersprungen werden; die Schleife wird anschließend mit dem nächsten Schleifendurchlauf fortgesetzt.

Mittels der `break`-Anweisung können beispielsweise Endlos-Schleifen programmiert werden, die nur unter einer bestimmten Bedingung beendet werden:

```
while True:

    ... Anweisungen ...

    if bedingung:
        break
```

Die Schlüsselwörter `break` und `continue` können ebenfalls in `for`-Schleifen eingesetzt werden.

`for`-Schleifen

Eine `for`-Schleife hat allgemein folgende Syntax:

```
for var in iterierbares-objekt:

    ... Anweisungen ...
```

Ein iterierbares Objekt kann beispielsweise eine Liste, ein Tupel, oder auch ein String sein. Im einfachsten Fall kann auch mittels der Funktion *range()* ein iterierbares Objekt mit bestimmter Länge erzeugt werden:

```
summe = 0
for i in range(1,10):
    summe += i

print(summe)
# Ergebnis: 45
```

Im diesem Beispiel durchläuft die Zählvariable `i` alle Werte im angegebenen Zahlenbereich, wobei bei Verwendung von `range()` die untere Schranke zum Zahlenbereich dazugehört, die obere jedoch nicht; es werden im obigen Beispiel also alle Zahlen von `1` bis `9` aufsummiert.

pass – Die Platzhalter-Anweisung

Beim Entwickeln eines Programms kann es passieren, dass eine Kontrollstruktur Funktion oder Fehlerroutine zunächst nur teilweise implementiert wird. Eine Anweisung ohne Inhalt würde allerdings einen Syntaxfehler zur Folge haben. Um dies zu vermeiden, kann die `pass`-Anweisung eingefügt werden, die keine weitere Bedeutung für das Programm hat. Beispiel:

```
var_1 = None

if var_1 is None:
    pass
else:
    print( "The value of var_1 is %s." % var_1 )
```

Die `pass`-Anweisung stellt somit eine Erleichterung beim Entwickeln von Programmen dar, da man sich mit ihrer Hilfe zunächst an wichtigeren Programmteilen arbeiten kann. In fertigen Programmen werden `pass`-Anweisungen nur selten verwendet.

Funktionen

Oftmals werden bestimmte Code-Stücke an verschiedenen Stellen eines Programms eingesetzt. Derartige Teile, die gewissermaßen kleine „Unterprogramme“ darstellen, werden als Funktionen definiert. Funktionen liefern am Ende stets genau *ein* Ergebnis zurück; dieses kann jedoch von den übergebenen Funktionsargumenten und/oder von äußeren Bedingungen abhängig sein.[1]

Definition eigener Funktionen

Eine Funktion wird in Python nach folgendem Schema definiert:

```
def function_name(argument_1, argument_2, ...):
    """
    Docstring (short description of the function's purpose)
    """

    function_code
    ...

    return result
```

Zum Benennen von Funktionen werden in Python üblicherweise nur kleine Buchstaben verwendet. Häufig besteht ein Funktionsname aus einem Verb, das die Wirkungsweise der Funktion beschreibt; liegt jedoch der Zweck einer Funktion nur darin, einen bestimmten Wert auszugeben, so kann der Funktionsname auch aus einem Substantiv bestehen, das den jeweiligen Wert beschreibt – beispielsweise wird die Wurzelfunktion meist mit `sqrt()` („square root“) anstelle von `calculate_sqrt()` bezeichnet. Besteht ein Funktionsname aus mehreren Wörtern, so werden diese zur besseren Lesbarkeit mittels Unterstrichen getrennt.

Nach ihrer Definition kann die Funktion dann folgendermaßen aufgerufen werden:

```
function_name(argument_1, argument_2, ...)
```

Eine Funktion kann, abhängig von ihrer Definition, beim Aufruf kein, ein oder auch mehrere Argumente in der jeweiligen Reihenfolge verlangen. Bei jedem Aufruf wird dann der Funktions-Code ausgeführt und ein entsprechender Ergebniswert zurück gegeben; ist im

[1] Soll eine Funktion mehrere Ergebnisse liefern, so müssen diese als *Liste* oder *Tupel* zurückgegeben werden.

Funktionsblock keine `return`-Anweisung enthalten, wird von der Funktion automatisch der Wert `None` zurückgegeben.[2]

Beim Aufruf der Funktion werden die einzelnen Argumente üblicherweise gemäß ihrer Position direkt übergeben, beispielsweise `argument_1 = 5 ; function_name(argument_1)`. Es ist jedoch auch für eine gegebenenfalls bessere Lesbarkeit des Quellcodes möglich, bei der Übergabe von Argumenten deren Bezeichnung mit anzugeben, beispielsweise `function_name(argument_1=5)`.

Da jede Funktion ein kleines Unterprogramm darstellt, sollte sie durch einen Docstring dokumentiert werden. Es gibt Werkzeuge, die Docstrings verwenden, um automatisch gedruckte Dokumentation zu erzeugen (beispielsweise Sphinx) oder um Benutzer interaktive Hilfe anzubieten (beispielsweise Ipython). Werden bei einer Funktion Argumente verlangt, so sollten diese ebenfalls im Docstring kurz beschrieben werden (Typ, Bedeutung).

Optionale Argumente

In Python ist es möglich, bei der Definition von Funktionen optionale Argumente mit einem Standard-Wert anzugeben. Wird ein solches optionales Argument beim Aufruf der Funktion nicht explizit angegeben, so wird stattdessen der Standard-Wert genutzt.

Die Definition einer solchen Funktion lautet etwa:

```
def my_function(argument_1, argument_2="Test")

    return (argument_1, argument_2)
```

Die Funktion kann anschließend wahlweise mit einem oder mit zwei Argumenten aufgerufen werden:

```
my_function(5)
# Ergebnis: (5, 'Test')

my_function(5, 7)
# Ergebnis: (5, 7)
```

Hat eine Funktion sowohl „normale“ als auch optionale Argumente, so müssen die optionalen Argumente am Ende der Funktion angegeben werden.

Ebenso ist es möglich, einer Funktion eine optionale Liste oder ein optionales Dict für zusätzliche Argumente anzugeben. Üblicherweise lautet die Syntax dafür:

[2] In einer Funktionsdefinition können auch, wenn `if`-Bedingungen in ihr vorliegen, an mehreren Stellen `return`-Anweisungen auftreten. Sobald eine `return`-Anweisung erreicht wird, wird die Funktion unmittelbar beendet und der jeweilige Ergebniswert zurück gegeben. Steht nur `return` (ohne expliziten Ergebniswert), so wird ebenfalls der Wert `None` zurück gegeben.

Soll eine Funktion mehrere Werte als Ergebnis liefern, so müssen diese als Liste oder Tupel an das Programm zurückgegeben werden.

```
def any_function(argument_1, argument_2="default value", *args, **kwargs)

    pass
```

Beim Aufruf der Funktion muss `argument_1` angegeben werden, die Angabe von `argument_2` ist optional; zusätzlich können weitere unbenannte oder benannte Argumente angegeben werden, beispielsweise `any_function(5,2,7,foo=9)`. Im obigen Beispiel können alle unbenannten Argumente innerhalb der Funktion über die Variable `args`, alle benannten über die Variable `kwargs` aufgerufen werden. Werden beim Funktionsaufruf keine weiteren Argumente übergeben, so ist `args` innerhalb der Funktion ein leeres Tupel und `kwargs` ein leeres Dict.

```
def sum_it_up(num_1, num_2, *nums):

    list_sum = functools.reduce(lambda x,y: x+y, nums)

    return num_1 + num_2 + len(nums)
```

Hat man im Quellcode vorab eine Liste an Objekten definiert, die man dann als Argumente an eine Funktion übergeben möchte, so muss diese bei der Übergabe entpackt werden. Dies ist mittels des `*`-Operators möglich:

```
my_list = [1,3,5,7]

sum_it_up(*my_list)
# Ergebnis: 6
```

Im obigen Beispiel wurde die Liste `my_list` bei der Übergabe an die Funktion entpackt; der Aufruf von `sum_it_up(*my_list)` ist in diesem Fall also identisch mit `sum_it_up(1, 3,5,7)`. Die ersten beiden Zahlen werden dabei als Argumente für `num_1` und `num_2` angesehen, die verbleibenden beiden werden innerhalb der Funktion in die Variable `nums` gespeichert.[3]

Nach dem gleichen Prinzip kann ein Dict mit Schlüsselwort-Wert-Paaren mittels `**` entpackt an eine Funktion übergeben werden; dabei müssen alle im Dict enthaltenen Schlüsselwörter als mögliche Funktionsargumente erlaubt sein.

Veränderliche und unveränderliche Argumente

In Python werden Argumente an Funktionen via Zuweisung (Position oder Benennung) übergeben. Wird beispielsweise eine Variable `x = any_object` als Argument an eine Funktion übergeben, so wird `x`, also eine Referenz auf `any_object`, als Wert übergeben. Welchen Einfluss die Funktion auf das Argument hat, hängt dabei vom Objekt-Typ ab:

[3] Dass die obige Funktion die Länge der zusätzlichen Zahlenliste zum Ergebnis dazu addiert, soll an dieser Stelle nur die Funktionalität der optionalen Argumente zeigen. Um alle Elemente dieser Liste zu summieren, muss zusätzlich das Modul `functools` geladen werden; in der Beispielaufgabe *Quader* wird dies näher behandelt.

- Ist `any_object` ein veränderliches Objekt, beispielsweise eine Liste, ein Dict oder ein Set, so kann dieses durch die Funktion direkt verändert werden:

```
my_list = [1,2,3]

def foo(any_list):
    any_list.append(4)

foo(my_list)

print(my_list)
# Ergebnis: [1,2,3,4]
```

Wird allerdings der übergebenen Referenz `x` in der Funktion ein neues Objekt zugewiesen, so entspricht dies einer Definition einer neuen Variablen innerhalb der Funktion als Namensraum.[4] Die ursprüngliche Referenz `x` bleibt in diesem Fall unverändert und zeigt nach wie vor auf das ursprüngliche Objekt:

```
my_list = [1,2,3]

def foo(any_list):
    any_list = [1,2,3,4]

foo(my_list)

print(my_list)
# Ergebnis: [1,2,3]
```

- Ist `any_object` ein unveränderliches Objekt, beispielsweise ein String oder ein Tupel, so wird bei der Übergabe an die Funktion eine Kopie des Objekts erzeugt; das ursprüngliche Objekt kann somit durch die Funktion nicht verändert werden:

```
my_string = 'Hello World!'

def foo(any_string):
  any_string.replace('World', 'Python')

foo(my_string)

print(my_string)
# Ergebnis: 'Hello World!'
```

Möchte man den obigen Beispielcode so umschreiben, dass nach Aufruf der Funktion die Variable `my_string` den Wert `'Hello Python!'` bekommt, so muss man folgendermaßen vorgehen:

[4] Variablen sind nur innerhalb ihres Namensraums gültig. Ausnahmen sind globale Variablen, die beispielsweise zu Beginn eines Moduls definiert sind. Eine solche kann dann innerhalb einer Funktion mittels `global var_name` als global gekennzeichnet werden, wobei `var_name` der Name der Variablen ist.

```
my_string = 'Hello World!'

def foo(any_string):
  x = any_string.replace('World', 'Python')
  return x

my_string = foo(my_string)

print(my_string)
# Ergebnis: 'Hello Python!'
```

Da Strings nicht verändert werden können, kann eine Veränderung der zugehörigen Variablen also nur über eine neue Zuweisung erfolgen.

Die Übergabe eines veränderlichen Datentyps, beispielsweise einer Liste, als Argument an eine Funktion ist zwar performance-technisch günstig, da nur eine neue Referenz auf die Liste erzeugt werden muss; es können aber unerwartete Fehler auftreten, wenn durch eine Funktion das übergebene Original der Liste unmittelbar verändert wird. Durch Verwendung von Tupeln anstelle von Listen kann dies auf Kosten von CPU und Memory ausgeschlossen werden.

Globaler und lokaler Namensraum

Jeder Funktionsblock hat einen eigenen Namensraum, was bedeutet, dass Variablen, die innerhalb des Funktionsblocks definiert werden, nur im Programm existieren, bis der Funktionsblock abgeschlossen ist. Derartige Variablen, die nur innerhalb der Funktion „sichtbar“ sind, werden „lokale“ Variablen genannt.

Einer lokalen Variablen kann dabei der gleiche Name zugewiesen werden wie einer Variablen, die außerhalb des Funktionsblocks definiert wurden; sie „überdeckt“ in diesem Fall die nicht-lokale Variable: Der Interpreter sucht primär im lokalen Namensraum nach dem Variablennamen und erst sekundär im nicht-lokalen Namensraum, wenn keine lokale Variable mit dem angegebenen Namen existiert. und nur.

Wird einer lokalen Variablen ein Wert zugewiesen, so ändert sich also der Wert einer nichtlokalen Variablen nicht:

```
# Nicht-lokale Variable definieren:
x = 0

# Funktion mit lokaler Variablen definieren:
def myfunc():
    x = 1
    print(x)

# Funktion aufrufen:

myfunc()
# Ergebnis: 1
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
# Wert von x prüfen:

x
# Ergebnis: 0
```

Soll der Wert einer nicht-lokalen Variablen durch die Funktion verändert werden, so kann diese bei einem veränderlichen Datentyp als Argument an die Funktion übergeben werden. Variablen mit nicht veränderlichem Datentyp können, wie im letzten Abschnitt beschrieben, eine Veränderung erreicht werden, nur durch eine Zuweisung des Rückgabewerts der Funktion geändert werden.

Eine weitere prinzipielle Möglichkeit, die jedoch möglichst vermieden werden sollte, ist es, mittels des Schlüsselworts `global` einen Variablennamen primär im nicht-lokalen Namensraum zu suchen:

```
# Nicht-lokale Variable definieren:
x = 0

# Funktion mit globaler Variablen definieren:
def myfunc():
    global x = 1
    print(x)

# Funktion aufrufen:

myfunc()
# Ergebnis: 1

# Wert von x prüfen:

x
# Ergebnis: 1
```

Das Schlüsselwort `global` hat nur Auswirkungen innerhalb der Funktion, eine Variable kann also nicht von außerhalb der Funktion als „global“ gekennzeichnet und als solche der Funktion aufgezwungen werden. Dennoch kann die Verwendung von `global` zu unerwarteten Seiteneffekten führen, wenn eine Variable prinzipiell von mehreren Stellen aus verändert werden kann, da in diesem Fall nicht immer auf den ersten Blick einwandfrei feststellbar ist, von wo aus die Variable verändert wurde.

Lambda-Ausdrücke

Bei so genannten Lambda-Ausdrücken handelt es sich um Mini-Funktionen, die sehr kompakt implementiert werden können. Das Schlüsselwort zur Definition eines Lambda-Ausdrucks ist `lambda`, gefolgt von möglichen Argumenten, die der Funktion beim Aufruf

übergeben werden sollen, und einem Doppelpunkt. Hinter diesem Doppelpunkt wird in der gleichen Zeile die Wirkungsweise des Lambda-Ausdrucks definiert, beispielsweise:

```
add = lambda x1, x2: x1 + x2

add(5,3)
# Ergebnis: 8
```

Bei der Definition eines Lambda-Ausdrucks entfällt also das Schlüsselwort `def`, und die Argumente werden unmittelbar hinter dem Schlüsselwort `lambda` ohne runde Klammern angegeben.

Ein Nachteil von Lambda-Ausdrücken ist, dass in ihnen keine Schleifen, Kontrollstrukturen oder komplexeren Anweisungsblöcke vorkommen dürfen.[5] Ein wichtiger Vorteil ist hingegen, dass Lambda-Ausdrücke keinen expliziten Namen haben müssen. Beispielsweise kann der gesamte Lambda-Ausdruck, wenn man ihn in runde Klammern setzt, unmittelbar wie eine Funktion aufgerufen werden:

```
result = (lambda x1, x2: x_1 + x_2)(5,3)

print(result)
# Ergebnis: 8
```

Lambda-Ausdrücke werden auch häufig in Kombination mit den Builtin-Funktion *filter()* und *map()* eingesetzt, um jeweils auf alle Elemente einer Liste angewendet zu werden:

```
# Beispiel 1:

my_list = [1,2,3,4,5,6,7,8,9]

even_numbers = filter(lambda x: x % 2 == 0, my_list)

list(even_numbers)
# Ergebnis: [2,4,6,8]


# Beispiel 2:

my_list = [1,2,3,4]

even_numbers = map(lambda x: x * 2, [1,2,3,4])

list(even_numbers)
# Ergebnis: [2,4,6,8]
```

Ebenso kann ein Lambda-Ausdruck als Kriterium für die `sort()`-Funktion genutzt werden, um beispielsweise eine Liste von Zweier-Tupeln nach den ersten Elementen der Tupel zu sortieren:

[5] Eine *Bedingte Wertzuweisung* in der Form `lambda x: a if condition else b` ist allerdings erlaubt.

```
my_list = [(3, 1), (1, 2), (11, -3), (5, 10)]

my_list.sort(key=lambda x: x[0])

print(my_list)
# Ergebnis: [(1, 2), (3, 1), (5, 10), (11, -3)]
```

Auch bei der Entwicklung von graphischen Oberflächen sind Lambda-Ausdrücke nützlich, um einzelnen Maus-Events oder Tastenkombinationen bestimmte Funktionen zuzuweisen.

Builtin-Funktionen

Neben der bereits erwähnten `print()`-Funktion, die zur Anzeige von Texten und Variablen auf dem Bildschirm dient, gibt es in Python weitere grundsätzlich verfügbare Funktionen, so genannte „Builtin"-Funktionen; Diese Funktionen sind ohne das Importieren weiterer *Module* unmittelbar verfügbar. Eine Übersicht über diese Funktionen findet sich im *Anhang: Standard-Funktionen*.

Klassen und Objektorientierung

Um Quellcode besser zu strukturieren, werden bei der objektorientierten Programmierung so genannte Klassen erzeugt, die jeweils bestimmte Eigenschaften („Attribute“) und Funktionen („Methoden“) besitzen. Klassen werden in Python allgemein durch einzelne Zeichenketten mit großen Anfangsbuchstaben gekennzeichnet.

Eine neue Klasse wird in Python mit dem Schlüsselwort `class`, gefolgt vom Klassennamen und einem Doppelpunkt eingeleitet. Alle darauf folgenden Definitionen von Eigenschaften und Funktionen, die zur Klasse gehören, werden um eine Tabulatorweite (üblicherweise 4 Leerzeichen) eingerückt.

Definition und Initialisierung eigener Klassen

Ebenso wichtig wie der Begriff einer Klasse ist der Begriff der Instanz einer Klasse. Während beispielsweise die Klasse „Wecker“ einen Objekttyp eines meist unhöflich Lärm erzeugenden Gerätes darstellt, so ist ein einzelner neben einem Bett stehender Wecker ein konkreter Vertreter dieser Klasse. Eine solche Instanz hat, zumindest in der Programmierung, stets alle in der Klasse definierten Eigenschaften und Funktionen, allerdings mit möglicher unterschiedlicher Ausprägung (beispielsweise Farbe oder Klingelton).

In Python könnte die Implementierung einer Beispielklasse etwa so aussehen:

```
class AlarmClock():
    """
    Just a simple Class example.
    """

    def __init__(self, color, sound):
        """
        Initialize a new alarm clock.

        Arguments:

        * color (string): Color of the alarm clock.
        * sound (string): Ringing sound of the clock.
        """
        self.color = color
        self.sound = sound
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
    def show_color(self):
        return self.color

    def ring(self):
        return self.sound + "!!!"
```

Im obigen Beispiel wurde zunächst die Klasse `AlarmClock` definiert und als. erstes mit einem *Docstring* versehen, der eine kurze Beschreibung der Klasse liefert.

In der Funktion `__init__()` wird anschließend festgelegt, wie eine neue Instanz der Klasse erzeugt wird. Die angegebenen Argumente werden dabei als Eigenschaften der Instanz, die in Python mit `self` bezeichnet wird, gespeichert. Nach der Initialisierung stehen im obigen Beispiel dem neuen Objekt dann die beiden weiteren angegebenen Funktionen `show_color()` und `ring()` bereit, die als Ausgabe der jeweiligen Objektvariablen dienen.

Die Methode `__init__()` wird automatisch aufgerufen, wenn man den Namen der Klasse als Funktion aufruft. Eine neue Instanz einer Klasse lässt sich im Fall des obigen Beispiels also folgendermaßen erzeugen:

```
# Initialisierung:
my_alarm_clock = AlarmClock("green", "Ring Ring Ring")


# Objekttyp testen:

type(my_alarm_clock)
# Ergebnis: __main__.AlarmClock


# Funktionen testen:

my_alarm_clock.show_color()
# Ergebnis: 'green'

my_alarm_clock.ring()
# Ergebnis: 'Ring Ring Ring!!!'
```

Mittels `type(objektname)` kann allgemein angezeigt werden, zu welcher Klasse ein beliebiges Python-Objekt gehört; ebenso kann mit `isinstance(objektname, klassenname)` geprüft werden, ob ein Objekt eine Instanz der angegebenen Klasse ist.

Möchte man eine konkrete Instanz wieder löschen, so ist dies allgemein mittels `del(name_der_instanz)` möglich, im obigen Fall also mittels `del(my_alarm_clock)`. Genau genommen wird die Instanz allerdings nur dann gelöscht, wenn die letzte Referenz auf die Instanz gelöscht wird. Sind beispielsweise mittels `alarm_clock_1 = alarm_clock_2 = AlarmClock("blue", "Ring!")` zwei Referenzen auf die gleiche Instanz erzeugt worden, so wird mittels `del(alarm_clock_1)` nur die erste Referenz gelöscht; die Instanz selbst bleibt weiter bestehen, da die Variable `alarm_clock_2` immer noch darauf verweist. Beim Löschen der letzten Referenz auf wird automatisch Pythons „Garbage Collector“ aktiv und übernimmt die Aufräumarbeiten, indem die entsprechende `__del__()`-Methode

aufgerufen wird; ebenso wird der benötigte Platz im Arbeitsspeicher dadurch automatisch wieder freigegeben.

Sollen bei der Löschung einer Instanz weitere Aufgaben abgearbeitet werden, so können diese in einer optionalen Funktion `__del__()` innerhalb der Klasse festgelegt werden.

Allgemeine Eigenschaften von Klassen

Klassen dienen allgemein dazu, die Attribute und Methoden (kurz: „Member“) einzelner Objekte festzulegen. Bei der Festlegung der Attribute und Methoden wird jeweils das Schlüsselwort `self` genutzt, das auf die jeweilige Instanz einer Klasse verweist.

Attribute eines Objekts werden mittels `self.attributname = wert` festgelegt; dies erfolgt üblicherweise bereits bei der Initialisierung. Anschließend kann auf die so definierten Attribute innerhalb der Klasse mittels `self.attributname` und außerhalb mittels `instanzname.attributname` zugegriffen werden.

Methoden eines Objekts werden ebenso wie Funktionen definiert, mit der einzigen Besonderheit, dass als erstes Argument stets `self` angegeben wird. Innerhalb der Klasse können so definierte Methoden mittels `self.methodenname()` und außerhalb mittels `instanzname.methodenname()` aufgerufen werden.

Geschützte und private Attribute und Methoden

In manchen Fällen möchte man vermeiden, dass die Attribute eines Objekts durch andere Objekte manipuliert werden können; ebenso sind manche Methoden nur für den „internen“ Gebrauch innerhalb einer Klasse geeignet. In Python gibt es für diesen Zweck sowohl geschützte („protected“) als auch private („private“) Attribute und Methoden:

- Geschützte Member (Attribute und Methoden) werden durch einen einfach Unterstrich vor dem jeweiligen Namen gekennzeichnet. Auf derartige Attribute oder Methoden kann weiterhin von einer anderen Stelle aus sowohl lesend als auch schreibend zugegriffen werden; es ist vielmehr eine Konvention zwischen Python-Entwicklern, dass auf derartige Member nicht direkt zugegriffen werden sollte.
- Private Member werden durch einen doppelten Unterstrich vor dem jeweiligen Namen gekennzeichnet. Auf derartige Attribute kann außerhalb der Klasse weder lesend noch schreibend zugegriffen werden.

Attribute sollten beispielsweise dann als geschützt oder privat gekennzeichnet werden, wenn sie nur bestimmte Werte annehmen sollen. In diesem Fall werden zusätzlich so genannte „Getter“ und „Setter“-Methoden definiert, deren Aufgabe es ist, nach einer Prüfung auf Korrektheit den Wert des entsprechenden Attributs auszugeben oder ihm einen neuen Wert zuzuweisen.

Setter- und Getter-Methoden werden bevorzugt als so genannte „Properties“ definiert; dazu wird folgende Syntax verwendet:

```
class MyClass:

def __init__(self):
    self._myattr = None

def get_myattr(self):
    return self._myattr

def set_myattr(self, value):
    # todo: check if value is valid

    self._myattr = value

myattr = property(get_myattr, set_myattr)
```

Mittels der `property()`-Funktion wird die Setter- und Getter-Methode eines geschützten oder privaten Attributs dem gleichen, nicht-geschützten Attributnamen zugewiesen. Von außerhalb der Klasse ist dieses gezielte Handling also nicht sichtbar, das entsprechende Attribut erscheint von außen also wie ein gewöhnliches Attribut. Dieses Prinzip der Kapselung von Aufgaben ist typisch für objektorientierte Programmierung: Wichtig ist es, die Aufgabe eines Objekts klar zu definieren sowie seine „Schnittstellen“, also seine von außerhalb zugänglichen Attribute und Methoden, festzulegen. Solange das Objekt als einzelner Baustein seine Aufgabe damit erfüllt, braucht man sich als Entwickler um die Interna dieses Bausteins nicht weiter Gedanken zu machen.

Statische Member

Neben Attributen und Methoden, die jede einzelne Instanz einer Klasse an sich bereitstellt, können innerhalb einer Klasse auch Attribute und/oder Methoden definiert werden, die für alle Instanzen der Klasse gleichermaßen gelten. Derartige Klassenmember werden „statisch“ genannt.

Statische Attribute lassen sich erzeugen, indem innerhalb der Klasse (üblicherweise gleich zu Beginn des Klassen-Blocks) die gewünschten Attribute wie normale Variablen gesetzt werden, also ohne vorangestelltes `self` und außerhalb einer Methoden-Definition. Der Zugriff auf statische Attribute kann dann (sowohl lesend als auch schreibend) wahlweise mittels `Klassenname.attributname` oder mittels `Instanzname.attributname` erfolgen:

```
# Definition eines statischen Attributs:

class MyClass:

    myattr = 42

    pass

# Zugriff auf ein statisches Attribut:
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
MyClass.myattr
# Ergebnis: 42

MyClass().myattr
# Ergebnis: 42
```

Statische Methoden werden ebenso wie gewöhnliche Methoden definiert, außer dass bei der Definition das `self` als erstes Argument weggelassen wird; stattdessen wird die Methode anschließend mittels der Funktion `staticmethod()` zur statische Methode deklariert:

```
# Definition einer statischen Methode:

class MyClass:

    def mymethod():
        print("Hello!")

    mymethod = staticmethod(mymethod)

# Aufruf einer statischen Methode:

MyClass.mymethod()
# Ergebnis: Hello!

MyClass().mymethod()
# Ergebnis: Hello!
```

Statische Attribute und Methoden können auch dann genutzt werden, wenn (noch) keine Instanz der Klasse existiert.

„Magic“ Member

Als „Magic Member“ werden private Attribute und Funktionen von Klassen bezeichnet, die es beispielsweise ermöglichen, einzelne Instanzen der Klassen mittels Operatoren miteinander in Relation zu setzen oder Builtin-Funktionen auf die einzelnen Instanzen anzuwenden. Die Bezeichnung „magisch“ stammt daher, dass diese Methoden und Attribute selten direkt mit ihrem Namen angesprochen werden, aber dafür oft implizit genutzt werden – wie beispielsweise die `__init__()` oder `__del__()`-Funktionen als Konstruktor- und Destruktor-Methoden einzelner Instanzen. Beispielsweise wird anstelle `MyClass.__init__()` wird üblicherweise `MyClass()` geschrieben, um eine neue Instanz einer Klasse zu erzeugen; bei letzterer Variante wird die `__init__()`-Funktion vom Python-Interpreter implizit aufgerufen.

Folgende Member sind für die Repräsentation von Objekten vorgesehen:

- Mittels der Methode `__str()__` wird eine für Menschen gut lesbare Zeichenkette definiert, die beim Aufruf von `str(MyClass)` als Objektbeschreibung ausgegeben werden soll.

- Die Methode `__repr()` wird so definiert, dass sie Python-Code als Ergebnis zurückgibt, bei dessen Ausführung eine neue Instanz der jeweiligen Klasse erzeugt wird.
- Die Methode `__call__()` bewirkt, sofern sie definiert ist, dass eine Instanz einer Klasse – ebenso wie eine Funktion – aufgerufen werden kann.

Folgende Member sind für den Zugriff auf Attribute vorgesehen:

- Mit dem Attribut `__dict__` wird als *dict* aufgelistet, welche Attribute und zugehörigen Werte die jeweilige Instanz der Klasse aktuell beinhaltet.
- Durch das (statische) Attribut `__slots__` kann mittels einer Liste festgelegt werden, welche Attributnamen die Instanzen einer Klasse haben. Wird versucht, mittels `del(instanzname.attributname)` ein Attribut einer Instanz zu löschen, dessen Name in der `__slots__`-Liste enthalten ist, so schlägt das Löschen mit einem `AttributeError` fehl. Umgekehrt wird auch dann ein `AttributeError` ausgelöst, wenn man versucht, ein neues Attribut für die Instanz festzulegen, dessen Name nicht in der `__slots__`-Liste enthalten ist.
- Mit der Methode `__getattr__()` wird definiert, wie sich die Klasse zu verhalten hat, wenn ein angegebenes Attribut abgefragt wird, aber nicht existiert. Wahlweise kann ein Standard-Wert zurückgegeben werden, üblicherweise wird jedoch ein `AttributeError` ausgelöst.

  Mit der Methode `__getattribute__()` wird das angegebene Attribut ausgegeben, sofern es existiert. Andernfalls kann – wie bei `__getattr__()` – wahlweise ein Standard-Wert zurückgegeben oder ein `AttributeError` ausgelöst werden. Wenn die `__getattribute()__` definiert ist, wird die `__getattr__()`-Methode nicht aufgerufen, außer sie wird innerhalb von `__getattribute()__` explizit aufgerufen.
  Zu beachten ist außerdem, dass innerhalb von `__getattribute()__` nur mittels `klassenname.__getattribute__(self, attributname)` auf ein Attribut zugegriffen werden darf, nicht mittels `self.attributname`; im letzteren Fall würde ansonsten eine Endlos-Schleife erzeugt.

- Mit der Methode `__setattr__()` wird eine Routine angegeben, die aufgerufen wird, wenn ein Attribut neu erstellt oder verändert wird. Dafür wird zunächst der Name des Attributs und als zweites Argument der zuzuweisende Wert angegeben. Insbesondere kann mit dieser Methode kontrolliert werden, dass keine unerwünschten Attribute vergeben werden können.
  Bei der Verwendung von `__setattr__()` ist zu beachten, dass die Wertzuweisung mittels `self.__dict__['attributname'] = wert` erfolgen muss, da ansonsten eine Endlos-Schleife erzeugt würde.
- Mit `__delattr__()` wird als Methode festgelegt, wie sich eine Instanz beim Aufruf von `del(instanzname.attributname)` verhalten soll.

Folgende Member sind für den Vergleich zweier Objekte vorgesehen:

- Mit den Methoden `__eq__()` („equal“) und `__ne__()` („not equal“) kann festgelegt werden, nach welchen Kriterien zwei Instanzen verglichen werden sol-

len, wenn `instanz_1 == instanz_2` beziehungsweise `instanz_1 != instanz_2` aufgerufen wird. Diese Operator-Kurzschreibweise wird intern in `instanz_1.__eq__(instanz_2)` übersetzt, es wird also die Equal-Methode des ersten Objekts aufgerufen.

- Mit den Methoden `__gt__()` („greater than“) und `__ge__()` („greater equal“) kann festgelegt werden, nach welchen Kriterien sich zwei Instanzen verglichen werden sollen, wenn `instanz_1 > instanz_2` beziehungsweise `instanz_1 >= instanz_2` aufgerufen wird. Entsprechend kann mit `__lt__()` („less than“) und `__le__()` („less equal“) kann festgelegt werden, nach welchen Kriterien sich zwei Objekte verglichen werden, wenn `instanz_1 < instanz_2` beziehungsweise `instanz_1 <= instanz_2` aufgerufen wird.
- Mit der Methode `__hash__()` wird ein zu der angegebenen Instanz gehörender Hash-Wert ausgegeben, der die Instanz als Objekt eindeutig identifiziert.

Folgendes Member ist für logische Operationen vorgesehen:

- Mit der Methode `__bool__()` wird festgelegt, in welchen Fällen eine Instanz den Wahrheitswert `True` oder den Wahrheitswert `False` zurückgeben soll, wenn `bool(instanz)` aufgerufen wird; dies erfolgt beispielsweise implizit bei *if*-Bedingungen.

Folgende Member sind für numerische Operationen vorgesehen:

- Mit den Methoden `__int__()`, `__oct__()`, `__hex__()`, `__float__()`, `__long__()` und `__complex__()` kann festgelegt werden, wie das Objekt durch einen Aufruf von `int(instanz)`, `oct(instanz)` usw. in den jeweiligen numerischen Datentyp umzuwandeln ist.
- Mit den Methoden `__pos__()` und `__neg__()` kann festgelegt werden, welche Ergebnisse die unären Operatoren `+instanz` beziehungsweise `-instanz` liefern sollen; zusätzlich kann mit der Methode `__abs__()` (Absolut-Betrag) festgelegt werden, welches Ergebnis ein Aufruf von `abs(instanz)` liefern soll. Ebenso wird durch die Methoden `__round__()` (Rundung) bestimmt, welches Ergebnis beim Aufruf von `round(instanz)` bzw. `round(instanz, num)` zu erwarten ist.
- Mit den Methoden `__add__()`, `__sub__()`, `__mul__()` und `__truediv__()` wird festgelegt, welches Ergebnis sich bei der Verknüpfung zweier Instanzen mit den vier Grundrechenarten ergeben soll, also `instanz_1 + instanz_2`, `instanz_1 - instanz_2`, usw. Zusätzlich wird durch die Methode `__pow__()` (Potenzrechnung) festgelegt, welches Ergebnis `instanz_1 ** instanz__2` liefern soll.
- Mit der Methode `__floordiv__()` wird die Bedeutung von `instanz_1 // instanz_2` festgelegt; bei normalen Zahlen wird damit derjenige `int`-Wert bezeichnet, der kleiner oder gleich dem Ergebnis der Division ist (beispielsweise ergibt `5 // 3` den Wert `1`, und `-5 // 3` den Wert `-2`). Zudem wird durch die Methode `__mod__()` (Modulo-Rechnung) die Bedeutung von `instanz_1 % instanz_2` festgelegt, was bei normalen Zahlen dem ganzzahligen Divisionsrest entspricht.

Folgende Member sind für Wertzuweisungen vorgesehen:

- Mit den Methoden `__iadd__()`, `__isub__()`, `__imul__()` und `__itruediv__()` wird festgelegt, welches Ergebnis sich bei der kombinierten Wertzuweisung zwei-

er Instanzen mit den vier Grundrechenarten ergeben soll, also `instanz_1 += instanz_2`, `instanz_1 -= instanz_2`, usw. Zusätzlich wird durch die Methode `__ipow__()` (Potenzrechnung) festgelegt, welches Ergebnis `instanz_1 **= instanz__2` liefern soll.

- Mit der Methode `__ifloordiv__()` wird die Bedeutung von `instanz_1 //= instanz_2` festgelegt; bei normalen Zahlen wird `instanz_1` dabei derjenige Wert zugewiesen, der sich bei der Auswertung von `instanz_1 // instanz_2` ergibt. In gleicher Weise wird durch die Methode `__imod__()` (Modulo-Rechnung) die Bedeutung von `instanz_1 %= instanz_2` festgelegt; hierbei erhält `instanz_1` als Wert das Ergebnis von `instanz_1 % instanz_2`.

Folgende Member sind für Container, Slicings und Iteratoren vorgesehen:

- Die Methode `__len__()` wird aufgerufen, wenn mittels `len(instanz)` die Länge eines Containers geprüft werden soll.
- Die Methode `__contains__()` wird aufgerufen, wenn mittels `element in instanz` geprüft wird, ob das als Argument angegebene Element im Container enthalten ist.
- Die Methode `__getitem__()` wird aufgerufen, wenn mittels `instanz[key]` der zu dem als Argument angegebenen Schlüssel gehörende Wert abgerufen werden soll.
- Die Methode `__setitem__()` wird aufgerufen, wenn mittels `instanz[key] = value` dem als erstes Argument angegebenen Schlüssel der als zweites Argument angegebene Wert zugewiesen werden soll
- Die Methode `__delitem__()` wird aufgerufen, wenn mittels `del instanz[element]` das als Argument angegebene Element aus dem Contaniner gelöscht werden soll.

Folgende Member sind für die Verwendung des Objekts innerhalb von *with*-Konstrukten vorgesehen:

- Mit der Methode `__enter__()` werden Anweisungen definiert, die einmalig ausgeführt werden sollen, wenn eine Instanz der Klasse in eine *with*-Umgebung geladen wird.
- Mit der Methode `__exit__()` werden Anweisungen definiert, die einmalig ausgeführt werden sollen, wenn eine *with*-Umgebung um eine Instanz der Klasse wieder verlassen wird.

Mit Hilfe der „Magic Members“ können also neue Klassen von Objekten definiert werden, deren Verhalten dem von bestehenden Klassen (Zahlen, Listen, usw.) ähnelt. Wird einem Operator, beispielsweise dem Plus-Operators `+`, mittels der Funktion `__add__()` eine neue, auf den Objekttyp passendere Bedeutung zugewiesen, so spricht man auch von „Operator-Überladung“.

Objekte zeichnen sich also weniger durch ihre Bezeichnung, sondern vielmehr durch ihre Eigenschaften und Methoden aus. Dieses Konzept wird bisweilen auch als „Duck Typing“ bezeichnet:

> „Wenn ich einen Vogel sehe, der wie eine Ente läuft, wie eine Ente schwimmt und wie eine Ente schnattert, dann nenne ich diesen Vogel eine Ente.“
>
> —James Riley

Kann eine Funktion umgekehrt auf verschiedene Objekt-Typen angewendet werden, so nennt man sie polymorph. Beispielsweise kann die Builtin-Funktion `sum()` auf beliebige listen-artige Objekttypen angewendet werden, sofern für diese eine `__add__()`-Funktion definiert ist. Durch Polymorphismus als Spracheigenschaft wird einmal geschriebener Code oftmals auch an einer anderen Stelle verwendbar.

Vererbung

Mit dem Begriff „Vererbung“ wird ein in der Objekt-orientierten Programmierung sehr wichtiges Konzept bezeichnet, das es ermöglicht, bei der Definition von fein spezifizierten Objekte auf allgemeinere Basis-Klassen zurückzugreifen; die Basis-Klasse „vererbt“ dabei ihre Attribute und Methoden an die abgeleitete Sub-Klasse, wobei in dieser weitere Eigenschaften hinzukommen oder die geerbten Eigenschaften angepasst werden können. Aus mathematischer Sicht ist die Basis-Klasse eine echte Teilmenge der daraus abgeleiteten Klasse, da diese alle Eigenschaften der ursprünglichen Klasse (und gegebenenfalls noch weitere) enthält.

Um die Eigenschaften einer Basis-Klasse in einer neuen Klasse zu übernehmen, muss diese bei der Klassendefinition in runden Klammern angegeben werden:

```
class SubClass(MyClass):

    pass
```

Bis auf diese Besonderheit werden alle Attribute und Methoden in einer abgeleiteten Klasse ebenso definiert wie in einer Klasse, die keine Basis-Klasse aufweist.

In Python ist es prinzipiell möglich, dass eine Klasse auch mehrere Basis-Klassen aufweist; in diesem Fall werden die einzelnen Klassennamen bei der Definition der neuen Klasse durch Kommata getrennt angegeben. Die links stehende Klasse hat dabei beim Vererben der Eigenschaften die höchste Priorität, gleichnamige Attribute oder Methoden werden durch die weiteren Basis-Klassen also nicht überschrieben, sondern nur ergänzt. Da durch Mehrfach-Vererbungen allerdings keine eindeutige Baumstruktur mehr vorliegt, also nicht mehr auf den ersten Blick erkennbar ist, aus welcher Klasse die abgeleiteten Attribute und Methoden ursprünglich stammen, sollte Mehrfach-Vererbung nur in Ausnahmefällen und mit Vorsicht eingesetzt werden.

Dekoratoren

Dekoratoren werden in Python als Kurzschreibweise verwendet, um bestimmte, innerhalb einer Klasse definierte Methoden mit zusätzlichen Methoden zu „umhüllen“.

Der wohl wichtigste Dekorator ist *@property*: Mit Hilfe dieses Dekorators kann eine get-Methode zu einem „Attribut“ gemacht werden, dessen Wert nicht statisch in einer Variablen abgelegt ist, sondern dynamisch mittels der dekorierten get-Methode abgefragt wird. Die grundlegende Funktionsweise ist folgende:

```
# Beispiel-Klasse definieren:
class C(object):

    # Variable, die nur "intern" verwendet werden soll:
    _counter = 0

    # get-Methode, mittels derer ein Wert ausgegeben werden soll:
    def get_counter(self):
        return self._counter

    # Markierung der get-Methode als Property
    counter = property(get_counter)
```

Anhand des obigen Beispiels kann man gut erkennen, dass in der Beispiel-Klasse `C` neben der als nur zur internen Verwendung vorgesehenen Variablen `_counter` auch noch ein zweites Attribug `counter` definiert wird, und zwar explizit als Aufruf von `property()`.

Wird via `c = C()` eine neue Instanz der Klasse erzeugt, so wird mittels `c.counter` die `get_counter()`-Funktion aufgerufen. Die für einen Methoden-Aufruf typischen runden Klammern entfallen also, von außen hat es den Anschein, als würde auf ein gewöhnliches Attribut zugegriffen. Intern hingegen wird die `_counter`-Variable, die womöglich an anderen Stellen innerhalb der Klasse verändert wird, ausgegeben.

Als Kurzschreibweise für ähnliche Methoden-„Wrapper“ gibt es in Python folgende Syntax:

```
class C(object):

    _counter = 0

    @property
    def get_counter(self):
        return self._counter
```

Die Zeile `@property` wird dabei „Dekorator“ genannt. Diese Kurzschreibweise hat den gleichen Effekt wie das obige, explizite Wrapping der zugehörigen Methode.

... to be continued ... .. TODO

Generatoren und Iteratoren

Generatoren sind Objekte, die über eine `__next__()`-Funktion verfügen, also bei jedem Aufruf von `next(generator_object)` einen neuen Rückgabewert liefern. Man kann sich einen Generator also vorstellen wie eine Funktion, die mit jedem Aufruf das nächste Element einer Liste zurückgibt. Kann der Generator keinen weiteren Rückgabewert liefern, wird ein `StopIteration`-Error ausgelöst.

Der Vorteil von Generatoren gegenüber Listen liegt darin, dass auch bei sehr großen Datenmengen kein großer Speicherbedarf nötig ist, da immer nur das jeweils aktuell zurückgegebene Objekt im Speicher existiert. Beispielsweise handelt es sich auch bei *File-*

Objekten um Generatoren, die beim Aufruf von `next(fileobject)` jeweils die nächste Zeile der geöffneten ausgeben. Auf diese Weise kann der Inhalt einer (beliebig) großen Datei beispielsweise mittels einer `for`-Schleife abgearbeitet werden, ohne dass der Inhalt der Datei auf einmal eingelesen und als Liste gespeichert werden muss.

Generatoren werden mit einer Syntax erzeugt, die der von *List Comprehensions* sehr ähnlich ist; es wird lediglich runde Klammern anstelle der eckigen Klammern zur Begrenzung des Ausdrucks verwendet:

```
# List Comprehension:

mylist = [i**2 for i in range(1,10)]

mylist
# Ergebnis: [1, 4, 9, 16, 25, 36, 49, 64, 81]

# Generator:

mygen  = (i**2 for i in range(1,10))

next(mygen)
# Ergebnis: 1

next(mygen)
# Ergebnis: 4

# ...
```

Generatoren können auch verschachtelt auftreten; bestehende Generatoren können also zur Konstruktion neuer Generatoren verwendet werden:

```
# Neuen Generator mittels eines existierenden erzeugen:

mygen2 = (i**2 for i in mygen)

next(mygen2)
# Ergebnis: 81

next(mygen2)
# Ergebnis: 256
```

Python kann diese Art von verschachtelten Generatoren sehr schnell auswerten, so dass Generatoren allgemein sehr gut für die Auswertung großer Datenströme geeignet sind. Zu beachten ist lediglich, dass der Generator bei jedem Aufruf – ähnlich wie eine Funktion – einen Rückgabewert liefert, der von sich aus nicht im Speicher verbleibt, sondern entweder unmittelbar weiter verarbeitet oder manuell gespeichert werden muss.

Module und Pakete

Module

Module bestehen, ebenso wie Python-Skript, aus Quellcode-Dateien mit der Endung `.py`. Prinzipiell kann somit jedes Python-Skript als eigenständiges Python-Modul angesehen werden, es existieren allerdings auch Module, die nur nur aus Hilfsfunktionen bestehen.

In der Praxis werden Module allerdings in erster Linie verwendet, um den Quellcode eines umfangreicheren Software-Projekts leichter zu organisieren und übersichtlich zu halten. Eine grobe Regel besagt, dass ein Modul nur so umfangreich sein sollte, dass sich seine Inhalte und sein Verwendungszweck mit ein bis zwei Sätzen zusammenfassen lassen sollten.

Jedes Modul stellt einen eigenen Namensraum für Variablen dar, so dass gleichnamige Funktionen, die in unterschiedlichen Modulen definiert sind, keine Konflikte verursachen.

Modulnamen werden in Python allgemein in Kleinbuchstaben geschrieben. Um in das aktuelle Programm ein externes Modul, also eine andere `.py`-Datei, zu integrieren, ist folgende Syntax üblich:

```
import modulname

# Beispiel:

import antigravity
import this
```

Das zu importierende Modul muss sich hierbei im Python-Pfad befinden, also entweder global installiert oder im aktuellen Verzeichnis enthalten sein. Andernfalls ist eine Anpassung der Variablen `sys.path` nötig.

Verwendung von Modulen

Nach dem Importieren eines Moduls können die im Modul definierten *Klassen*, *Funktionen* und *Variablen* mit folgender Syntax aufgerufen werden:

```
modulname.Klassenname
modulname.variablenname
modulname.funktionsname()
```

Um bei längeren Modulnamen Schreibarbeit sparen zu können, ist beim Importieren eines Moduls auch eine abkürzende Syntax möglich:

```
import modulname as myname

# Beispiel:

import math as m
```

Anschließend kann das Kürzel als Synonym für den Modulnamen verwendet werden, beispielsweise `m.pi` anstelle von `math.pi`.

Eine weitere Vereinfachung ist möglich, wenn man nur einzelne Klassen oder Funktionen eines Moduls importieren möchte. Hierfür ist folgende Syntax möglich:

```
from modulname import Klassenname   # oder
from modulname import funktionsname
```

Dabei können auch mehrere Klassen- oder Funktionsnamen jeweils durch ein Komma getrennt angegeben werden. Die so importieren Klassen bzw. Funktionen können dann direkt aufgerufen werden, als wären sie in der aktuellen Datei definiert.

Es ist auch möglich, mittels `from modulname import *` alle Klassen und Funktionen eines Moduls zu importieren. Hiervon ist allerdings (zumindest in Skript-Dateien) dringend abzuraten, da es dadurch unter Umständen schwer nachvollziehbar ist, aus welchem Modul eine später benutzte Funktion stammt. Zudem können Namenskonflikte entstehen, wenn mehrere Module gleichnamige Funktionen bereitstellen.

Hilfe zu Modulen

Mittels `help(modulname)` kann wiederum eine Hilfeseite zu dem jeweiligen Modul eingeblendet werden, in der üblicherweise eine Beschreibung des Moduls angezeigt wird und eine Auflistung der im Modul definierten Funktionen. Bei den Standard-Modulen wird zudem ein Link zur entsprechenden Seite der offiziellen Python-Dokumentation https://docs.python.org/3/ angegeben.

Die `__name__`-Variable

Jedes Modul bekommt, wenn es importiert wird, automatisch eine `__name__`-Variable zugewiesen, die den Namen des Moduls angibt. Wird allerdings eine Python-Datei unmittelbar als Skript mit dem Interpreter aufgerufen, so bekommt dieses „Modul“ als `__name__`-Variable den Wert `__main__` zugewiesen.

Wenn Python-Module importiert werden, dann werden sie einmalig vom Interpreter ausgeführt, das heißt alle darin aufgelisteten Definitionen und Funktionsaufrufe werden zum Zeitpunkt des Importierens (einmalig) aufgerufen. Möchte man einige Funktionen in einer Python-Datei nur dann ausführen, wenn die Datei als Skript aufgerufen wird, nicht jedoch, wenn sie als Modul in ein anderes Programm eingebunden wird, so kann man dies mittels folgender Anweisung erreichen:

```
if __name__ == __main__:

    # execute this only if the current file is interpreted directly
```

Dies ist insbesondere für Mini-Programme nützlich, die wahlweise als selbstständiges Programm aufgerufen, oder in ein anderes Programm eingebettet werden können.

Module erneut laden

Ist ein Modul einmal importiert, so wird jede weitere `import`-Anweisung des gleichen Moduls vom Python-Interpreter ignoriert. Dies ist in den meisten Fällen von Vorteil, denn auch wenn beispielsweise mehrere Module eines Programms das Modul `sys` importieren, so wird dieses nur einmal geladen.

Schreibt man allerdings selbst aktiv an einem Programmteil und möchte die Änderungen in einer laufenden Interpreter-Sitzung (z.B. Ipython) übernehmen, so müsste der Interpreter nach jeder Änderung geschlossen und neu gestartet werden. Abhilfe schafft in diesem Fall die im Modul `imp` definierte Funktion `reload()`, die ein erneutes Laden eines Moduls ermöglicht:

```
import imp
import modulname

# Modul neu laden:
imp.reload(modulname)
```

Dies funktioniert auch, wenn ein Modul mit einer Abkürzung importiert wurde, beispielsweise `import modulname as m`; in diesem Fall kann das Modul mittels `imp.reload(m)` neu geladen werden.

Pakete

Mehrere zusammengehörende Module können in Python weiter in so genannten Paketen zusammengefasst werden. Ein Paket besteht dabei aus einem einzelnen Ordner, der mehrere Module (`.py`-Dateien) enthält.

Ein Programm kann somit in mehrere Teilpakete untergliedert werden, die wiederum mittels der `import`-Anweisung wie Module geladen werden können. Enthält beispielsweise ein Paket `pac` die Module `a` und `b`, so können diese mittels `import pac` geladen und mittels `pac.a` beziehungsweise `pac.b` benutzt werden; zur Trennung des Paket- und des Modulnamens wird also wiederum ein Punkt verwendet. Ebenso kann mittels `import pac.a` nur das Modul `a` aus dem Paket `pac` geladen werden.

Die `import`-Syntax für Pakete lautet somit allgemein:

```
# Alle Module eines Pakets importieren:
import paket
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
# Oder: Einzelne Module eines Pakets importieren:
import paket.modulname

# Alternativ:
from paket import modulname
```

Wird ein Modul mittels `from paket import modulname` importiert, so kann es ohne Angabe des Paketnamens benutzt werden; beispielsweise können darin definierte Funktionen mittels `modulname.funktionsname()` aufgerufen werden. Eine weitere Verschachtelung von Paketen in Unterpakete ist ebenfalls möglich.

Relative und absolute Pfadangaben

Um Module aus der gleichen Verzeichnisebene zu importieren, wird in Python folgende Syntax verwendet:

```
# Modul aus dem gleichen Verzeichnis importieren
from . import modulname
```

Mit `.` wird dabei der aktuelle Ordner bezeichnet. Ebenso kann ein Modul mittels `from .. import modulname` aus dem übergeordneten Verzeichnis und mittels `from ... import modulname` aus dem nochmals übergeordneten Verzeichnis importiert werden. Dies ist allerdings nicht in der Hauptdatei möglich, mit welcher der Python-Interpreter aufgerufen wurde und die intern lediglich den Namen `__main__` zugewiesen bekommt: Diese darf nur absolute Pfadangaben für Imports enthalten.

Empfehlungen für Paket-Strukturen

Möchte man selbst ein Paket zu einer Python-Anwendung erstellen, so ist etwa folgender Aufbau empfehlenswert:[1]

```
My_Project/
|
|- docs
|   |- conf.py
|   |- index.rst
|   |- installation.rst
|   |- modules.rst
|   |- quickstart.rst
|   |- reference.rst
|
|- my_project/
```

(continues on next page)

[1] Siehe auch Open Sourcing a Python Project the Right Way und Filesystem structure of a Python project für weitere Tips.

(Fortsetzung der vorherigen Seite)

```
|   |- main.py
|   |- ...
|
|- tests/
|   |- test_main.py
|   |- ...
|
|- CHANGES.rst
|- LICENSE
|- README.rst
|- requirements.txt
|- setup.py
|- TODO.rst
```

Das Projekt-Verzeichnis sollte den gleichen Namen wie die spätere Anwendung haben, allerdings mit einem Großbuchstaben beginnen, um Verwechslungen mit dem gleichnamigen Paket-Verzeichnis zu vermeiden. Das Projekt-Verzeichnis sollte neben dem eigentlichen Projekt-Paket zumindest noch die Ordner `docs`, und `test` umfassen, in denen eine Dokumentation des Projekts und zum Programm passende *Tests* enthalten sind; zusätzlich kann das Projekt einen `lib`-Ordner mit möglichen C-Erweiterungen enthalten. In manchen Projekten werden zudem ausführbare Programm-Dateien (meist ohne die Endung `.py`) in einem weiteren Ordner `bin` abgelegt; hat ein Programm allerdings nur eine einzelne aufrufbare Datei, so kann diese auch im Projekt-Hauptverzeichnis abgelegt werden.

Die mit Großbuchstaben benannten Dateien sind selbsterklärend, die Dateien `requirements.txt` und `setup.py` sind für die Installationsroutine vorgesehen.

Paketinstallation mit `setuptools` und `setup.py`

Das Paket `setuptools` aus der Python-Standardbibliothek stellt einige Funktionen bereit, die für das Installieren von Python-Paketen hilfreich sind. Dazu wird üblicherweise im Projekt eine Datei `setup.py` angelegt, die unter anderem eine Programmbeschreibung, den Namen und die Emailadresse des Autors, Informationen zur Programmversion und zu benötigten Fremdpaketen enthält. Zudem können mit der Funktion `find_packages()` aus dem `setuptools`-Paket die im Projektverzeichnis enthaltenen Pakete automatisch gefunden und bei Bedarf installiert werden.

Python-Entwickler haben das `setuptools`-Paket oftmals bereits installiert (`aptitude install python3-setuptools`), da es für das zusätzliche Installieren von weiteren Paketen mittels `pip3 install paketname` oder `easy_install3 paketname` hilfreich ist. Soll allerdings sichergestellt sein, dass auch bei Anwendern das `setuptools`-Paket installiert ist oder bei Bedarf nachinstalliert wird, kann die Datei ez_install.py von der Projektseite heruntergeladen und in das eigene Projektverzeichnis kopiert werden. In der Datei `setup.py` sind dann üblicherweise folgende Routinen vorhanden:

```
try:
    from setuptools import setup, find_packages
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
except ImportError:
    import ez_setup
    ez_setup.use_setuptools()
    from setuptools import setup, find_packages

import os

import my_project

my_project_path = os.path.abspath(os.path.dirname(__file__))

long_description =
"""
Eine ausführliche Beschreibung des Programms.
"""

setup(
    name='my_project',
    version=my_project.__version__,
    url='http://github.com/my_account/my_project/',
    license='GPLv3',
    author='Vorname Nachname',
    author_email='name@adresse.de',
    install_requires=[
        'Flask>=0.10.1',
        'SQLAlchemy==0.8.2',
        ],
    tests_require=['nose'],
    packages=find_packages(exclude=['tests']),
    description='Eine kurze Beschreibung.',
    long_description = long_description,
    platforms='any',
    keywords = "different tags here",
    classifiers = [
        'Programming Language :: Python',
        'Development Status :: 4 - Beta',
        'Natural Language :: English',
        'Intended Audience :: Developers',
        'Operating System :: Linux',
        'Topic :: Software Development :: Libraries :: Application Frameworks',
        'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
        ],

    )
```

Fehler und Ausnahmen

Fehler gehören für Programmierer zum Alltag: Komplexe Computerprogramme laufen nur selten fehlerfrei, schon gar nicht in der Entwicklungsphase. Die möglichen auftretenden Fehler lassen sich allgemein in drei Arten unterteilen:

Arten von Programm-Fehlern

- **Syntax-Fehler** bewirken, dass ein Programm aufgrund eines „Grammatik-Fehlers“ vom Computer nicht in ausführbaren Maschinencode übersetzt werden kann. Derartige Fehler können aus unvollständigen Klammerpaaren, fehlenden Doppelpunkt-Zeichen am Ende einer `if`-Bedingung, ungültige Zeichen oder ähnlichem bestehen.

  Enthält ein Programm Syntax-Fehler, so werden diese beim Versuch eines Programmaufrufs angezeigt, und das Programm startet nicht.

  Die zwei geläufigsten Syntax-Checker für Python-Code sind pylint und pyflakes; zudem gibt es einen „Style“-Checker namens pep8, der prüft, ob die offiziellen Empfehlung für das Aussehen von Python-Code eingehalten werden (beispielsweise keine Zeilen mit mehr als 80 Zeichen auftreten). Diese zusätzlichen Werkzeuge können folgendermaßen installiert werden:

  ```
  pip3 install pylint
  pip3 install pyflakes
  pip3 install pep8
  ```

  Gibt man nach der Installation `pylint skriptname.py` ein, so werden einerseits darin möglicherweise enthaltene Syntax-Fehler aufgelistet, andererseits werden Empfehlungen zur Verbesserung der Code-Syntax gegeben – unter anderem wird darauf hingewiesen, wenn eine Funktionsdefinition keinen Docstring enthält.

  Gibt man `pep8 skriptname.py` ein, so werden ebenfalls Verbesserungsvorschläge angezeigt, wie der enthaltene Python-Code in einer besser lesbaren Form geschrieben werden kann.

- **Laufzeit-Fehler** treten auf, wenn ein Programm versucht, eine ungültige Operation durchzuführen, beispielsweise eine Division durch Null oder ein Öffnen einer nicht vorhandenen Datei. Laufzeit-Fehler treten also erst auf, wenn das Programm (in der Regel ohne Fehlermeldung) bereits läuft.

Laufzeit-Fehler können in Python allgemein mittels *try-except*-Konstrukten abgefangen werden.

- **Logische Fehler** treten auf, wenn ein Programm zwar (anscheinend) fehlerfrei funktioniert, jedoch andere Ergebnisse liefert als erwartet. Bei solchen Fehlern liegt das Problem also an einem Denkfehler des Programmierers.

  Logische Fehler sind oft schwer zu finden; am besten hilft hierbei ein Debugger, mit dem man den Programmablauf Schritt für Schritt nachvollziehen kann (siehe Abschnitt *pdb – der Python-Debugger*).

Tritt in einem Python-Programm ein Fehler auf, der nicht von einer entsprechenden Routine abgefangen wird, so wird das Progarmm beendet und ein so genannter „Traceback“ angezeigt. Bei dieser Art von Fehlermeldung, die man von unten nach oben lesen sollte, wird als logische Aufruf-Struktur angezeigt, welche Funktion beziehungsweise Stelle im Programm den Fehler verursacht hat; für diese Stelle wird explizit der Dateiname und die Zeilennummer angegeben.

`try`, `except` und `finally` – Fehlerbehandlung

Mit dem Schlüsselwort `try` wird eine Ausnahmebehandlung eingeleitet: Läuft der `try`-Block nicht fehlerfrei durch, so kann der Fehler mittels einer `except`-Anweisung abgefangen werden; in diesem Fall werden alle Anweisungen des jeweiligen `except`-Blocks ausgeführt.

Optional kann neben einer `try` und einer beziehungsweise mehreren `except`-Anweisungen auch eine `else`-Anweisung angegeben werden, die genau dann ausgeführt wird, wenn die `try`-Anweisung *keinen* Fehler ausgelöst hat:

```
try:
    # Diese Anweisung kann einen FileNotFoundError auslösen:
    file = open('/tmp/any_file.txt')

except FileNotFoundError:
    print("Datei nicht gefunden!")

except IOError:
    print("Datei nicht lesbar!")

else:
    # Datei einlesen, wenn kein Fehler augetreten ist:
    data = file.read()

finally:
    # Diese Anweisung in jedem Fall ausführen:
        file.close()
```

Zusätzlich zu `try` und `except` kann man optional auch einen `finally`-Block angeben; Code, der innerhalb von diesem Block steht, wird auf alle Fälle am Ende der Ausnah-

mebehandlung ausgeführt, egal ob der `try`-Block fehlerfrei ausgeführt wurde oder eine Exception aufgetreten ist.

Das `with`-Statement

Ausnahmebehandlungen sind insbesondere wichtig, wenn Dateien eingelesen oder geschrieben werden sollen. Tritt nämlich bei der Bearbeitung ein Fehler auf, so muss das `file`-Objekt trotzdem wieder geschlossen werden. Mit einer „klassischen“ `try`- und `finally`-Schreibweise sieht dies etwa wie folgt aus:

```
# Datei-Objekt erzeugen:
myfile = open("filename.txt", "r")

try:
    # Datei einlesen und Inhalt auf dem Bildschirm ausgeben:
    content = myfile.read()
    print(content)
finally:
    # Dateiobjekt schließen:
    f.close()
```

Diese verhältnismäßig häufig vorkommende Routine kann kürzer und eleganter auch mittels eines `with`-Statements geschrieben werden:

```
with open("filename.txt", "r") as myfile:
    content = myfile.read()
    print(content)
```

Hierbei versucht Python ebenfalls, den `with`-Block ebenso wie einen `try`-Block auszuführen. Die Methode ist allerdings wesentlich „objekt-orientierter“: Durch die im `with`-Statement angegebene Anweisung wird eine Instanz eines Objekts erzeugt, in dem obigen Beispiel ein `file`-Objekt; innerhalb des `with`-Blocks kann auf dieses Objekt mittels des hinter dem Schlüsselwort `as` angegebenen Bezeichners zugegriffen werden.

In der Klasse des durch das `with`-Statement erzeugten Objekts sollten die beiden Methoden `__enter__()` und `__exit()__` definiert sein, welche Anweisungen enthalten, die unmittelbar zu Beginn beziehungsweise am Ende des `with`-Blocks aufgerufen werden. Beispielsweise besitzen `file`-Objekte eine `__exit__()`-Methode, in denen die jeweilige Datei wieder geschlossen wird.

`raise` – Fehler selbst auslösen

Mit dem Schlüsselwort `raise` kann eine Ausnahme an der jeweiligen Stelle im Code selbst ausgelöst werden. Dies ist unter anderem nützlich, um bei der Interpretation einer Benutzereingabe fehlerhafte Eingaben frühzeitig abzufangen.

Wird von einem Benutzer beispielsweise anstelle einer Zahl ein Buchstabe eingegeben, so kann dies beim Aufruf der weiterverarbeitenden Funktion mit großer Wahrscheinlichkeit

zu Fehlern führen. Da der Fehler jedoch bei der Eingabe entstanden ist, sollte auch an dieser Stelle die entsprechende Fehlermeldung (ein `ValueError`) ausgelöst werden.

Debugging, Logging und Testing

Im folgenden Abschnitt werden Methoden vorgestellt, die beim Auffinden, Beheben und Vermeiden von Fehlern hilfreich sind.

pdb – Der Python-Debugger

Ein Debugger wird verwendet, um ein fehlerhaftes Programm Schritt für Schritt ablaufen zu lassen, um den Fehler schnell ausfindig machen zu können; er kann ebenso verwendet werden, um die Funktionsweise ein unbekanntes Programms leichter nachvollziehen zu können, indem man sieht, welche Funktionen im Laufe des Programms nacheinander aufgerufen werden.

Der Python-Debugger pdb kann in einer Shell folgendermaßen aufgerufen werden:

```
pdb3 scriptfile.py
```

Nach dem Aufruf erscheint der `pdb`-Eingabeprompt `(Pdb)`. Hier können unter anderem folgende Anweisungen eingegeben werden:

- `help` (oder kurz: `h`):

  Mit `help` wird eine Übersicht über die wichtigsten Funktionen von `pdb` angezeigt.

- `step` (oder kurz: `s`):

  Mit `step` wird die aktuelle Zeile ausgeführt; der Debugger hält allerdings bei der nächst möglichen Stelle an (beispielsweise einem neuen Funktionsaufruf).

- `p` und `pp`:

  Mit `p` wird der angegebene Ausdruck ausgewertet und das Ergebnis angezeigt; beispielsweise gibt `p variablenname` den Wert der angegebenen Variablen zum aktuellen Zeitpunkt im Programm an. Mit `pp` wird das Ergebnis in „pretty print“-Form ausgegeben.

- `list` (oder kurz: `l`):

  Mit `list` wird der Code-Bereich ausgegeben, in dem man sich beim Debuggen aktuell im Programm befindet. Standardmäßig (und meist völlig ausreichend) werden elf Zeilen an Code ausgegeben, wobei sich die aktuelle Code-Zeile in der Mitte befindet und mit `->` gekennzeichnet ist.

- `return` (oder kurz: `r`):

  Mit `return` wird das Programm bis zum Ende der aktuellen Funktion weiter ausgeführt.

- `break` (oder kurz: `b`):

  Wird `break` ohne weiteres Argument aufgerufen, gibt es alle aktuellen Haltepunkte („Breakpoints“) und ihre laufende Nummer aus. Diese können ebenfalls mittels `break` manuell gesetzt werden:

  - Wird `break` mit einer ganzzahligen Nummer als Argument aufgerufen, so wird ein Breakpoint an dieser Stelle im Quellcode des Programms gesetzt; das heißt, der Debugger hält an, wenn diese Stelle erreicht wird.
  - Wird `break` mit einem Funktionsnamen als Argument aufgerufen, so wird ein Breakpoint bei dieser Funktion gesetzt, das heißt, der Debugger hält jedes mal an, wenn diese Funktion aufgerufen wird.

- `clear` (oder kurz: `cl`):

  Mit `clear nummer` kann der Breakpoint mit der angegebenen Nummer wieder gelöscht werden.

- `continue` (oder kurz: `c`):

  Mit `continue` wird das Programm bis zum Ende weiter ausgeführt, außer ein mit `break` gesetzter Breakpoint wird erreicht.

- `run` beziehungsweise `restart`:

  Mit `run` beziehungsweise `restart` wird das zu debuggende Programm von Neuem gestartet. Wurde das Programm seit dem letzten Aufruf von `pdb` verändert, wird es neu geladen; Breakpoints bleiben dabei erhalten.

- `exit` beziehungsweise `quit`:

  Mit `exit` oder `quit` wird der Debugger beendet.

Zusätzlich lassen sich im Debugger auch Python-Statements ausführen; hierzu muss die jeweilige Eingabe-Zeile lediglich mit einem Ausrufezeichen (`!`) begonnen werden.

Eine weitere, oftmals sogar bessere Nutzungs-Variante ist es, den Debugger nicht direkt aufzurufen, sondern ihn erst ab einer bestimmten Stelle aus dem Programm heraus zu starten. Hierzu kann man folgendermaßen vorgehen:

```
import pdb

# ... anderer Python-Code ...

# Debugging ab hier beginnen:
pdb.set_trace()

# ... anderer Python-Code ...
```

Nach dem Importieren des `pdb`-Moduls kann man an einer beliebigen Stelle, auch innerhalb einer Funktion, mit `pdb.set_trace()` den Debugger starten. Auch bei dieser Benutzungsweise erscheint der `pdb`-Eingabeprompt, wo wiederum die obigen Anweisungen genutzt werden können; das Debuggen beginnt dann allerdings erst an der „spannenden“ Stelle.

`logging` – Arbeiten mit Logdateien

Logdateien können ebenso wie ein Debugger genutzt werden, um Fehler in einem Programm zu finden. Hierzu nutzt man im Quellcode des Programms Anweisungen, die große Ähnlichkeit mit simplen `print()`-Anweisungen haben. Die Ausgabe wird allerdings üblicherweise nicht auf dem Bildschirm, sondern mit einer Standard-Formatierung in eine Logdatei geschrieben.[1]

In Python kann ein Logging einfach mit Hilfe des *logging <https://docs.python.org/3/library/logging.html>*-Moduls umgesetzt werden. Beispielsweise kann man ein Logging in einer interaktiven Interpreter-Sitzung folgendermaßen aktivieren:

```
import logging

# Basis-Einstellungen festlegen:
logging.basicConfig(level=logging.INFO)

# Logger zur aktuellen Sitzung erstellen:
logger = logging.getLogger(__name__)

# Logger-Nachricht erzeugen:
logger.info("Los geht's!")
# Ergebnis:INFO:__main__:Los geht's!
```

Über die Angabe des Log-Levels wird festgelegt, wie dringlich eine Nachricht ist. Im `logging`-Modul sind dabei folgende Werte festgelegt:

CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

Die einzelnen Stufen können mittels `logger.info()`, `logger.warning()`, `logger.error()` usw. unmittelbar genutzt werden. Ausgegeben werden derartige Nachrichten

[1] Gegenüber einfachen `print()`-Anweisungen, die ebenfalls beispielsweise zur Ausgabe von Variablenwerten zu einem bestimmten Zeitpunkt genutzt werden können, haben Logger als Vorteil, nach dem ‚Debuggen‘ zwingend wieder aus dem Code entfernt werden zu müssen; zudem stehen für Logger verschiedene Dringlichkeits-Stufen zur Verfügung, so dass die Ausgabe der Logging-Nachrichten nur dann erfolgt, wenn die jeweilige Stufe (mittels einer Einstellung) aktiviert wird.

immer dann, wenn ihr Dringlichkeitswert über dem in der Basis-Einstellung festgelegten Level liegt.

Meist werden Logger nicht in interaktiven Interpreter-Sitzungen, sondern innerhalb von Quellcode-Dateien in Verbindung mit einer Logdatei genutzt. Hierfür kann die Basis-Konfiguration beispielsweise so aussehen:

```
import logging

import modul1
import modul2

# Basis-Einstellungen festlegen:
logging.basicConfig(filename='log.txt',
                    format='%(levelname)s: %(message)s',
                    level=logging.DEBUG)

# Logger-Nachricht erzeugen:
logger.info("Los geht's!")
```

In diesem Fall wurden bei der Festlegung der Basis-Einstellungen zusätzlich eine Logdatei und eine Standardfomat angegeben. Wird das Programm aufgerufen, so wird hierdurch in der angegebenen Logdatei folgender Eintrag erzeugt:

```
INFO: Los geht's!
```

Wird die obige Konfiguration in der Basis-Datei eines Programms vorgenommen, das als Ankerpunkt für weitere Module dient, so genügt innerhalb dieser Module bereits die Anweisung `import logging` zu Beginn der jeweiligen Datei, um innerhalb des Moduls ebenfalls mittels `logger.info(nachricht)` Einträge in die Logdatei des Basis-Programms schreiben zu können.

Da mittels Lognachrichten auch, ebenso wie mit *print()*, Variablenwerte ausgegeben werden können, kann die Verwendung von Logdateien in vielen Fällen sogar einen Debugger ersetzen.

doctest – Testen mittels Docstrings

Zu Beginn eines jeden Funktionsblocks sollte mittels dreifachen Anführungszeichen ein kurzer *Docstring* geschrieben werden, welcher eine kurze Beschreibung der Funktion enthält. Ein solcher Docstring kann ebenfalls ein kurzes Code-Beispiel enthalten, wie die Funktion angewendet wird und welches Ergebnis die Funktion liefert.

```
def power(base, n):
    """
    Berechne die n-te Potenz des Basis-Werts.

    >>> power(5, 3)
    125
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
    :param base: Basiswert  (int oder float)
    :param n:    Exponent   (int oder float)
    :returns:    Potenzwert (int oder float)
    """
    return base ** n
```

Beim Schreiben von Doctests werden Zeilen, die normalerweise direkt im Python-Interpreter eingegeben werden, mit >>> eingeleitet; in der darauffolgenden Zeile wird dann eingegeben, welches Ergebnis beim Aufruf der vorherigen Zeile erwartet wird. Stimmt beim Durchlaufen der Doctests ein tatsächliches Ergebnis nicht mit dem erwarteten Ergebnis überein, so schlägt der jeweilige Test fehl, und eine entsprechende Fehlermeldung wird angezeigt.

Das Schreiben von so gestalteten Docstrings macht einerseits Code nachvollziehbarer; andererseits die integrierten Code-Beispiele auch ein Testen der jeweiligen Funktionen. Dazu muss das Paket doctest importiert werden. Bei einem Modul, das ausschließlich Hilfsfunktionen enthält (also üblicherweise nur importiert, aber nicht ausgeführt wird, kann folgende Syntax verwendet werden:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=1)
```

Werden diese Zeilen an das Ende des zu testenden Moduls geschrieben, so kann man anschließend `python3 modulname.py` aufrufen, um die Tests zu aktivieren; wird Das Modul hingegen nur importiert, so wird der Code-Abschnitt ignoriert.

Alternativ können Doctests auch direkt durch den Aufruf des Interpreters aktiviert wrden:

```
python3 -m doctest modulname.py -v
```

Hierbei wird mittels der Interpreter-Option `-m` das `doctest`-Modul geladen, zudem werden mittels der Option `-v` („verbose“) ausführliche Ausgabe-Informationen angezeigt.

Doctests eignen sich nur für verhältnismäßig einfache Tests, in denen nur eine geringe Anzahl von Tests je Funktion durchgeführt werden und auch keine umfangreiche Vorbereitung der Einsatz-Umgebung notwendig ist; dies würde die Docstrings allzu umfangreich und die Code-Dateien damit zu unübersichtlich machen. Eine bessere Alternative bieten an dieser Stelle Unit-Tests.

`unittest` – Automatisiertes Testen

Beim Schreiben von Unit-Tests mit Hilfe des unittest-Pakets wird zu jedem Modul `modulname.py` ein entsprechendes Test-Modul `test_modulname.py`, mit dessen Hilfe welche die im Hauptmodul enthaltenen Funktionen getestet werden können. Alle diese so genannten Unit Tests sollten voneinander unabhängig sein.

Da manche Funktionen oder Module im normalen Betrieb eine bestimmte Umgebung benötigen, beispielsweise einen aktiven Webserver, eine Datenbank, oder eine geöffnete Beispieldatei, können innerhalb der Test-Module mittels der Funktionen `setup()` und `teardown()` solche Umgebungen bereitgesetellt werden; diese beiden Funktionen werden bei jedem Test aufgerufen und erzeugen beziehungsweise bereinigen die benötigte Umgebung.

Ein Test-Funktionen einer Unitt-Test-Datei beginnen jeweils mit mit `test_`, gefolgt vom Namen der zu testenden Funktion. Um Klassen zu testen, werden in der Unit-Test-Datei ebenfalls Klassen definiert, deren Namen sich aus der Zeichenkette `Test_` und und den eigentlichen Klassennamen zusammensetzt. Diese Klassen haben `unittest.TestCase` als Basisklasse.

Eine Unit-Test-Klasse kann somit etwa folgenden Aufbau haben:

```
import unittest
from modulname import KlassenName

class Test_KlassenName(unittest.TestCase):

    def setUp(self):
        pass

    def test_funktionsname1(self):
        ...

    def test_funktionsname2(self):
        ...

    ...

    def tearDown(self):
        pass
```

Die einzelnen Test-Funktionen enthalten – neben möglichen Variablen-Definitionen oder Funktionsaufrufen – stets so genannte Assertions, also „Behauptungen“ oder „Hypothesen“. Hierbei wird jeweils geprüft, ob das tatsächliche Ergebnis einer `assert`-Anweisung mit dem erwarteten Ergebnis übereinstimmt. Ist dies der Fall, so gilt der Test als bestanden, andererseits wird ein `AssertionError` ausgelöst.

In Python gibt es, je nach Art der Hypothese, mehrere mögliche `assert`-Anweisungen:

- Mit `assertEqual(funktion(), ergebnis)` kann geprüft werden, ob der Rückgabewert der angegebenen Funktion mit dem erwarteten Ergebnis übereinstimmt.
- Mit `assertAlmostEqual(funktion(), ergebnis)` kann bei numerischen Auswertungen geprüft werden, ob der Rückgabewert der angegebenen Funktion bis auf Rundungs-Ungenauigkeiten mit dem erwarteten Ergebnis übereinstimmt.
- ...

Um Unit-Tests zu starten, kann die Test-Datei am Ende um folgende Zeilen ergänzt werden:

```
if __name__ == '__main__':
    unittest.main()
```

Gibt man dann `python3 test_modulname.py` ein, so werden durch die Funktion `unittest.main()` alle in der Datei enthaltenen Tests durchlaufen. Als Ergebnis wird dann angezeigt, wieviele Tests erfolgreich absolviert wurden und an welcher Stelle gegebenenfalls Fehler aufgetreten sind.

Test-Automatisierung mit `nose`

Das Programm `nose` vereinfacht das Aufrufen von Unit-Tests, da es automatisch alle Test-Funktionen aufruft, die es im aktuellen Verzeichnis mitsamt aller Unterverzeichnisse findet; eine Test-Funktion muss dazu lediglich in ihrem Funktionsnamen die Zeichenkette `test` enthalten.

Um Tests mittels `nose` zu finden und zu aktivieren, genügt es in einer Shell in das Test-Verzeichnis zu wechseln und folgende Zeile einzugeben:

```
nosetest3
```

Bei Verwendung von `nose` erübrigt sich also das Schreiben von Test-Suits. Wird `nosetests3 --pdb` aufgerufen, so wird automatisch der Python-Debugger `pdb` gestartet, falls ein Fehler auftritt.

Design Patterns

An dieser Stelle sollen einige hilfreiche Entwurfsmuster („Design Patterns“) und ihre Implementierungen in Python3 vorgestellt werden.[1] Derartige Entwicklungsmuster kann man als „Entwicklungshilfen“ ansehen, da sie bewährte Code-Strukturen für bestimmte Problem-Typen bieten.bestimmte Problem-Typen bieten.

Erzeugungsmuster

Als Erzeugungsmuster („Creational Patterns“) werden Entwurfsprinzipien bezeichnet, die für das Erstellen neuer Objekte hilfreich sein können.

Factory

Als „Factories“ werden Hilfsobjekte bezeichnet, deren Aufgabe es ist, die Erzeugung eines Objekts von seiner Verwendung zu trennen. Man unterscheidet im Allgemeinen zwischen der „Factory Method“ und der „Factory Class“ als zwei verschiedenen Möglichkeiten, dieses Prinzip zu implementieren.

Factory Method

Bei einer „Factory Method“ wird ein neues Objekt durch den Aufruf einer Funktion erzeugt und von dieser als Ergebnis zurückgegeben. Die erzeugende Funktion erstellt das neue Objekt dabei in Abhängigkeit vom Kontext. Beispielsweise kann nach diesem Prinzip eine Funktion `read_file()` ein Datei-Reader-Objekt in Abhängigkeit vom Dateityp bzw. der Datei-Endung des angegebenen Pfads generieren:

```
# Factory-Method-Beispiel

class CSV_Reader():

    def __init__(self, path):
        pass
```

(continues on next page)

[1] Eine allgemeine, nicht Python-spezifische Übersicht über Design Patterns gibt es unter anderem auf Wikipedia.

(Fortsetzung der vorherigen Seite)

```
def file_reader(path):

    # Todo: Check if file exists

    if path.endswith(".csv"):
        return CSV_READER(path)
    else:
        return None

csv_reader = file_reader('test.csv')
```

Soll durch eine Factory Method eine direkte Instanziierung einer Klasse verhindert werden, so kann die Definition dieser Klasse auch innerhalb der Funktionsdefinition erfolgen.

Factory Class

Bei einer „Factory Class“ wird eine Factory-Methode (oder mehrere davon) zu einer *Klasse* zusammengefasst. Bei Verwendung dieses Patterns kann man beispielsweise eine Klasse names ConnectionCreator mit einer Methode `build_connection(connection_type)` erstellen, die je nach angegebenem Protokoll-Typ eine SSH- oder FTP-Verbindung zu einem Webserver aufbaut. Strukturell könnte der Code dann folgendermaßen aussehen:

```
# Factory-Class-Beispiel

# Connection-Klassen:

class SSH_Connection():

    def __init__(self, path):
        pass

class FTP_Connection():

    def __init__(self, path):
        pass


# Factory-Klasse:

class ConnectionCreator():

    def __init__(self, path):
        self.path == path

    def build_connection(self, connection_type, path):

        if connection_type == "HTML":
            return HTML_Connection(path)
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
        elif connection_type == "SSH":
            return SSH_Connection(path)

        else:
              return None
```

Nach dem gleichen Prinzip denkbar wäre beispielsweise auch eine DatabaseConnection-Klasse, die eine Verbindung zu einer bestehenden Datenbank herstellen kann, oder gegebenenfalls auch eine neue Datenbank anlegen kann.

Abstract Factory

In der Programmierung werden bisweilen „abstrakte“ Klassen definiert. Diese geben zwar bereits strukturelle Prinzipien vor, es können allerdings noch keine Instanzen einer solchen Klasse erzeugt werden, da konkrete Ausprägungen noch nicht festgelegt sind. Beispielsweise könnte eine abstrakte Klasse ein „Kraftfahrzeug“ sein, das ganz allgemein Methoden wie „Motor starten“ oder „Bremse betätigen“ bereit stellt. Jeder reelle Kraftfahrzeug-Typ, der auf dieser Klasse via *Vererbung* aufbaut, implementiert diese Funktionen, allerdings konkretisiert auf die konkrete Ausprägung.

Bei Verwendung einer „Abstract Factory“ wird entsprechend ein Factory-Typ mit strukturellen Prinzipien vorgegeben, aus dem wiederum konkrete Factory-Klassen (mittels Vererbung) hervorgehen können.[2] Dieses Pattern kann beispielsweise für ein Computer-Strategiespiel wie 0.A.D genutzt werden, so dass „Gebäude-Typ“ je nach Kultur und Entwicklungsstufe zwar ähnliche, aber nicht komplett identische Objekte generieren kann.

Builder

Das „Builder“-Pattern kann verwendet werden, wenn ein Objekt schrittweise aus einzelnen Komponenten zusammengestellt werden muss. Die einzelnen Komponenten werden dabei durch Factory-Methoden einer (oder mehrerer) „Builder“-Klassen erzeugt. Die Builder-Methoden werden wiederum von einem „Director“-Objekt in der gewünschten Reihenfolge aufgerufen.

Das gewünschte Objekt als Ganzes wird also über den Direktor in Auftrag gegeben, der die Anfrage an den passenden Builder weiter reicht. Ist das Objekt erstellt, kann der Director es wiederum beim Builder abholen und als Ergebnis zurückgeben. Während die einzelnen Builder wiederum „Factories“ darstellen, ist der Director ein steuerndes Element, das kontext-bezogen den relevanten Builder auswählt und gewissermaßen „nach Rezept“ nacheinander dessen Methoden aktiviert.

[2] Konkrete Factories können konkrete Objekte generieren, abstrakte Faktories hingegen nicht. Zu den konkreten objekten können zwar separat abstrakte Klassen definiert werden, eine abstrakte Factory kann allerdings nicht einmal diese generieren, da sie selbst eine abstrakte Klasse darstellt: Eine Abstract Factory kann nicht instanziiert werden, sie gibt also nur den strukturellen Aufbau einer konkreten Factory vor.

Prototype

Mittels eines „Prototyps“ kann ein neues Objekt erstellt werden, indem ein bestehendes Objekt als Startpunkt verwendet wird. Um einen Prototypen zu erzeugen, muss also zunächst eine exakte Kopie eines bestehenden Objekts erzeugt werden.

In Python ist dies einfach mittels der Funktion `deepcopy()` aus dem Paket `copy` der Standard-Library möglich.

Singleton

Als Singleton bezeichnet man ein Objekt, das innerhalb eines laufenden Programms nur in einer Ausprägung („Instanz“) existieren darf; beispielsweise ist bei jedem Betriebsystem mit grafischer Oberfläche genau ein Window-Manager in Betrieb. Zugleich muss das Singleton-Objekt unter Umständen für viele Stellen zugriffsbereit sein.

Singletons stellen also eine Art von klar definierten „Access Points“ dar, auf die von mehreren Clienten aus zugegriffen werden kann. Ein solches Objekt könnte zwar prinzipiell auch mittels einer globalen Variable initiiert werden, jedoch könnten dabei immer noch mehrere Instanzen des Objekts existieren – man hätte dann zwar das gleiche, aber nicht das selbe Objekt. Zudem soll die Klasse des Grundobjekts durch die Erstellung von Unterklassen erweiterbar sein.

Singleton-Klasse

In Python kann eine Singleton-Klasse folgendermaßen als Klasse implementiert werden:

```
class Singleton(object):
    def __new__(cls):
        if not hasattr(cls, 'instance'):
            cls.instance = super().__new__(cls)
        return cls.instance
```

Wird ein solches Objekt initiiert, so wird es nur dann eine neue Instanz des Objekts erzeugt, falls noch keine solche existiert; andernfalls gibt die Initiierung die bereits existierende Instanz als Ergebnis zurück. Auf diese Weise kann man von beliebiger Stelle aus auf das Singleton zugreifen, indem man eine neue Instanz des Singletons erzeugt:

```
# Ein neues Singleton erzeugen:
singleton_1 = Singleton()

# Das existierende Singleton an anderer Stelle nutzen:
singleton_2 = Singleton()
```

Jedes Objekt, das ein Singleton darstellen soll, kann damit der obigen Implementierung als Unterklasse eines Singletons definiert werden:

```
class Any_Singleton_Object(Singleton):
    """
    A Class for a Singleton Object.
    """

    # Class methods and attributes..
```

Bei der Initiierung eines solchen Objekts wird aufgrund der geerbten `__new__()`-Funktion nur dann ein neues Objekt (mit allen „Standardeinstellungen“) erstellt, falls ein solches noch nicht existiert. Ansonsten wird dieses mit all seinen Methoden und Attributen genutzt.

Singleton-Module

Die Initiierung eines Objekts ist stets mit etwas Rechenaufwand verbunden. Soll auf ein Singleton häufig und möglichst schnell zugegriffen werden und ist keine Aufgliederung des Singletons in mehrere mögliche Unterklassen nötig, so kann anstatt der oben beschriebenen Klasse auch ein Singleton-Modul erzeugt werden. Dieses Modul, das den Namen des Singletons (in Kleinbuchstaben) als Dateinamen (mit Endung `.py`) trägt, bekommt „Methoden“ als Funktionen und „Attribute“ als Variablen auf Modulebene zugewiesen – d.h. in diesem Modul werden keine Klassen angelegt.

Da Module nach erstmaligem Importieren durch `import modulname` stets nur in Form einer Referenz genutzt werden, kann auf die gewünschten Singleton-Methoden unmittelbar mittels `modul.funktionsname()` und die gewünschten Attribute mittels `modul.variable` zugegriffen werden.

Strukturmuster

Als Strukturmuster („Structural Patterns“) werden Entwurfsprinzipien bezeichnet, die für das Zusammenwirken mehrerer Objekte im Programm nützlich sein können.

Adapter

Composite

Facade

Ein Facade-Pattern kann genutzt werden, um einem Benutzer ein einfaches, intuitiv nutzbares Interface zu bieten, so dass sich dieser nicht mit den Schnittstellen der einzelnen Klassen eines Programms auseinander setzen muss. In einer solchen „Fassade“ eines Programms sollen also keine neuen Funktionen hinzu kommen, es soll vielmehr der Zugriff auf die eigentlichen Programm-Funktionen erleichtert werden.

Model-View-Controller

Das Prinzip „Model-View-Controller“ soll dabei helfen, die eigentliche Logik des Programms (das „Modell“) von der Datenausgabe (dem „View“) zu trennen.

Der „Controller“ vermittelt als Schnittstelle zwischen diesen Ebenen:

- Er soll die Eingabe des Benutzers, die ebenfalls in der View-Ebene erfolgt, entgegennehmen und an das Modell weiterleiten.
- Er soll, nachdem die Eingabe in der Modell-Ebene verarbeitet wurde, die resultierende Ausgabe wieder an die View-Ebene weiterleiten.

Der Controller wirkt wie ein „Kleber“ zwischen der Modell- und der View-Schicht; man sagt entsprechend, der Kontroller solle „dünn“ sein, also nur die für die Vermittlung absolut nötigen Funktionen beinhalten.
Die Modell-Ebene hingegen soll „schlau“ sein, an dieser Stelle sollte also die eigentliche Programm-Logik liegen.
Die View-Ebene wiederum soll „dumm“ sein, also nur die zur Entgegennahme des Inputs und zur Darstellung der Ausgabe nötigen Funktionen (und keine weitere Logik) beinhalten; beispielsweise sollte aus der View-Ebene heraus kein Zugriff auf eine Datenbank erfolgen.

Der Vorteil dieses Entwicklungs-Musters liegt darin, dass das eigentliche Programm von der Benutzeroberfläche abstrahiert wird. Etliche Linux-Programme, aber beispielsweise auch den Python-Interpreter *Ipython*, gibt es dadurch sowohl als text-basierte Anwendungen für die Shell wie auch als Variante mit einer eigenen graphischen Bedienoberfläche.

Verhaltensmuster

Memento

- Memento (Wikipedia, de.)

Observer

Das Observer-Muster besteht darin, dass ein bestimmtes Objekt als „Subjekt“ deklariert wird, dass von anderen Objekten „beobachtet“ wird. Das Subjekt verwaltet dabei eine Liste aller Objekte, die es beobachten. Ändert sich eine bestimmte Eigenschaft des Subjekts, dann benachrichtigt es darüber alle Beobachter-Objekte, indem es eine deren Methoden aufruft.

Visitor

... to be continued ...

Links

Scientific Python

Unter dem Überbegriff „Scientific Python“ werden in Python einige Pakete zusammengefasst, die speziell für wissenschaftliches Arbeiten entwickelt wurden.

Mathematik mit Standard-Modulen

In jeder Python-Installation ist das *Standard-Modul math* enthalten. Es enthält zahlreiche Funktionen, die aus dem Python-Interpreter einen umfangreichen und programmierbaren Taschenrechner machen.

Das `math`-Modul wird meistens in folgender Form eingebunden:

```
import math as m
```

Anschließend kann so das `math`-Modul unter der Bezeichnung `m` angesprochen werden; dies spart Tipp-Arbeit. Als einzige (verkraftbare) Einschränkung hat dies zur Folge, dass keine andere Variable mit `m` benannt werden darf.

Das `math`-Modul enthält unter anderem folgende Funktionen, die jeweils auf einzelne Zahlenwerte angewendet werden können:

`m.pow(zahl,n)`	n-te Potenz einer Zahl (oder n-te Wurzel wenn $0 < n < 1$)
`m.sqrt(zahl)`	Quadrat-Wurzel einer positiven Zahl
`m.exp(x)`	Exponentialfunktion e^x mit Exponent x
`m.log(x, a)`	Logarithmusfunktion $\log_a(x)$ von x zur Basis a
`m.radians(winkelwert)`	Winkelumrechnung von Grad auf Radians
`m.degrees(winkelwert)`	Winkelumrechnung von Radians auf Grad
`m.sin(x)`, `m.cos(x)` und `m.tan(x)`	Trigonometrische Funktionen für x-Werte in Radians
`m.asin(x)`, `m.acos(x)` und `m.atan(x)`	Umkehrfunktionen der trigonometrischen Funktionen (Ergebnisse in Radians)
`math.floor(zahl)`	Nächst kleinerer `int`-Wert $n \leq q$ einer Zahl q
`math.ceil(zahl)`	Nächst größerer `int`-Wert $n \geq q$ einer Zahl q

Zudem sind im `math`-Modul die Konstanten `m.pi` und `m.e` für die Kreiszahl $\pi \approx 3,14$ beziehungsweise die Eulersche Zahl $e \approx 2,71$ vordefiniert.

Neben dem `math`-Modul existieren weiter für mathematische Zwecke nützliche Module, beispielsweise *cmath* für das Rechnen mit komplexen Zahlen oder `functools` für das Anwenden von Operatoren und/oder Funktionen auf eine ganze Liste von Zahlen. Im folgenden werden einige Anwendungsbeispiele vorgestellt, die sich an den Aufgaben zur Arithmetik orientieren.

Primfaktor-Zerlegung

Um die Primfaktoren einer ganzen Zahl $n > 0$ zu bestimmen, kann man folgende Funktion nutzen (Quelle: StackOverflow):

```
def prim(n):
    '''Calculates all prime factors of the given integer.'''

    from math import sqrt

    pfactors = []
    limit = int(sqrt(n)) + 1
    check = 2
    num   = n

    if n == 1:
        return [1]

    for check in range(2, limit):
        while num % check == 0:
            pfactors.append(check)
            num /= check

    if num > 1:
        pfactors.append(num)

    return pfactors
```

Die Grund-Idee von diesem Algorithmus liegt darin, dass eine Zahl keinen Primzahl-Faktor haben kann, der größer ist als die Quadratwurzel dieser Zahl. Für $n = 100$ ist beispielsweise $\sqrt{100} = 10$ der größtmögliche Faktor, der eine Primzahl sein könnte.

- Andere Produkte mit größeren Faktoren $100 = 50 \cdot 2$ lassen sich zwar ebenfalls bilden, enthalten dann allerdings stets einen Faktor, der kleiner als die Quadrat-Wurzel der Original-Zahl ist.
- Die einzelnen möglichen Primfaktoren lassen sich finden, indem man nacheinander die Faktoren 2, 3 ... in aufsteigender Reihenfolge durchprobiert. Um eine mögliche Mehrfachheit einzelner Faktoren nicht außer Acht zu lassen, muss bei jedem gefundenen Faktor geprüft werden, ob sich die Original-Zahl gegebenenfalls auch mehrfach durch diesen dividieren lässt.

- Ist ein Faktor gefunden, so kann man die Original-Zahl durch diesen dividieren und den Algorithmus erneut mit dem resultierenden Divisions-Rest fortsetzen.

Die gefundenen Faktoren sind tatsächlich allesamt Primzahlen: Jeder andere Faktor ließe sich selbst als Produkt jeweils kleinerer Primzahlen darstellen. Die Teilbarkeit durch diese Zahlen wurde im Lauf des Algorithmus allerdings bereits geprüft; der jeweilige Divisionsrest, mit dem der Algorithmus fortfährt, enthält diese Faktoren nicht mehr.

Mittels der obigen Funktion kann nun die Primzahl einer Zahl oder eines Zahlenbereichs folgendermaßen berechnet werden:

```
# Einzelne Zahl in Primfaktoren zerlegen:
prim(2017)

# Zahlenbereich in Primfaktoren zerlegen:
for n in range(1,1000):
        print(prim(n))
```

Größter gemeinsamer Teiler und kleinstes gemeinsames Vielfaches

Beim Rechnen mit rationalen Zahlen, insbesondere beim Kürzen eines Bruchterms oder beim Multiplizieren zweier Brüche, ist es nützlich, den größten gemeinsamen Teiler zweier Zahlen zu finden. Hierzu wird bis heute ein Algorithmus verwendet, den bereits Euklid in der Antike entdeckt hat:

- Hat man zwei Zahlen a und b mit $a > b$, so ist der größte gemeinsame Teiler von a und b identisch mit dem größten gemeinsamen Teiler von $(a - b)$ und b. Man kann also wiederholt immer wieder die kleinere Zahl von der größeren abziehen und das Prinzip erneut anwenden, solange bis man beim größten gemeinsamen Teiler angekommen ist.

  Ist beispielsweise $a = 72$ und $b = 45$, so ist der größte gemeinsame Teiler dieser beider Zahlen identisch mit dem der Zahlen 45 und $(72 - 45) = 27$. erneut kann man die Differenz beider Zahlen bilden und erhält damit das Zahlenpaar 27 und $(45 - 27) = 18$; ein wiederholtes Anwenden des gleichen Prinzips liefert das zunächst das Zahlenpaar 18 und $(27 - 18) = 9$ und schließlich 9 und 9; der größte gemeinsame Teiler beider Zahlen ist somit 9.

- Ist die eine Zahl a wesentlich größer als die andere Zahl b, so müsste bei Anwendung des vorherigen Algorithmus sehr oft b von a subtrahiert werden. Ist beispielsweise $a = 968$ und $b = 24$, so ergäbe die erste Differenz $(968 - 24) = 944$, die zweite Differenz $(944 - 24) = 920$, usw. Bei Verwendung eines Computers ist es effektiver, anstelle der wiederholten Subtraktion eine Modulo-Rechnung vorzunehmen, also bei einer Division mit Rest nur auf den Divisionsrest zu achten. Hier ergibt 968 mod 24 den Wert 18; der Algorithmus kann somit unmittelbar mit 24 mod 18 = 6 fortgeführt werden und liefert schon im dritten Schritt als Ergebnis 18 mod 6 = 0. Der größte gemeinsame Teiler (6) wurde so in nur drei Rechenschritten bestimmt.

In Python lässt sich diese zweite, schnellere Variante des Euklidschen Algorithmus dank des Modulo-Operators `%` mit nur sehr wenig Code implementieren.

```
def gcd_simple(a, b):
    '''Quite simple implementation of Euclid's Algorithm.'''
    while b != 0:
        tmp = a % b
        a = b
        b = tmp
    return a
```

Dieser Code lässt sich noch weiter optimieren. Der Trick der folgenden Implementierung besteht darin, dass der Zuweisungsoperator = eine geringere Priorität besitzt als der Modulo-Operator, und somit erst die rechte Seite ausgewertet wird, bevor die Ergebnisse in die links angegebenen Variablen gespeichert werden; dies erspart das Speichern der Zwischenergebnisse in temporären Variablen:

```
def gcd(a, b):
    '''Return the greatest common divisor using Euclid's Algorithm.'''
    while b:
        a, b = b, a % b
    return a
```

Hat man den größten gemeinsamen Teiler gefunden, so kann auch das kleinste gemeinsame Vielfache zweier Zahlen einfach berechnet werden: Man multipliziert beide Zahlen miteinander und dividiert das Ergebnis anschließend durch den größten gemeinsamen Teiler. In Python könnte die entsprechende Funktion also folgendermaßen aussehen:

```
def lcm(a, b):
    '''Return lowest common multiple.'''
    return a * b / gcd(a, b)
```

Möchte man das kleinste gemeinsame Vielfache nicht nur zweier, sondern einer Liste mit beliebig vielen ganzen Zahlen ermitteln, so müsste man die obige `lcm()`-Funktion iterativ auf die einzelnen Elemente der Liste anwenden. Nutzt man hierfür die Funktion `reduce()` aus dem Standard-Modul *functools*, so lässt sich der Algorithmus folgendermaßen implementieren (Quelle: Stackoverflow):

```
import functools as ft

def lcmm(*args):
    '''turn lcm of args.'''
    return ft.reduce(lcm, args)


# Beispiel:

lcmm(6, 13, 27, 84)
# Ergebnis: 9828
```

... to be continued ...

`ipython` – eine Python-Entwicklungsumgebung

Ipython ist ein Interpreter-Frontend, das vielerlei nützliche Features zum interaktiven Entwickeln von Python-Projekten bietet. Er kann durch das gleichnamige Paket installiert werden:

```
sudo aptitude install ipython3 ipython3-qtconsole ipython3-notebook
```

Die beiden letzten Pakete ermöglichen es, Ipython mit einer graphischen Oberfläche oder als Anwendung in einem Webbrowser zu starten. Nach der Installation kann das Programm in einer Shell mittels `ipython3` oder als graphische Oberfläche mittels `ipython3 qtconsole` aufgerufen werden.

Gibt man in Ipython einen Python-Ausdruck ein und drückt `Enter`, so wird dieser ausgewertet. Besteht der Ausdruck beispielsweise aus einem einzelnen Variablennamen, so wird eine String-Version dieser Variablen ausgegeben.

Navigationshilfen

Ipython bietet gegenüber dem normalen Python-Interpreter für ein interaktives Arbeiten am Quellcode nicht nur ein Syntax-Highlighting, sondern auch folgende Funktionen:

Tab-Vervollständigung

Während man einen Python-Ausdruck oder einen Dateinamen schreibt, kann man diesen durch Drücken der `Tab`-Taste vervollständigen. Gibt es mehrere Möglichkeiten zur Vervollständigung, so werden diese aufgelistet; bei Nutzung der Qt-Konsole kann zwischen den einzelnen Möglichkeiten durch erneutes Drücken von `Tab` gewechselt und die Auswahl mit `Enter` bestätigt werden.

Gibt man einen Modulnamen ein, gefolgt von `.`, so werden durch Drücken von `Tab` alle Funktionen des Moduls aufgelistet; Attributs- und Funktionsnamen, die mit `_` oder `__` beginnen, werden allerdings als „privat“ gewertet und ignoriert. Gibt man zusätzlich den einleitenden Unterstrich ein, so lassen auch diese Attributs- beziehungsweise Funktionsnamen mit `Tab` auflisten und vervollständigen.

Weitere Tastenkürzel

In Ipython erleichtern folgende weitere Tastenkürzel das Arbeiten:

Navigation:

- `Ctrl u` Lösche die aktuelle Zeile
- `Ctrl k` Lösche vom Cursor bis zum Ende der aktuellen Zeile
- `Ctrl a` Gehe an den Anfang der aktuellen Zeile
- `Ctrl e` Gehe ans Ende der aktuellen Zeile

Code-History:

- ↑: History rückwärts durchsuchen
- ↓: History vorwärts durchsuchen
- `Ctrl` ↑: History rückwärts nach Aufruf durchsuchen, der genauso beginnt wie der aktuell eingegebene Text
- `Ctrl` ↓: History vorwärts nach Aufruf durchsuchen, der genauso beginnt wie der aktuell eingegebene Text
- `Ctrl r`: History rückwärts nach Aufrufen durchsuchen, die den eingegebenen Text als Muster beinhalten

Die letzten drei eingegeben Code-Zeilen können in Ipython zudem über die Variablen `_`, `__` und `___` referiert werden.

Startet man Ipython mittels `ipython3` als Shell-Anwendung, so kann (abhängig von den Einstellungen des jeweiligen Shell-Interpreters) nur begrenzt weit „zurück scrollen“. Ruft man in Ipython Funktionen auf, die eigentlich sehr lange Ausgaben auf dem Bildschirm erzeugen würden, so kann man diese unterdrücken, indem man an die Anweisung ein `;` anhängt. Dies hat keine Auswirkung auf die Anweisung selbst, hilft aber dabei, die Interpreter-Sitzung übersichtlich zu halten.

Informationen zu Objekten

Gibt man vor oder nach einem Variablennamen ein `?` an, so werden ausführliche Informationen zur jeweiligen Variable angezeigt. Setzt man das Fragezeichen vor oder nach einen Funktions- oder Klassennamen, so wird auch der Docstring der jeweiligen Funktion oder Klasse angezeigt:

```
print?
```

```
# Ergebnis:
# Type:        builtin_function_or_method
# String form: <built-in function print>
# Namespace:   Python builtin
# Docstring:
# print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

# Prints the values to a stream, or to sys.stdout by default.
# Optional keyword arguments:
# file:  a file-like object (stream); defaults to the current sys.stdout.
# sep:   string inserted between values, default a space.
# end:   string appended after the last value, default a newline.
# flush: whether to forcibly flush the stream.
```

Schreibt man `??` anstelle von `?`, so wird zusätzlich der Quellcode des jeweiligen Objekts angezeigt.

magic-Funktionen

In Ipython ist es möglich, einige zusätzliche Funktionen aufzurufen; diese nur in Ipython-Sitzungen definierten Funktionen werden als `magic`-Funktionen bezeichnet und mit `%` bzw. `%%` eingeleitet. Mittels `%lsmagic` werden beispielsweise alle derartigen Funktionen aufgelistet:

```
%lsmagic
```

```
# Ergebnis:

# Available line magics:
# %alias  %alias_magic  %autocall  %autoindent  %automagic  %bookmark  %cd
# %colors  %config  %cpaste  %debug  %dhist  %dirs  %doctest_mode  %ed  %edit
# %env  %gui  %hist  %history  %install_default_config  %install_ext
# %install_profiles  %killbgscripts  %load  %load_ext  %loadpy  %logoff
# %logon  %logstart  %logstate  %logstop  %lsmagic  %macro  %magic
# %matplotlib  %notebook  %page  %paste  %pastebin  %pdb  %pdef  %pdoc  %pfile
# %pinfo  %pinfo2  %popd  %pprint  %precision  %profile  %prun  %psearch
# %psource  %pushd  %pwd  %pycat  %pylab  %quickref  %recall  %rehashx
# %reload_ext  %rep  %rerun  %reset  %reset_selective  %run  %save  %sc
# %store  %sx  %system  %tb  %time  %timeit  %unalias  %unload_ext  %who
# %who_ls  %whos  %xdel  %xmode

# Available cell magics:
# %%!  %%HTML  %%SVG  %%bash  %%capture  %%debug  %%file  %%html  %%javascript
# %%latex  %%perl  %%prun  %%pypy  %%python  %%python3  %%ruby  %%script  %%sh
# %%svg  %%sx  %%system  %%time  %%timeit  %%writefile
```

Im folgenden werden einige der `magic`-Funktionen kurz vorgestellt.

Magic-Funktionen automatisch erkennen

Durch eine Eingabe von `%automagic` im Ipython-Interpreter werden im Verlauf der Sitzung die Namen der Magic-Funktionen in den globalen Namensraum aufgenommen. Im folgenden kann damit wahlweise `pwd` oder `%pwd`

eingegeben werden, um den Namen des aktuellen Arbeitsverzeichnisses anzuzeigen; das beziehungsweise die `%`-Zeichen können anschließend also weggelassen werden.

Wissenschaftliche Notation von Zahlen

Mittels der `%precicion`-Anweisung kann eingestellt werden, in welchem Format Zahlenergebnisse im Ipython-Interpreter ausgegeben werden sollen. Persönlich empfinde ich folgende Einstellung als angenehm:

```
%precision %.4g
```

Mit dieser Einstellung erhält man Ergebnisse mit wissenschaftlicher „e-Notation“ dargestellt, wobei maximal drei Nachkommastellen ausgegeben werden; beispielsweise liefert auf diese Weise die Eingabe `5/100000000000000000` das Ergebnis `5e-20`.

Zeilen und Zellen

Ipython kennt – ebenso wie der Standard-Python-Interpreter – zwei Arten von Anweisungen: Zum einen „simple“ einzeilige Anweisungen, zum anderen „zusammengesetzte“ Anweisungen, die aus mehreren Zeilen bestehen. In Ipython wird eine solche Block-Anweisung, die stets mit einer leeren Zeile endet, auch als „Zelle“ bezeichnet.

Die `line magic`-Funktionen beziehen sich auf eine einzelne, einzeilige Anweisung; den `cell magic`-Funktionen werden hingegen der jeweiligen Anweisung weitere Zeilen angefügt. Beispielsweise kann mittels `%% writefile dateiname` der unmittelbar folgenden Text (ohne Anführungszeichen!) in eine Datei geschrieben werden, bis die Eingabe durch ein zweimaliges Drücken von `Enter` beendet wird.

```
%%writefile test.txt
Hallo
Welt!
```

```
#Ergebnis: Writing tmp.txt
```

Mittels `%%writefile -a` wird der folgende Text an die angegebene Datei angehängt; eine Eingabe von leeren Zeilen oder von formatiertem Text ist so allerdings nicht möglich, die unmittelbar folgende Texteingabe wird „as it is“ geschrieben.

Code via Copy-und-Paste einfügen

Versucht man einen ganzen Code-Block per Copy-und-Paste einzufügen, so kann es zu einer Fehlermeldung kommen, wenn der Block leere Zeilen enthält: Ipython sieht an dieser Stelle die Eingabe der „Zelle“ als beendet an und beginnt die nächste (die dann meist eine falsche Einrückung aufweist).

Um dieses Problem zu umgehen, kann man die Magic-Funktion `%cpaste` aufrufen: Anschließend wird der gesamte (beispielsweise mittels `Paste`) eingefügte Text als eine einzige Eingabe-Zelle interpretiert – bis `Ctrl d` gedrückt wird, oder eine Textzeile eingegeben wird, die lediglich die Zeichenkette `--` enthält.

Auf diese Weise kann man beispielsweise Vim mit dem Plugin Vicle als Editor verwenden und von dort aus Code an einen Ipython-Shell-Interpreter senden.

Python-Skripte aufrufen

Python-Skripte lassen sich folgendermaßen vom Ipython-Interpreter aus aufrufen:

```
%run path/script.py [arguments]
```

Befindet man sich bereits im Pfad der Programmdatei oder wechselt mittels `os.chdir(path)` dorthin, so kann die Pfadangabe im obigen Aufruf weggelassen werden.

Der obige Aufruf entspricht dem üblichen Aufruf von `python3 path/script.py` in einer Shell. Benötigt das Skript gegebenenfalls weitere Argumente, so können diese im Anschluss an die Pfadangabe des Skripts angegeben werden. Ist das aufgerufene Skript fehlerhaft und/oder benötigt es zu lange zur Ausführung, so kann es mit `Ctrl c` unterbrochen werden (KeyboardInterrupt).

Ein Vorteil der `%run`-Anweisung liegt darin, dass alle im aufgerufenen Skript definierten Variablen und Funktionen importiert und anschließend in der interaktiven Sitzung genutzt werden können (als wären sie direkt eingeben worden). Ein weiterer Vorteil liegt darin, dass beim Aufruf von `run` zusätzliche Optionen angegeben werden können:

- Mit `%run -t` („timer“) wird die Laufzeit des Python-Skript in Kurzform dargestellt.

  Der Timer listet auf, wie viel Zeit beim Ausführen des Skripts für System-Aufrufe, wie viel auf benutzerspezifische Rechenschritte und wie viel Gesamt benötigt wurde.

- Mit `%run -t` („profiler“) wird die Laufzeit der einzelnen im Python-Skript aufgerufenen Anweisungen detailliert dargestellt.

  Der Profiler listet dabei auf, wie häufig eine Funktion aufgerufen wurde und wie viel Zeit dabei je Ausführung beziehungsweise insgesamt benötigt wurde.

- Mit `%run -d` („debugger“) wird das Programm im Python-Debugger `pdb` gestartet.

  Der Debugger durchläuft das Programm Anweisung für Anweisung und hält dabei an vorgegebenen Breakpoints oder bei nicht abgefangenen Exceptions; man kann sich dann beispielsweise die Werte von Variablen anzeigen lassen, die für den auftretenden Fehler verantwortlich sein können.

Debugging

Anstelle ein Python-Skript mittels `%run -d script.py` von Anfang an im Debugger zu starten, kann man in Ipython mittels `%debug` einen allgemeinen Debug-Modus aktivieren. In diesem Fall wird der Debugger automatisch gestartet, wenn eine nicht abgefangene Exception auftritt.

Interaktion mit der Shell

Im Ipython-Interpreter lassen sich Shell-Anweisungen ausführen, indem man diesen ein `!` voranstellt; beispielsweise listet `!ls` den Inhalt des aktuellen Verzeichnisses auf. Gibt man `files = !ls` ein, so wird die Ausgabe der Shell-Anweisung `ls` als Liste in der Python-Variablen `files` gespeichert.

Umgekehrt kann man den Inhalt von Python-Variablen an die Shell-Anweisung übergeben, indem man der Variablen ein `$`-Zeichen voranstellt. Man könnte also `!echo "$files` eingeben, um die in der Variablen `files` gespeicherten Inhalte mittels echo auszugeben.

Konfigurationen

Eigene Konfigurationen lassen sich in Ipython mittels so genannter „Profile“ festlegen. Auf diese Weise kann beispielsweise festgelegt werden, welche Module beim Start von Ipython automatisch geladen oder welche Variablen standardmäßig definiert werden sollen; die Ipython-Profile ermöglichen darüber hinaus weitere Möglichkeiten, das Aussehen und Verhalten des Interpreters zu steuern.

Ein neues Profil kann folgendermaßen erstellt werden:

```
# Profil "default" erstellen:
ipython3 profile create

# Profil "profilname" erstellen:
ipython3 profile create profilname
```

Hierdurch wird das Profil mit dem angegebenen Namen im Verzeichnis `~/.ipython` neu angelegt; lässt man den Profilnamen weg, so wird das Profil automatisch `default` genannt. Bei künftigen Sitzungen wird, sofern vorhanden, das Profil `default` automatisch geladen, außer man wählt startet Ipython explizit mit dem angegebenen Profilnamen:

```
# Ipython mit "default"-Profil starten:
ipython3

# Ipython mit "profilname"-Profil starten:
ipython3 --profile=profilname
```

Durch das Erstellen eines Profils wird im Verzeichnis `~/.ipython/profile_default` (oder einem entsprechenden Profilnamen) automatisch ein Satz Konfigurationsdateien erstellt. Die wichtigsten Konfigurationsdateien sind:

- `ipython_qtconsole_config.py`: Diese Datei wird aufgerufen, wenn Ipython mittels `ipython3 qtconsole`, also mit graphischer QT-Console gestartet wird.
- `ipython_notebook_config.py`: Diese Datei wird aufgerufen, wenn Ipython mittels `ipython3 notebook`, also als Webanwendung gestartet wird.
- `ipython_config.py`: Diese Datei wird *immer* aufgerufen, wenn Ipython mit dem angegebenen Profil gestartet wird.

Alle Konfigurationen enthalten sämtliche Einstellungsoptionen mitsamt der zugehörigen Beschreibungen in Kommentarform; um eine Einstellung vorzunehmen, muss also nur das Kommentarzeichen am Anfange der jeweiligen Zeile entfernt und der Konfig-Variable der gewünschte Wert zugewiesen werden.

Abgesehen von vielen zusätzlichen Kommentaren kann eine Konfigurationsdatei somit beispielsweise folgendermaßen aussehen:

```
# sample ipython_config.py
# Configuration file for ipython.

c = get_config()

# lines of code to run at IPython startup.
c.InteractiveShellApp.exec_lines = [
    'import math as m',
    'import numpy as np',
    'import matplotlib as mpl',
    'import matplotlib.pyplot as plt',
    'import sympy as sy',
    'import pandas as pd',
    'pi = m.pi',
    'sq = m.sqrt',
    'sin = m.sin',
    'cos = m.cos',
    'tan = m.tan',
    'atan = m.atan',
    'rad = m.radians',
    'deg = m.degrees',
    'g = 9.81',
    '%precision %.4g',
    ]

# Autoindent IPython code entered interactively.
c.InteractiveShell.autoindent = True

# Enable magic commands to be called without the leading %.
c.TerminalInteractiveShell.automagic = True

c.TerminalInteractiveShell.history_length = 10000

# Set the color scheme (NoColor, Linux, or LightBG).
c.TerminalInteractiveShell.colors = 'Linux'
c.TerminalInteractiveShell.color_info = True
```

Die einzelnen Optionen können bei Bedarf auch innerhalb einer laufenden Sitzung geändert werden; hierzu muss man lediglich eine Anweisung der Form `%config InteractiveShell.autoindent = True` eingeben.

Weitere Infos zu Ipython gibt es in der offiziellen Dokumentation (en.).

`matplotlib` – ein Plotter für Diagramme

Die Matplotlib ist eine umfangreichste Bibliothek, mit deren Hilfe verschiedene Diagrammtypen wie Linien-, Stab- oder Kuchendiagramme, Histogramme, Boxplots, Kontourdiagramme, aber auch dreidimensionale Diagramme und Funktionenplots auf einfache Weise erstellt werden können.

Die Matplotlib ist nicht im Python3-Standard enthalten und muss daher nachinstalliert werden :

```
sudo aptitude install python3-matplotlib python3-tk

# oder easy_install3 matplotlib nach Installation von python3-setuptools
```

Das zweite Paket ist notwendig, um geplottete Diagramme in einer graphischen Bedienoberfläche anzeigen zu können.

Liniendiagramme mit `plot()` erstellen

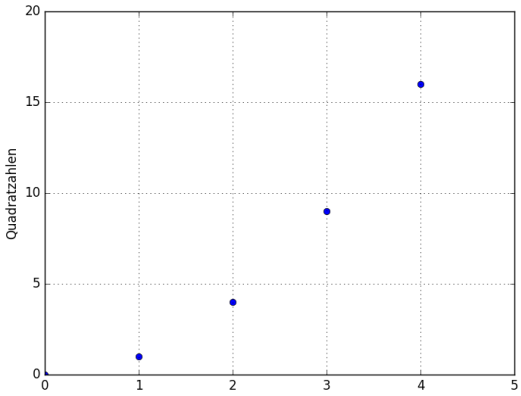
Die Matplotlib kann mittels `import matplotlib` eingebunden werden; da der Name sehr lang ist, empfiehlt sich beispielsweise folgende Abkürzung:

```
import matplotlib as mlp
import matplotlib.pyplot as plt
```

Das erste Paket ist nützlich, um Einstellungen für die Matplotlib vorzunehmen; das zweite beinhaltet unter anderem die wichtige `plot()`-Funktion, die zum Erstellen von 2D-Graphiken genutzt werden kann:

```
# Label für die y-Achse vergeben:
plt.ylabel('Quadratzahlen')

# Einen x-y-Plot erstellen:
plt.plot([1,2,3,4], [1,4,9,16], 'bo')

# Achsen-Bereiche manuell festlegen
# Syntax: plt.axis([xmin, xmax, ymin, ymax])
plt.axis([0, 5, 0, 20])

# Ein gepunktetes Diagramm-Gitter einblenden:
plt.grid(True)

# Diagramm anzeigen:
plt.show()
```

Durch einen Aufruf von `plt.show()` wird das zuvor definierte Diagramm in einem graphischen Ausgabefenster angezeigt:



Schließt man das Ausgabefenster wieder (beispielsweise durch Drücken von `Ctrl w`), so kann man mit der Ipython-Sitzung fortfahren.

Optionen für Diagramm-Linien

Wie man im obigen Beispiel sieht, gibt es zahlreiche Möglichkeiten, das Aussehen des Diagramms zu beeinflussen:

- Mit `plt.axis( [xmin,xmax,ymin,ymax] )` kann der Wertebereich der Achsen manuell festgelegt werden.
- Mit `plt.grid(True)` wird ein zur Achsen-Skalierung passendes Gitter als Diagramm-Hintergrund eingezeichnet.
- Mit `plt.xscale('log')` wird die x-Achse logarithmisch skaliert.
  Mit `plt.yscale('log')` wird entsprechend die y-Achse logarithmisch skaliert.
- Mit `plt.xlabel('Text')` kann die x-Achse des Diagramms beschriftet werden.
  Mit `plt.ylabel('Text')` wird entsprechend die y-Achse beschriftet.
- Mit `plt.title('Text')` kann eine Überschrift über das Diagramm drucken.

Beim Setzen von derartigen Optionen für ein Diagramm muss prinzipiell nicht auf die Reihenfolge geachtet werden; es kann allerdings eine existierende Optionen durch eine später eingegebene Option überschrieben (oder korrigiert) werden.

Die Funktion `plt.plot()` bietet bereits selbst einige Einstellungsmöglichkeiten:

- Wird `plt.plot()` mit nur einer Zahlen-Liste als Argument aufgerufen, so werden diese automatisch durchnummeriert (beginnend mit `0`); für die x-Achse wird dann diese Nummerierung als Wertebereich verwendet.
- Wird `plt.plot()` mit Zahlen-Listen als Argument aufgerufen, so wird die erste Liste als Wertebereich der x-Achse und die zweite Liste als Wertebereich der y-Achse angesehen.
- Zusätzlich zu der oder den Zahlen-Listen kann der Funktion `plt.plot()` als letztes Argument eine Zeichenkette übergeben werden, welche die Farbe und Form der Diagramm-Linie festlegt:

– *Zu Beginn* dieser Zeichenkette kann die *Farbe* der Diagrammlinie festgelegt werden:

Symbol	Farbe
`b`	`blue`
`c`	`cyan`
`g`	`green`
`m`	`magenta`
`r`	`red`
`y`	`yellow`
`k`	`black`
`w`	`white`

– *Am Ende* dieser Zeichenkette kann die *Form* der Diagrammlinie festgelegt werden:

Symbol	Form
`-`	Durchgezogene Linie
`--`	Gestrichelte Linie
`-.`	Abwechselnd gestrichelte und gepunktete Linie
`:`	Gepunktete Linie
`o`	Einzelne Punkte, Darstellung als farbige Kreise
`s`	Einzelne Punkte, Darstellung als farbige Rechtecke
`D`	Einzelne Punkte, Darstellung als Diamant-Form
`^`	Einzelne Punkte, Darstellung als farbige Dreiecke
`x`	Einzelne Punkte, Darstellung als farbige `x`-Zeichen
`*`	Einzelne Punkte, Darstellung als farbige `*`-Zeichen
`+`	Einzelne Punkte, Darstellung als farbige `+`-Zeichen

Eine vollständige Liste möglicher Marker findet sich hier. Es ist auch möglich, beispielsweise mittels `'b'` nur die Linienfarbe auf `blue` oder mittels `'--'` nur die Linienform als gestrichelte Linie festzulegen.

- Als Alternative zu der zuletzt genannten Festlegung von Farbe und Form einer Diagrammlinie kann für das obige Beispiel auch folgende explizite Syntax gewählt werden:

```
plt.plot([1,2,3,4], [1,4,9,16], color='blue', linestyle='', marker='o')
```

Diese Syntax ist zwar mit mehr Schreibarbeit verbunden, ermöglicht es allerdings, beispielsweise bei einem Linien-Plot mit einem weiteren Attribut `linewith=2.0` die Linienstärke auf den doppelten Wert zu setzen.

Diagramme mit mehreren Linien

Möchte man mehrere Linien in einem einzelnen Diagramm darstellen, so muss man lediglich die Funktion `plt.plot()` mehrfach mit den jeweiligen Wertelisten aufrufen – alle

Linien werden dadurch in das selbe Diagramm geplottet. Erst durch eine Eingabe von `plt.show()` wird das fertige Diagramm auf dem Bildschirm ausgegeben.

Beispiel:

```
# Wertebereich für x-Achse festlegen:
x = [0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0]

x2 = [num**2 for num in x]
x3 = [num**3 for num in x]

# Einzelne Diagramm-Linien plotten:
plt.plot(x, x, 'r--')
plt.plot(x, x2,'bs')
plt.plot(x, x3,'g^')

# Diagramm-Gitter einblenden:
plt.grid(True)

# Diagramm ausgeben:
plt.show()
```

Ergebnis:

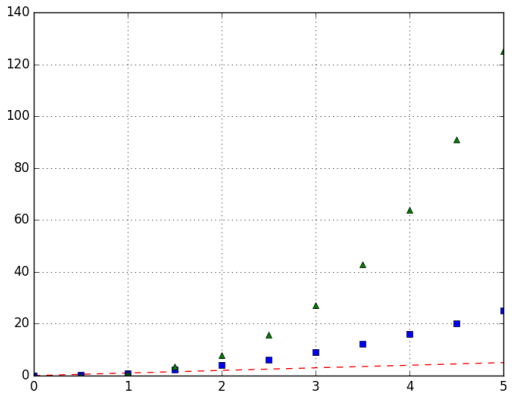


Diagramme speichern

Um ein Diagramm als Graphik-Datei abzuspeichern, kann man einfach im Ausgabe-Fenster auf das „Save Figure“-Icon klicken und im erscheinenden Fenster einen Dateinamen eingeben und den gewünschten Zielpfad wählen:

Eine andere Möglichkeit besteht darin, das Diagramm **vor** einem Aufruf von `plt.show()` mittels folgender Anweisung zu speichern:

```
# Speichern als PNG-Datei:
plt.savefig('/pfad/dateiname.png')

# Speichern als SVG-Datei:
plt.savefig('/pfad/dateiname.png', format='svg')
```

Ist der Aufruf von `plt.show()` beendet, so verfällt nämlich das bisherige Diagramm mitsamt allen dafür getroffenen Festlegungen. Dies hat einerseits den Vorteil, dass man unmittelbar mit der Eingabe des nächsten Diagramms beginnen kann, ohne die Einstellungen erst zurücksetzen zu müssen. Andererseits ist ein Ändern des bisherigen Diagramms nur möglich, indem man die bisherigen Eingaben über die History-Funktion des Interpreters (↑-Taste) zurückholt und als Vorlage für ein neues Diagramm nimmt.

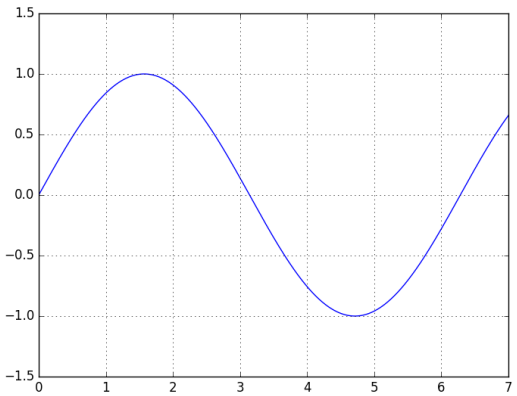
Zahlenbereiche als Werte-Listen

Anstelle einer Zahlenliste kann der Funktion `plt.plot()` auch eine beziehungsweise zwei Zahlenbereiche übergeben werden, die beispielsweise mittels *np.arange()* oder `np.linspace()` aus dem *numpy*-Modul generiert wurden. Dies hat den Vorteil, dass man mit einer sehr guten Rechen-Performance die Wertebereiche in sehr kleine Schritte unterteilen kann und die erstellten Linien somit nahezu „glatt“ erscheinen:

Beispiel:

```
import numpy as np

# Werte-Listen für Sinus-Funktion generieren:
x = np.arange(0, 10, 0.01)  # Start, Stop, Step
y = np.sin(x)

# Sinus-Kurve plotten:
plt.plot(x, y)
plt.axis( [0, 7, -1.5, 1.5] )
plt.grid(True)
plt.show()
```

Ergebnis:



Anpassung von Matplotlib-Diagrammen

Matplotlib-Diagramme haben allgemein folgenden Aufbau:

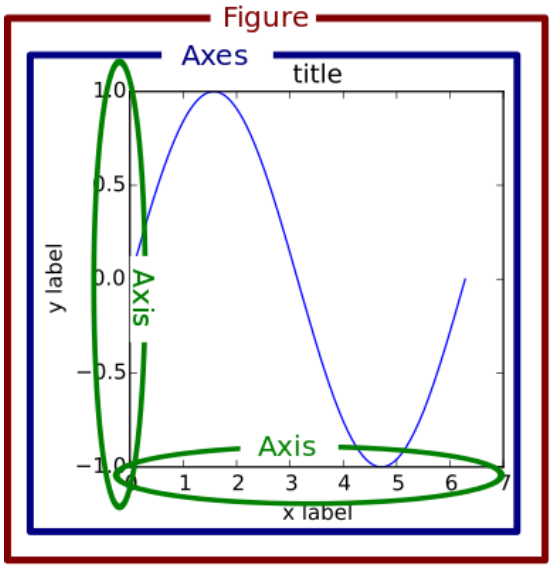


Abb. 1: Allgemeiner Aufbau eines Matplotlib-Diagramms (Quelle: Matplotlib-Dokumentation

Die Basis eines jeden Diagramms ist also ein `Figure`-Objekt (mit möglichem Titel), das eigentliche Diagramm wird durch den Wertebereich der Achsen (`axes`) festgelegt. Insbesondere muss zwischen `axes` und `axis` unterschieden werden: Die letztere Bezeichnung bezieht sich nur auf entweder die x- oder die y-Achse.

Anhand des Sinus-Plots aus dem letzten Abschnitt soll im folgenden Beispiel gezeigt werden, wie man mittels der obigen Objekte das Aussehen eines Matplotlib-Diagramms anpassen beziehungsweise verbessern kann.[1]

```
import numpy as np
import matplotlib.pyplot as plt

# Werte-Listen für eine Sinus- und Cosinus-Funktion erstellen:
x = np.linspace(-np.pi, np.pi, 500, endpoint=True)
cos_x = np.cos(x)
sin_x = np.sin(x)

# Diagramm-Linien plotten:
plt.plot(x, cos_x)
plt.plot(x, sin_x)

# Diagramm anzeigen:
plt.show()
```

Mit der obigen Plot-Anweisung erhält man – ohne weitere Einstellungen – folgendes Diagramm:

[1] Die Quelle zu diesem Tutorial (englischsprachig, ebenfalls unter einer Creative-Commons-License) stammt von Nicolas P. Rougier und ist Teil seines Matplotlib-Tutorials.

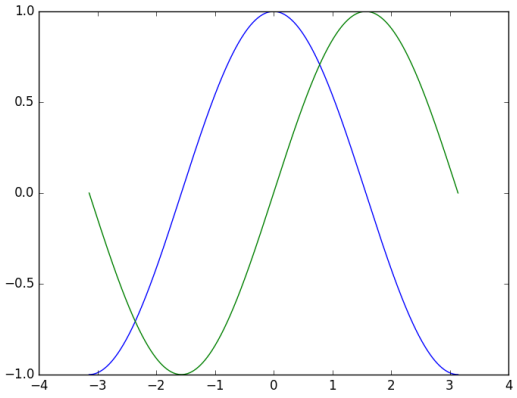


Abb. 2: Sinus- und Cosinus-Plot mit Basis-Einstellungen

Auf diese Weise kann man sich mit sehr wenig Aufwand ein Bild von einer mathematischen Funktion verschaffen. Dass nur eine minimale Code-Menge nötig ist, liegt auch daran, dass in den Matplotlib-Funktionen für viele Einstellungen Standard-Werte vorgegeben sind. Würde man alle diese Werte explizit angeben, so würde das obige Code-Beispiel folgendermaßen aussehen:

```
# Module importieren
import numpy as np
import matplotlib.pyplot as plt

# Werte-Listen für eine Sinus- und Cosinus-Funktion erstellen:
x = np.linspace(-np.pi, np.pi, 500, endpoint=True)
cos_x = np.cos(x)
sin_x = np.sin(x)

# Eine neues Matplot-Figure-Objekt mit 8x6 Zoll und
# einer Auflösung von 100 dpi erstellen:
plt.figure(figsize=(8, 6), dpi=80)

# In diese Abbildung ein 1x1 großes Diagramm-Gitter erstellen;
# Als aktuelles Diagramm wird das erste dieses Gitters ausgewählt:
plt.subplot(111)

# Cosinus-Funktion mit blauer Farbe, durchgehender Linie und 1 Pixel
# Linienbreite plotten:
plt.plot(x, cos_x, color="blue", linewidth=1.0, linestyle="-")

# Sinus-Funktion mit grüner Farbe, durchgehender Linie und 1 Pixel
# Linienbreite plotten:
plt.plot(x, sin_x, color="green", linewidth=1.0, linestyle="-")

# Grenzen für die x-Achse festlegen:
plt.xlim(-4.0, 4.0)

```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
29 # Grenzen für die y-Achse festlegen:
30 plt.ylim(-1.0, 1.0)
31
32 # "Ticks" (Bezugspunkte) für x-Achse festlegen:
33 plt.xticks(np.linspace(-4, 4, 9,endpoint=True))
34
35 # "Ticks" (Bezugspunkte) für y-Achse festlegen:
36 plt.yticks(np.linspace(-1, 1, 5,endpoint=True))
37
38 # Diagramm anzeigen:
39 plt.show()
```

Die Ausgabe dieses Codes ist mit dem obigen Diagramm absolut identisch. Man kann dieses „ausführlichere“ Code-Beispiel allerdings sehr gut als Ausgangsbasis für verschiedene Anpassungen verwenden.

Größe, Farben und Grenzen anpassen

Das ursprüngliche Diagramm erscheint bei der Original-Größe (8×6 Zoll) in vertikaler Richtung stark gestreckt; im Fall der Sinus- und Cosinus-Funktion, deren Wertebereich nur zwischen -1 und $+1$ liegt, ist wohl ein eher breiteres Diagramm besser geeignet. Hierzu kann man im obigen Code-Beispiel die Zeile 12 durch folgende Zeile ersetzen:

```
# Größe des Plots anpassen:
plt.figure(figsize=(10,6), dpi=80)
```

Die Linien werden im ursprünglichen Diagramm zudem mit einer Linienbreite von nur 1 px gedruckt. Würde man dieses Diagramm beispielsweise mit einem Schwarz-Weiß-Drucker drucken (und die gedruckte Seite womöglich anschließend noch kopieren), so wären die Linien nicht mehr gut zu erkennen – zudem hätten die Linien ähnliche Grau-Werte. Um dies zu verbessern, kann man im obigen Code-Beispiel die Zeilen 20 und 24 durch folgende Zeilen ersetzen:

```
# Farbe und Dicke der Diagrammlinien anpassen:
plt.plot(x, cos_x, color='blue', linewidth=2.5, linestyle='-')
plt.plot(x, sin_x, color='red',  linewidth=2.5, linestyle='-')
```

Gibt man zudem keinen Wertebereich für die x- und y-Achse an, verwendet die Matplotlib einfach die Minima und Maxima der darzustellenden Werte als Diagramm-Grenzen (zum betraglich nächst größeren Integer-Wert aufgerundet); die Diagramm-Linie stößt somit an den Diagramm-Grenzen an. Möchte man hier einen „weicheren“ Übergang, also nicht anstoßende Diagrammlinien, so kann ein geringfügig größerer Wertebereich für die Diagramm-Grenzen gewählt werden. Hierzu kann man im obigen Code-Beispiel die Zeilen 27 und 30 durch folgende Zeilen ersetzen:

```
# Wertebereiche der Achsen anpassen:
plt.xlim(x.min()*1.1, x.max()*1.1)
plt.ylim(cos_x.min()*1.1, cos_x.max()*1.1)
```

Mit diesen Anpassungen erhält man bei einem Aufruf von `plt.show()` nun folgendes Diagramm:

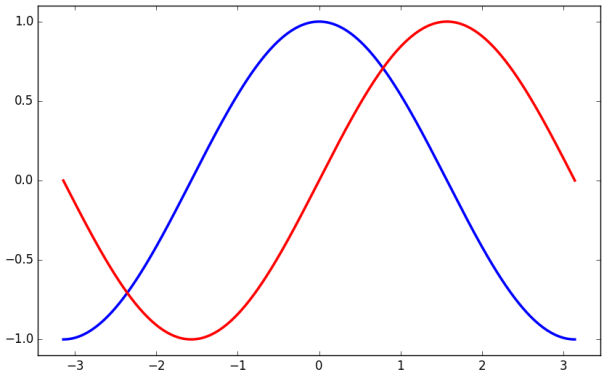


Abb. 3: Sinus- und Cosinus-Plot mit anderen Farb- und Größenanpassung.

„Ticks“ (Bezugspunkte) für die Achsen anpassen:

Zur Darstellung der trigonometrischen Funktionen ist die gewöhnliche Skalierung der x-Achse nicht ideal: Man kann damit beispielsweise nur näherungsweise ablesen, an welchen Stellen die Funktionen Nullstellen oder Maxima haben. Praktischer wäre es, die Skalierung der x-Achse anhand der Kreiszahl π festzulegen. Hierzu kann man im obigen Code-Beispiel die Zeile `33` durch folgende Zeile ersetzen:

```
# Auf der x-Achse fünf Bezugspunkte (als Vielfache von pi) festlegen:
plt.xticks( [-np.pi, -np.pi/2, 0, np.pi/2, np.pi] )
```

Plottet man das so abgeänderte Diagramm, so bekommt man auf der x-Achse die numerischen Werte angezeigt, also beispielsweise `3.142` anstelle von π. Glücklicherweise kann man beim Aufruf von `plt.xticks()` nicht nur die Position der Ticks, sondern durch Angabe einer zweiten Liste auch ihre Beschriftung festlegen, und hierfür optional auch LaTeX-Formelzeichen nutzen:

```
# Auf der x-Achse fünf Bezugspunkte (als Vielfache von pi) festlegen
# und mittels LaTeX-Symbolen beschriften:
plt.xticks( [-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
            [ r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$']
    )

# Auch Ticks für die y-Achse anpassen:
plt.yticks( [-1.0, -0.5, 0, 0.5, 1],
            [ r'$-1$', r'$-1/2$', r'', r'$+1/2$', r'$+1$']
    )
```

Das `r` vor den einzelnen Zeichenketten bewirkt, dass diese als `raw` angesehen werden, also Sonderzeichen wie `$` nicht mit einem zusätzlichen \-Zeichen versehen werden müssen.

Im obigen Beispiel wurde auch die Beschriftung der y-Achse angepasst, damit die Schriftart identisch ist. Als Ergebnis erhält man bei einem Aufruf von `plt.show()` damit folgendes Diagramm:

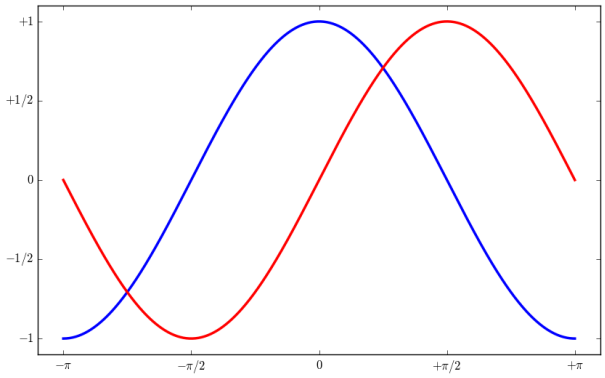


Abb. 4: Sinus- und Cosinus-Plot mit Anpassung der Achsenbeschriftung.

Diagramm-Achsen verschieben:

Das Achsen-Objekt `axes` eines Diagramms hat eine Eigenschaft, die mit `spines` bezeichnet wird; darin wird festgelegt, an welcher Stelle die Achsen dargestellt werden sollen. Standardmäßig wird in der Matplotlib die y-Achse am linken Rand des Diagramms gedruckt. Dies hat den Vorteil, dass man auch den rechten Rand für das zusätzliche Plotten einer zweiten Kurve als Werte-Achse nutzen kann, sofern sich die Wertebereiche beider Linien stark voneinander unterscheiden.

Soll allerdings, beispielsweise für Kurvendiskussionen oder geometrische Aufgaben üblich, ein Koordinatensystem mit vier Quadranten gezeichnet werden, so ist bisweilen ein Diagramm mit einem in der Mitte liegenden Nullpunkt und zwei durch diesen verlaufenden x- beziehungsweise y-Achsen besser geeignet. Dazu müssen die Achsen in die Mitte „verschoben“ werden.. Man kann dies erreichen, indem man zwei der vier `spines` (links, rechts, oben, unten) entfernt und die anderen beiden in die Mitte verschiebt. Hierzu kann man im obigen Code-Beispiel folgende Zeilen vor der Anweisung `plt.show()` einfügen:

```
# Das Achsen-Objekt des Diagramms in einer Variablen ablegen:
ax = plt.gca()

# Die obere und rechte Achse unsichtbar machen:
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

# Die linke Diagrammachse auf den Bezugspunkt '0' der x-Achse legen:
ax.spines['left'].set_position(('data',0))
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
# Die untere Diagrammachse auf den Bezugspunkt '0' der y-Achse legen:
ax.spines['bottom'].set_position(('data',0))

# Ausrichtung der Achsen-Beschriftung festlegen:
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
```

Als Ergebnis erhält man damit bei einem Aufruf von `plt.show()` folgendes Diagramm:

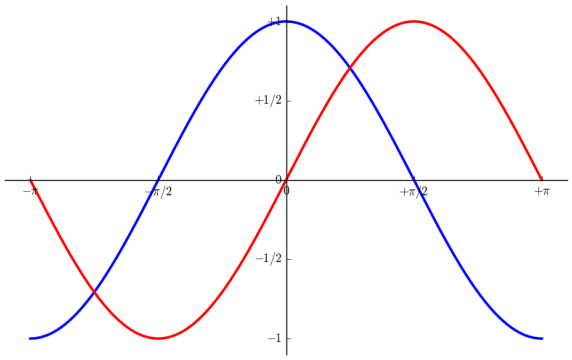


Abb. 5: Sinus- und Cosinus-Plot mit anderer Positionierung der Achsen.

Diese Darstellungsform ist zwar elegant, doch werden in diesem Fall die Beschriftungen der x- und y-Achse teilweise durch die Funktionsgraphen verdeckt. Als Workaround kann man einerseits Schriftgröße der Achsenbeschriftung ändern, und andererseits diese durch eine halb-transparente Umrandung hervorheben. Hierzu kann man folgenden Code vor der Anweisung `plt.show()` einfügen:

```
# Achse-Beschriftungen durch weiß-transparenten Hintergrund hervorheben:
for label in ax.get_xticklabels() + ax.get_yticklabels():
    label.set_fontsize(16)
    label.set_bbox(dict(facecolor='white', edgecolor='None', alpha=0.65 ))
```

Als Ergebnis erhält man damit durch einen Aufruf von `plt.show()`:

Titel, Legende und Text hinzufügen

Normalerweise werden Diagramme bei der Textsatzung mit einer Bildunterschrift („Caption“) eingebunden; meist wird dabei auch eine Abbildungs-Nummer mit eingefügt, so dass das Diagramm aus dem Text heraus eindeutig referenziert werden kann. Erstellt man ein Diagramm hingegen für eine Pinnwand oder eine Overhead-Folie / Beamer-Präsentation, so ist ein groß gedruckter Titel über dem Diagramm bisweilen nützlich. Ein solcher kann folgendermaßen zum Diagramm hinzugefügt werden:

```
# Titel hinzufügen:
plt.title('Sinus und Cosinus', fontsize=20, color='gray')
```

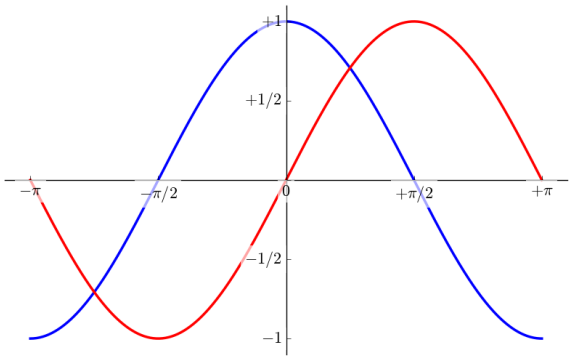


Abb. 6: Sinus- und Cosinus-Plot mit anderer Positionierung der Achsen und hervorgehobener Achsen-Bsechriftung.

Auch hier kann bei Bedarf wieder LaTeX-Code im Titeltext verwendet werden.

Als weitere Verbesserung ist es sehr nützlich, wenn in einem Diagramm mit mehreren Linien als „Legende“ angezeigt wird, welche Bedeutung die einzelnen Linien haben. Hierzu kann man bei den einzelnen `plt.plot()`-Anweisungen zusätzlich einen `label`-Parameter angeben und anschließend die Legende mittels `plt.legend()` anzeigen:

```
# Plots mit einem Label versehen:
plt.plot(x, cos_x, color="blue", linewidth=2.5, linestyle="-", label=r'$\cos(x)$
↪')
plt.plot(x, sin_x, color="red",  linewidth=2.5, linestyle="-", label=r'$\sin(x)$
↪')

# Legende einblenden:
plt.legend(loc='upper left', frameon=True)
```

Als zusätzliche Beschriftung können noch weitere Text-Elemente in das Diagramm aufgenommen werden. Besondere Stellen lassen sich zudem mit Pfeilen oder Hilfslinien hervorheben.

```
# Hervorzuhebende Stelle festlegen:
pos = 2*np.pi/3

# Vertikale gestrichelte Linie an der Stelle 'pos' einzeichnen
# (von der x-Achse bis zum Graph der cos-Funktion):
plt.plot([pos,pos], [0,np.cos(pos)], color ='blue', linewidth=1.5, linestyle="--
↪")

# Punkt(e) auf der cos-Linie mit Marker versehen:
# (Die x- und y-Werte müssen -- wie bei plot() -- als Liste angegeben werden)
# (Mit s=50 wird die Größe ('size') auf 50 Pixel festgelegt)
plt.scatter([pos], [np.cos(pos)], s=50, marker='o', color ='blue')

# Eigenen Text einfügen:
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
plt.annotate(r'$\sin(\frac{2\pi}{3}) = \frac{\sqrt{3}}{2}$',
             xy=(pos, np.sin(pos)), xycoords='data',
             xytext=(+10, +30), textcoords='offset points', fontsize=16,
             arrowprops=dict(arrowstyle="->", connectionstyle="arc3, rad=.2"))

# Vertikale gestrichelte Linie an der Stelle 'pos' einzeichnen
# (von der x-Achse bis zum Graph der sin-Funktion):
plt.plot([pos,pos], [0,np.sin(pos)], color ='red', linewidth=1.5, linestyle="--
↪")

# Punkt(e) auf der sin-Linie mit Marker versehen:
plt.scatter([pos], [np.sin(pos)], s=50, marker='o', color ='red')

# Eigenen Text einfügen:
plt.annotate(r'$\cos(\frac{2\pi}{3}) = -\frac{1}{2}$',
             xy=(pos, np.cos(pos)), xycoords='data',
             xytext=(-90, -50), textcoords='offset points', fontsize=16,
             arrowprops=dict(arrowstyle="->", connectionstyle="arc3, rad=.2"))
```

Damit erhält man schließlich das folgende Diagramm:

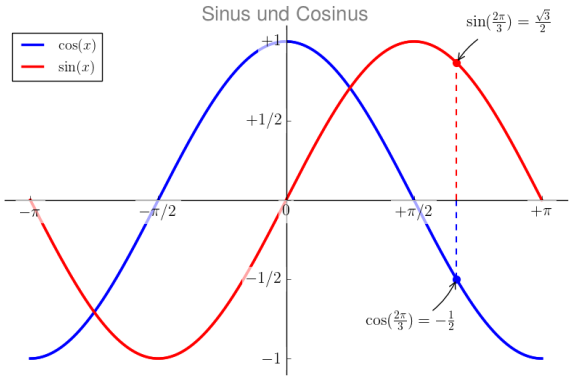


Abb. 7: Sinus- und Cosinus-Plot mit zusätzlichen Beschriftungen.

... to be continued ...

Links

- Matplotlib-Gallery mit Code-Beispielen (en.)

numpy – eine Bibliothek für numerische Berechnungen

Numpy ist eine Python-Bibliothek mit Datentypen und Funktionen, die für numerische Berechnungen optimiert sind. Meist lassen sich solche Aufgaben zwar auch mit den norma-

len Python-Funktionen berechnen, gerade bei großen Zahlenmengen ist Numpy allerdings wesentlich schneller.

Numpy ist nicht im Python3-Standard enthalten und muss daher separat installiert werden :

```
sudo aptitude install python3-numpy

# oder easy_install3 numpy nach Installation von python3-setuptools
```

Der zentrale Objekttyp in Numpy ist das `ndarray` (meist kurz „Array“ genannt), das viele Gemeinsamkeiten mit der normalen *Liste* aufweist. Ein wesentlicher Unterschied besteht allerdings darin, dass alle im Array gespeicherten Elemente den gleichen Objekttyp haben müssen. Die Größe von Numpy-Arrays kann zudem nicht verändert werden, und es sind keine leeren Objekte erlaubt. Durch derartige Eigenschaften können Numpy-Arrays vom Python-Interpreter schneller durchlaufen werden. Darüber hinaus stellt Numpy etliche grundlegende Funktionen bereit, um mit den Inhalten solcher Arrays zu arbeiten und/oder Änderungen an solchen Arrays vorzunehmen.

Numpy-Arrays erstellen

Ein neues Numpy-Array kann folgendermaßen aus einer normalen Liste, deren Elemente alle den gleichen Datentyp haben müssen, erzeugt werden:

```
import numpy as np

nums = [1,2,3,4,5]

# Eindimensionales Array erstellen:
a = np.array(nums)

a
# Ergebnis: array([1, 2, 3, 4, 5])
```

Die beim Funktionsaufruf von `array()` übergebene Liste kann auch aus mehreren Teillisten bestehen, um beispielsweise zeilenweise die Werte einer Matrix als Numpy-Array zu speichern:

```
# Zweidimensionale Matrix erstellen
m1 = np.array([ [1,2,3], [4,5,6], [7,8,9] ])

m1
# Ergebnis:
# array([[1, 2, 3],
#        [4, 5, 6],
#        [7, 8, 9]])
```

Durch ein zweites Argument kann beim Aufruf der `array()`-Funktion der Datentyp der Elemente explizit festgelegt werden. Beispielsweise könnten im obigen Beispiel durch eine

zusätzliche Angabe von `dtype=float` die in der Liste enthaltenen Integer-Werte automatisch in Gleitkomma-Zahlen umgewandelt werden.

Da auch Matrizen voller Nullen oder Einsen häufig vorkommen, können diese mittels der dafür vorgesehenen Funktionen `zeros()` bzw. `ones()` erzeugt werden. Dabei wird als erstes Argument ein Tupel als Argument angegeben, welches die Anzahl an Zeilen und Spalten der Matrix festlegt, sowie als zweites Argument wiederum optional der Datentyp der einzelnen Elemente:

```
# 2x3-Matrix aus Nullen erstellen:

# Zweidimensionale Matrix erstellen
m2 = np.zeros( (2,3), int)

m2
# Ergebnis:
# array([[0, 0, 0],
#    [0, 0, 0]])
```

Eindimensionale Arrays mittels `arange()` und `linspace()`

Mittels der Funktion `arange()` kann ein (eindimensionales) Numpy-Array auf Grundlage eines Zahlenbereichs erstellt werden:

```
# Numpy-Array aus Zahlenbereich mit angegebener Schrittweite erstellen:
# Syntax: np.arange(start, stop, step)

r = np.arange(0, 10, 0.1)

r
# Ergebnis:
# array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ,
#         1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9,  2. ,  2.1,
#         2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9,  3. ,  3.1,  3.2,
#         3.3,  3.4,  3.5,  3.6,  3.7,  3.8,  3.9,  4. ,  4.1,  4.2,  4.3,
#         4.4,  4.5,  4.6,  4.7,  4.8,  4.9,  5. ,  5.1,  5.2,  5.3,  5.4,
#         5.5,  5.6,  5.7,  5.8,  5.9,  6. ,  6.1,  6.2,  6.3,  6.4,  6.5,
#         6.6,  6.7,  6.8,  6.9,  7. ,  7.1,  7.2,  7.3,  7.4,  7.5,  7.6,
#         7.7,  7.8,  7.9,  8. ,  8.1,  8.2,  8.3,  8.4,  8.5,  8.6,  8.7,
#         8.8,  8.9,  9. ,  9.1,  9.2,  9.3,  9.4,  9.5,  9.6,  9.7,  9.8,
#         9.9])
```

Die Funktion `arange()` verhält sich also genauso wie die Funktion *range()*, liefert allerdings ein Numpy-Array mit den entsprechenden Werten als Ergebnis zurück.[1]

[1] Auch bei der `arange()`-Funktion ist die untere Grenze im Zahlenbereich enthalten, die obere jedoch nicht.

Das optionale dritte Argument gibt, ebenso wie bei *range()*, die Schrittweite zwischen den beiden Zahlengrenzen an. Ist der Zahlenwert der unteren Bereichsgrenze größer als derjenige der oberen Bereichsgrenze, so muss ein negativer Wert als Schrittweite angegeben werden, andererseits bleibt das resultierende Array leer.

Eine zweite, sehr ähnliche Möglichkeit zur Erstellung eines Numpy-Arrays bietet die Funktion `linspace()`: Bei dieser wird allerdings die Anzahl der Schritte zwischen dem Start- und dem Endwert angegeben; die Schrittweite wird dann automatisch berechnet.

```
# Numpy-Array aus Zahlenbereich mit angegebener Listen-Länge erstellen:
# Syntax: np.arange(start, stop, num)

l = np.linspace(0, 10, 100, endpoint=True)

l
# Ergebnis:
# array([  0.        ,   0.1010101 ,   0.2020202 ,   0.3030303 ,
#        0.4040404 ,   0.50505051,   0.60606061,   0.70707071,
#        0.80808081,   0.90909091,   1.01010101,   1.11111111,
#        1.21212121,   1.31313131,   1.41414141,   1.51515152,
#        1.61616162,   1.71717172,   1.81818182,   1.91919192,
#        2.02020202,   2.12121212,   2.22222222,   2.32323232,
#        2.42424242,   2.52525253,   2.62626263,   2.72727273,
#        2.82828283,   2.92929293,   3.03030303,   3.13131313,
#        3.23232323,   3.33333333,   3.43434343,   3.53535354,
#        3.63636364,   3.73737374,   3.83838384,   3.93939394,
#        4.04040404,   4.14141414,   4.24242424,   4.34343434,
#        4.44444444,   4.54545455,   4.64646465,   4.74747475,
#        4.84848485,   4.94949495,   5.05050505,   5.15151515,
#        5.25252525,   5.35353535,   5.45454545,   5.55555556,
#        5.65656566,   5.75757576,   5.85858586,   5.95959596,
#        6.06060606,   6.16161616,   6.26262626,   6.36363636,
#        6.46464646,   6.56565657,   6.66666667,   6.76767677,
#        6.86868687,   6.96969697,   7.07070707,   7.17171717,
#        7.27272727,   7.37373737,   7.47474747,   7.57575758,
#        7.67676768,   7.77777778,   7.87878788,   7.97979798,
#        8.08080808,   8.18181818,   8.28282828,   8.38383838,
#        8.48484848,   8.58585859,   8.68686869,   8.78787879,
#        8.88888889,   8.98989899,   9.09090909,   9.19191919,
#        9.29292929,   9.39393939,   9.49494949,   9.5959596 ,
#        9.6969697 ,   9.7979798 ,   9.8989899 ,  10.        ])
```

Setzt man im obigen Beispiel `endpoint=False`, so ist das mit `linspace()` erzeugte Array `l` mit dem Array `r` aus dem vorherigen Beispiel identisch.

Inhalte von Numpy-Arrays abrufen und verändern

Entspricht ein Numpy-Array einem eindimensionalen Vektor, so kann auf die einzelnen Elemente in gleicher Weise wie bei einer Liste zugegriffen werden:

```
nums = [1,2,3,4,5]

a = np.array(nums)
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
a[3]
# Ergebnis: 4

a[-1]
# Ergebnis: 5
```

Als positive Indizes sind Werte zwischen `i >= 0` und `i < len(array)` möglich; sie liefern jeweils den Wert des `i+1`-ten Listenelements als Ergebnis zurück. Für negative Indizes sind Werte ab `i <= -1` möglich; sie liefern jeweils den Wert des `i`-ten Listenelements – vom Ende der Liste her gerechnet – als Ergebnis zurück. Die Indizierung kann ebenso genutzt werden, um den Inhalt des Arrays an einer bestimmten Stelle zu verändern:

```
a[-1] = 10

a
# Ergebnis: array([1, 2, 3, 4, 10])
```

Um auf Zahlenbereiche innerhalb eines Numpy-Arrays zuzugreifen, können wiederum – wie bei der Indizierung von *Listen und Tupeln* – so genannte *Slicings* genutzt werden. Dabei wird innerhalb des Indexoperators `[]` der auszuwählende Bereich mittels der Syntax `start:stop` festgelegt, wobei für `start` und `stop` die Index-Werte der Bereichsgrenzen eingesetzt werden:

```
r = np.arange(10)

# Intervall selektieren:

r[3:8]
# Ergebnis: array([3, 4, 5, 6, 7])

# Jedes zweite Element im angegebenen Intervall auswählen:

r[3:8:2]
# Ergebnis: array([3, 5, 7])
```

Wie üblich wird bei Slicings die untere Grenze ins Intervall mit eingeschlossen, die obere nicht. Mit der Syntax `start:stop:step` kann bei Slicings zudem festgelegt werden, dass innerhalb des ausgewählten Zahlenbereichs nur jede durch die Zahl `step` bezeichnete Zahl ausgewählt wird. Wird für `start` oder `step` kein Wert angegeben, so wird der ganze Bereich ausgewählt:

```
# Ab dem fünften Element (von hinten beginnend) jedes Element auswählen:

r[5::-1]
# Ergebnis: array([5, 4, 3, 2, 1, 0])
```

Slicings können bei Zuweisungen von neuen Werten auch auf der linken Seite des `=`-Zeichens stehen. Auf diese Weise kann bisweilen auf eine `for`-Schleife verzichtet und der Code somit lesbarer gemacht werden.

Um in mehrdimensionalen Numpy-Arrays Werte zu selektieren, wird folgende Syntax verwendet:

```
m = np.array([ [1,2,3], [4,5,6] ])

m
# Ergebnis:
# array([[1, 2, 3],
#    [4, 5, 6]])


# Element in der zweiten Zeile in der dritten Spalte auswählen:

m[1][2]
# Ergebnis: 6
```

Bei Numpy-Arrays können die „Verschachtelungstiefen“ wie bei Listen durch eine mehrfache Anwendung des Index-Operators `[]` aufgelöst werden; ebenso ist für das obige Beispiel allerdings auch die Syntax `m3[1,2]` erlaubt und auch üblich. Bei der Auswahl eines Elements aus einer Matrix können also innerhalb des Index-Operators die Zeile und Spalte durch ein Komma getrennt ausgewählt werden; Slicings sind hierbei ebenfalls möglich.

Funktionen für Numpy-Arrays

Viele Funktionen wie die Betragsfunktion `abs()`, die Wurzelfunktion `sqrt()` oder trigonometrische Funktionen wie `sin()`, `cos()` und `tan()`, die im `math`-Modul definiert sind, existieren in ähnlicher Weise auch im Numpy-Modul – mit dem Unterschied, dass sie auf Numpy-Arrays angewendet werden können. Dabei wird die jeweilige mathematische Funktion auf jedes einzelne Element des Arrays angewendet, und als Ergebnis ebenfalls ein Array mit den entsprechenden Funktionswerten zurück gegeben.[2]

Ebenso können die gewöhnlichen Operationen `+`, `-`, `*` und `/` angewendet werden, um beispielsweise zu allen Elemente eines Numpy-Arrays eine bestimmte Zahl zu addieren/subtrahieren oder um alle Elemente mit einer bestimmten Zahl zu multiplizieren. Die Numpy-Funktionen erzeugen dabei stets neue Numpy-Arrays, lassen die originalen Arrays also stets unverändert.

```
r = np.arange(10)

r
# Ergebnis: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8, 9])

r+1
# Ergebnis: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

(continues on next page)

[2] Die gleichnamigen Funktionen aus dem `math`-Modul können also auf einzelne Elemente eines Numpy-Arrays, nicht jedoch auf das ganze Array an sich angewendet werden. Letzteres könnte man zwar beispielsweies mittels einer `for`-Schleife erreichen, doch die Ausführung des Codes bei Verwendung der Numpy-Varianten ist erheblich schneller.

(Fortsetzung der vorherigen Seite)

```
r**2
# Ergebnis: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])

np.sqrt(r**4)
# Ergebnis: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])

np.sin(r)
# Ergebnis: array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.
↪7568025 ,
#  -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
```

Zusätzlich gibt es in Numpy Funktionen, die speziell für Zahlenreihen und Matrizen vorgesehen sind. Beispielsweise kann mit der Numpy-Funktionen `argmin()` und `argmax()` der Index des kleinsten und größten Elements in einem Array gefunden werden. Wendet man diese Funktionen auf ein Matrix-Array an, so erhält man diejenige Index-Nummer des kleinsten beziehungsweise größten Elements, die sich bei einem eindimensionalen Array mit den gleichen Werten ergeben würde. Ist man hingegen spalten- oder zeilenweise an den jeweiligen Minima beziehungsweise Maxima interessiert, so kann beim Aufruf dieser beiden Funktionen als zweites Argument `axis=0` für eine spaltenweise Auswertung oder `axis=1` für eine zeilenweie Auswertung angegeben werden:

```
a = np.array( [3,1,2,6,5,4] )
m = np.array([ [3,1,2], [6,5,4] ])

np.argmin(a)
# Ergebnis: 1

np.argmin(m)
# Ergebnis: 1

np.argmin(m, axis=0)
# Ergebnis: array([0, 0, 0])

np.argmin(m, axis=1)
# Ergebnis: array([1, 2])
```

Für Matrix-Arrays existieren zusätzlich die Numpy-Funktionen `dot()`, `inner()` und `outer()`, mit deren Hilfe Multiplikationen von Matrizen beziehungsweise Vektoren durchgeführt werden können.

... to be continued ...

Links

- Numpy-Tutorial (en.) von Nicolas P. Rougier
- 100 Numpy Exercises (en.)

pandas – eine Bibliothek für tabellarische Daten

Pandas ist eine Python-Bibliothek, die vorrangig zum Auswerten und Bearbeiten tabellarischer Daten gedacht ist. Dafür sind in Pandas drei Arten von Objekten definiert:

- Eine `Series` entspricht in vielerlei Hinsicht einer „eindimensionalen“ Liste, beispielsweise einer Zeitreihe, einer Liste, einem Dict, oder einem *Numpy* -Array.
- Ein `Dataframe` besteht aus einer „zweidimensionalen“ Tabelle. Die einzelnen Reihen beziehungsweise Spalten dieser Tabelle können wie `Series`-Objekte bearbeitet werden.
- Ein `Panel` besteht aus einer „dreidimensionalen“ Tabelle. Die einzelnen Ebenen dieser Tabelle bestehen wiederum aus `Dataframe`-Objekten.

In den folgenden Abschnitten sollen in Anlehnung an das berühmte 10 minutes to pandas-Tutorial die `Series`- und die `Dataframe`-Objekte als grundlegende und am häufigsten verwendeten Pandas-Objekte kurz vorgestellt werden.

Arbeiten mit `Series`-Objekten

Ein neues Series-Objekt kann mittels der gleichnamigen Funktion beispielsweise aus einer gewöhnlichen Liste generiert werden:

```
import pandas as pd

s = pd.Series( [5,10,15,20,25] )

s
# Ergebnis:
# 0     5
# 1    10
# 2    15
# 3    20
# 4    25
# dtype: int64
```

Das Series-Objekt erhält automatisch einen Index, so dass beispielsweise mittels `s[0]` auf das erste Element, mit `s[1]` auf das zweite Element, usw. zugegriffen werden kann. Neben diesen numerischen Indizes, die auch bei gewöhnlichen Listen verwendet werden, können explizit auch andere Indizes vergeben werden:

```
s.index =  ['a','b','c','d','e']

s
# Ergebnis:
# a     5
# b    10
# c    15
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
# d    20
# e    25
# dtype: int64
```

Nun können die einzelnen Elemente zwar immer noch mit `s[0]`, `s[1]`, usw., aber zusätzlich auch mittels `s['a']`, `s['b']` usw. ausgewählt werden.[1] Wird bei der Generierung eines Series-Objekts ein *Dict* angegeben, so werden automatisch die Schlüssel als Indizes und die Werte als eigentliche Listenelemente gespeichert.

Slicings

Sollen mehrere Elemente ausgewählt werden,so können die entsprechenden Indizes wahlweise als Liste oder als so genannter „Slice“ angegeben werden:

```
# Zweites und drittes Element auswählen:

s[ [1,2] ]
# Ergebnis:
# b    10
# c    15

# Identische Auswahl mittels Slicing:

s[ 1:3 ]
# Ergebnis:
# b    10
# c    15
```

Bei Slicings wird, ebenso wie bei *range()*-Angaben, die obere Grenze nicht in den Auswahlbereich mit eingeschlossen. Die Auswahl mittels Slicing hat bei Series-Objekten also die gleiche Syntax wie die *Auswahl von Listenobjekten*.

Zeitreihen

Zeitangaben in Series-Objekten können mittels der Pandas-Funktion `date_range()` generiert werden:

```
dates = pd.date_range('2000-01-01', '2000-01-07')

dates
# <class 'pandas.tseries.index.DatetimeIndex'>
# [2000-01-01, ..., 2000-01-07]
# Length: 7, Freq: D, Timezone: None
```

[1] Die Index-Liste kann auch bereits bei der Erzeugung eines neuen Series-Objekts mittels `Series(datenliste, index=indexliste)` angegeben werden.

Als Start- und Endpunkt werden allgemein Datumsangaben mit einer gleichen Syntax wie im `datetime`-Modul verwendet. Zusätzlich kann angegeben werden, in welchen Zeitschritten die Zeitreihe erstellt werden soll:

```
weekly = pd.date_range('2000-01-01', '2000-02-01', freq="W")

weekly
# Ergebnis:
# <class 'pandas.tseries.index.DatetimeIndex'>
# [2000-01-02, ..., 2000-01-30]
# Length: 5, Freq: W-SUN, Timezone: None


hourly = pd.date_range('2000-01-01 8:00', '2000-01-01 18:00', freq="H")

hourly
# Ergebnis:
# <class 'pandas.tseries.index.DatetimeIndex'>
# [2000-01-01 08:00:00, ..., 2000-01-01 18:00:00]
# Length: 11, Freq: H, Timezone: None
```

Die Elemente der Zeitreihe können explizit mittels `list(zeitreihe`, beispielsweise `list(dates)`, ausgegeben werden; in Series-Objekten werden Zeitreihen häufig als Index-Listen verwendet.

Arbeiten mit `Dataframe`-Objekten

Ein neues Dataframe-Objekt kann mittels der Funktion `DataFrame()` beispielsweise aus einer gewöhnlichen Liste generiert werden:

```
import pandas as pd

# 1D-Beispiel-Dataframe erstellen:
df = pd.DataFrame( [5,10,15,20,25] )

df
# Ergebnis:
#     0
# 0   5
# 1  10
# 2  15
# 3  20
# 4  25
#
# [5 rows x 1 columns]
```

Als Unterschied zu einem Series-Objekt werden bei einem Dataframe sowohl die Zeilen als auch die Spalten mit einem Index versehen.

Mehrspaltige Dataframes können auch über ein `dict`-Objekt definiert werden, wobei die Schlüsselwerte den Spaltennamen und die damit verbundenen Werte einzelnen Daten entsprechen, aus denen der Dataframe generiert werden soll:

```
# 2D-Beispiel-Dataframe erstellen:
df2 = pd.DataFrame({
    'A' : 1.,
    'B' : pd.date_range('2000-01-01', '2000-01-07'),
    'C' : pd.Series(range(7), dtype='float32'),
    'D' : np.random.randn(7),
    'E' : pd.Categorical(['on', 'off', 'on', 'off', 'on', 'off', 'on']),
    'F' : 'foo' })

df2
# Ergebnis:
#    A          B  C         D    E    F
# 0  1 2000-01-01  0 -2.611072   on  foo
# 1  1 2000-01-02  1  0.630309  off  foo
# 2  1 2000-01-03  2 -1.645430   on  foo
# 3  1 2000-01-04  3  1.056535  off  foo
# 4  1 2000-01-05  4  2.194970   on  foo
# 5  1 2000-01-06  5  0.537804  off  foo
# 6  1 2000-01-07  6  1.011678   on  foo
```

Wie man sieht, wird bei Angabe eines einzelnen Wertes für eine Spalte dieser als konstant für die ganze Spalte angenommen; listenartige Objekte hingegen müssen allesamt die gleiche Länge aufweisen.

Datentypen

Innerhalb einer Spalte eines Dataframe-Objekts müssen alle Werte den gleichen Datentyp aufweisen. Man kann sich die Datentypen der einzelnen Spalten folgendermaßen anzeigen lassen:

```
# Datentypen anzeigen:

df2.dtypes
# Ergebnis:
# A           float64
# B    datetime64[ns]
# C           float32
# D           float64
# E          category
# F            object
# dtype: object
```

Daten anzeigen und sortieren

Bei längeren Datensätzen kann es bereits hilfreich sein, nur einen kurzen Blick auf den Anfang oder das Ende der Tabelle werfen zu können. Bei Dataframe-Objekten ist dies mittels der Funktionen `head()` beziehungsweise `tail()` möglich:

```
# Die ersten fünf Zeilen des Dataframes anzeigen:

df2.head()
# Ergebnis:
#    A          B  C         D    E    F
# 0  1 2000-01-01  0 -2.611072   on  foo
# 1  1 2000-01-02  1  0.630309  off  foo
# 2  1 2000-01-03  2 -1.645430   on  foo
# 3  1 2000-01-04  3  1.056535  off  foo
# 4  1 2000-01-05  4  2.194970   on  foo

# Die letzten drei Zeilen des Dataframes anzeigen:

df2.tail(3)
# Ergebnis:
#   A          B  C         D    E    F
# 4 1 2000-01-05  4  2.194970   on  foo
# 5 1 2000-01-06  5  0.537804  off  foo
# 6 1 2000-01-07  6  1.011678   on  foo
```

Standardmäßig geben `head()` und `tail()` je fünf Zeilen aus; ist eine andere Anzahl gewünscht, so kann diese als Argument angegeben werden.

Spalten und Index-Werte

Die einzelnen Bestandteile eines Dataframes, d.h. die Spaltennamen, die Index-Werte sowie die eigentlichen Daten, können über die Attribute `columns`, `index` und `values` des Dataframes abgerufen werden:

```
# Spaltennamen, Index-Werte und Inhalt des Dataframes ausgeben:

df2.columns
# Ergebnis:
# Index(['A', 'B', 'C', 'D', 'E', 'F'], dtype='object')

df2.index
# Ergebnis:
# Int64Index([0, 1, 2, 3, 4, 5, 6], dtype='int64')

df2.values
# Ergebnis:
# array([[1.0, Timestamp('2000-01-01 00:00:00'), 0.0, -2.611072451193798, 'on',
↪'foo'],
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
#   [1.0, Timestamp('2000-01-02 00:00:00'), 1.0, 0.6303090119623712, 'off', 'foo
↪'],
#   [1.0, Timestamp('2000-01-03 00:00:00'), 2.0, -1.645429619256174, 'on', 'foo
↪'],
#   [1.0, Timestamp('2000-01-04 00:00:00'), 3.0, 1.056535156797566, 'off', 'foo
↪'],
#   [1.0, Timestamp('2000-01-05 00:00:00'), 4.0, 2.1949702833421596, 'on', 'foo
↪'],
#   [1.0, Timestamp('2000-01-06 00:00:00'), 5.0, 0.5378036597920774, 'off', 'foo
↪'],
#   [1.0, Timestamp('2000-01-07 00:00:00'), 6.0, 1.01167812002758, 'on', 'foo
↪']],
#   dtype=object)
```

Statistische Übersicht

Eine Kurz-Analyse der Daten ist über die Methode `describe()` des Dataframes möglich. Man erhält als Ergebnis eine Übersicht über die jeweiligen Mittelwerte sowie einige statistische Streuungsmaße (Standardabweichung, größter und kleinster Wert, Quartile). Da sich diese Größen nur für quantitative (genauer: invervall-skalierte) Merkmalswerte bestimmen lassen, werden die jeweiligen Werte auch nur für die in Frage kommenden Spalten angezeigt:

```
# Statistische Kurz-Info anzeigen:

df2.describe()
# Ergebnis:
#        A          C         D
# count  7  7.000000  7.000000
# mean   1  3.000000  0.167828
# std    0  2.160247  1.681872
# min    1  0.000000 -2.611072
# 25%    1  1.500000 -0.553813
# 50%    1  3.000000  0.630309
# 75%    1  4.500000  1.034107
# max    1  6.000000  2.194970
```

Sortiermethoden

Die Daten eines Dataframes können zudem wahlweise nach Zeilen oder Spalten oder auch anhand der jeweiligen Werte sortiert werden:

- Mit der Methode `sort_index()` können die Daten nach Zeilen (`axis=0`) oder Spalten (`axis=1`) sortiert werden; mittels `ascending=False` kann zudem die Reihenfolge der Sortierung umgekehrt werden.

```
df2.sort_index(axis=1, ascending=False)
# Ergebnis:
#      F    E         D  C          B  A
# 0  foo   on -2.611072  0 2000-01-01  1
# 1  foo  off  0.630309  1 2000-01-02  1
# 2  foo   on -1.645430  2 2000-01-03  1
# 3  foo  off  1.056535  3 2000-01-04  1
# 4  foo   on  2.194970  4 2000-01-05  1
# 5  foo  off  0.537804  5 2000-01-06  1
# 6  foo   on  1.011678  6 2000-01-07  1
```

Wird zusätzlich das optionale Argument `inline=True` gesetzt, so wird nicht ein verändertes Resultat angezeigt (das beispielsweise in einer neuen Variablen gespeichert werden könnte); vielmehr wird in diesem Fall die Änderung auch im ursprünlichen Dataframe-Objekt übernommen.

- Mit der Methode `sort_value()` können die Daten ihrer Größe nach sortiert werden. Standardmäßig werden die Daten dabei zeilenweise (`axis=0`) und in aufsteigender Reihenfolge (`ascending=True`) sortiert; bei Bedarf können diese Variablen angepasst werden.

```
df2.sort_values(by='D')
# Ergebnis:
#    A          B  C         D    E    F
# 0  1 2000-01-01  0 -2.611072   on  foo
# 2  1 2000-01-03  2 -1.645430   on  foo
# 5  1 2000-01-06  5  0.537804  off  foo
# 1  1 2000-01-02  1  0.630309  off  foo
# 6  1 2000-01-07  6  1.011678   on  foo
# 3  1 2000-01-04  3  1.056535  off  foo
# 4  1 2000-01-05  4  2.194970   on  foo
```

Auch bei dieser Sortiermethode können die Änderungen mittels `inline=True` nicht nur angezeigt, sondern direkt in den Original-Dataframe übernommen werden.

Daten auswählen

Dataframe-Objekte ähneln in gewisser Hinsicht `dict`-Objekten: Die einzelnen Spalten beziehungsweise Zeilen können mithilfe des Spalten- beziehungsweise Index-Namens ausgewählt werden.

Ein Zugriff auf einzelne Zeilen oder Spalten ist beispielsweise mit Hilfe des Index-Operators `[ ]` möglich. Gibt man hierbei einen Spaltennamen oder eine Liste mit Spaltennamen an, so werden die jeweiligen Spalten ausgewählt; gibt man hingegen eine Zeilennummer oder einen Zeilenbereich an, so erhält man die jeweilige(n) Zeile(n) als Ergebnis:

```
df2['B']
# Ergebnis:
# 0   2000-01-01
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
# 1   2000-01-02
# 2   2000-01-03
# 3   2000-01-04
# 4   2000-01-05
# 5   2000-01-06
# 6   2000-01-07
# Name: B, dtype: datetime64[ns]

df2[['B','D']]
# Ergebnis:
#            B          D
# 0 2000-01-01  -2.611072
# 1 2000-01-02   0.630309
# 2 2000-01-03  -1.645430
# 3 2000-01-04   1.056535
# 4 2000-01-05   2.194970
# 5 2000-01-06   0.537804
# 6 2000-01-07   1.011678

df2[1:3]
#    A          B  C         D    E    F
# 1  1 2000-01-02  1  0.630309  off  foo
# 2  1 2000-01-03  2 -1.645430   on  foo
```

Bei Bereichsangaben mittels Slicings ist wie gewöhnlich die untere Grenze im Bereich mit enthalten, die obere hingegen nicht.

Selektion mittels Labeln

Um auf einzelne Elemente eines Dataframes zugreifen zu können, muss sowohl eine Zeilen- wie auch eine Reihenauswahl möglich sein. Für Dataframes ist dafür unter anderem der `.loc[]`-Operator definiert, mit dem eine Zeilen- beziehungsweise Spaltenauswahl anhand der `index`- beziehungsweise `columns`-Bezeichnungen möglich ist. Die Syntax lautet hierbei `dataframe.loc[zeilenbereich,spaltenbereich]`, wobei für die Bereichsangaben sowohl einzelne Index-Werte, Werte-Listen oder auch Slicings erlaubt sind; eine Bereichs-Angabe von `:` bewirkt, dass der gesamte Zeilen- beziehungsweise Spaltenbereich ausgewählt werden soll.

Beispiel:

```
# df2.loc[1:3, ['B','D']]
#            B         D
# 1 2000-01-02  0.630309
# 2 2000-01-03 -1.645430
# 3 2000-01-04  1.056535
```

Anders als beim gewöhnlichen Auswahloperator werden bei Benutzung des `.loc[]`-Operators bei Slicings *beide* Grenzen zum Bereich dazugerechnet.

Möchte man nur einen *einzelnen* Wert auswählen, als Resultat also einen Skalar erhalten, so kann mit gleicher Syntax auch der `.at[]`-Operator verwendet werden, der für diese Aufgabe eine geringere Rechenzeit benötigt.

Selektion mittels Positionsangaben

Ein zweiter Auswahl-Operator für Dataframes ist der `.iloc[]`-Operator. Das „i“ steht dabei für „integer“ und soll darauf hinweisen, dass dieser Auswahl sowohl für die Angabe des Zeilen- wie auch des Spaltenbereichs eine numerische Positionsangabe erwartet. Wie bei einer Liste wird die erste Zeile beziehungsweise Spalte eines Dataframes intern mit `0`, die zweite mit `1`, usw. nummeriert, unabhängig von den `index`- beziehungsweise `columns`-Bezeichnungen. Die Syntax für den `.iloc`-Operator lautet also `dataframe.iloc[zeilenbereich,spaltenbereich]`, wobei wiederum einzelne Werte, Werte-Listen oder auch Slicings zur Angabe der Positionen erlaubt sind:

Beispiel:

```
# df2.iloc[1:3,[1,3]]
#            B         D
# 1 2000-01-02  0.630309
# 2 2000-01-03 -1.645430
# 3 2000-01-04  1.056535
```

Auch beim `.loc[]`-Operator werden bei Slicings *beide* Grenzen zum Bereich dazugerechnet.

Möchte man nur einen *einzelnen* Wert auswählen, als Resultat also einen Skalar erhalten, so kann mit gleicher Syntax auch der `.iat[]`-Operator verwendet werden, der für diese Aufgabe eine geringere Rechenzeit benötigt.

Eine Mischung zwischen dem `.loc[]` und dem `.iloc[]`-Operator stellt der `.ix[]`-Operator dar: Dieser versucht anhand der angegebenen Bereiche – ebenso wie der `.loc[]`-Operator – zunächst eine Auswahl anhand der `index`- beziehungsweise `columns`-Werte zu erreichen; ist dies allerdings nicht möglich, so versucht dieser Operator anschließend die fehlgeschlagene Bereichsauswahl wie der `.iloc[]`-Operator als Positionsangabe zu deuten.

Selektion mittels Bedingungen

Oftmals interessiert man sich nur für eine Teilmenge eines Dataframes, deren Daten bestimmte Bedingungen erfüllen; man weiß jedoch nicht unmittelbar, an welchen Stellen im Dataframe diese Daten abgelegt sind. Eine schnelle und elegante Methode für eine derartige Datenauswahl besteht darin, die obigen Auswahl-Operatoren mit der jeweiligen Bedingung anstelle einer Bereichsangabe zu verwenden.

Bei der Formulierung der Auswahl-Bedingungen kann genutzt werden, dass man bei der Anwendung von von Vergleichsoperatoren auf Dataframes boolesche Werte erhält:

Beispiel:

```
df2['D'] > 1
# 0    False
# 1    False
# 2    False
# 3     True
# 4     True
# 5    False
# 6     True
# Name: D, dtype: bool
```

Anstelle der obigen Syntax kann auch `df['D'].gt(0)` geschrieben werden, wobei `gt()` für „greater than“ steht. Diese und ähnliche Methoden gibt es sowohl für (mehrdimensionale) Dataframes als auch für (eindimensionale) Series-Objekte; ihr Vorteil besteht darin, dass sie sich verketten lassen. Beispielsweise liefert so `df2['D'].gt(0).lt(2)` den booleschen Wert `True` für alle Werte, die größer als `0` und kleiner als `2` sind.

Boolesche Methode	Bedeutung
`gt()`	Größer als
`lt()`	Kleiner als
`ge()`	Größer gleich
`le()`	Kleiner gleich
`eq()`	Gleich

Ein Series-Objekt mit booleschen Werten, wie man sie im obigen Beispiel erhalten hat, kann wiederum als Bereichsangabe für die oben genannten Auswahl-Operatoren genutzt werden:

```
df2[ df2['D'] > 1 ]
# Ergebnis:
#    A                    B  C        D    E    F
# 3  1  2000-01-04 00:00:00  3  1.05654  off  foo
# 4  1  2000-01-05 00:00:00  4  2.19497   on  foo
# 6  1  2000-01-07 00:00:00  6  1.01168   on  foo

df2.ix[ df2['D'] > 1, 'B' ]
# Ergebnis:
# 3   2000-01-04
# 4   2000-01-05
# 6   2000-01-07
# Name: B, dtype: datetime64[ns]
```

Durch die oben genannten Auswahl-Operatoren werden die ursprünglichen Dataframes nicht beeinflusst; man kann die Ergebnisse allerdings wiederum in extra Variablen ablegen und/oder erneut Auswahl-Operatoren auf die Resultate anwenden.

... to be continued ...

`sympy` – ein Computer-Algebra-System

Sympy ist ein Modul, das ein Computer-Algebra-System für Python bereitstellt. Es kann, wenn bei der *Python-Installation* das Paket `python3-setuptools` mit installiert wurde, in einer Shell folgendermaßen heruntergeladen installiert werden:

```
sudo easy_install3 sympy
```

Anschließend kann es beispielsweise mittels `import sympy` oder `import sympy as sy` importiert werden. Im folgenden wird von letzterer Variante ausgegangen, um Schreibarbeit zu sparen.

Im folgenden werden nur einige häufig vorkommende Funktionen von Sympy beschrieben. Eine vollständige Dokumentation findet man auf der Sympy-Projektseite.

Konstanten und mathematische Funktionen

Ähnlich wie im `math`-Modul sind auch in Sympy einige mathematische Konstanten definiert:

`sy.E`	Eulersche Zahl	$e = 2.71828\ldots$
`sy.pi`	Kreiszahl	$\pi = 3.14159\ldots$
`sy.GoldenRatio`	Goldener Schnitt	$\Phi = 1.61803\ldots$
`sy.oo`	Unendlich	∞

Ebenso sind in Sympy alle elementaren Funktionen wie `sin()`, `cos()`, `exp()` usw. definiert und können in gleicher Weise wie im `math`-Modul verwendet werden. Weitere hilfreiche Funktionen sind beispielsweise:

`sy.Abs(x)`	Betragsfunktion
`sy.binomial(n,k)`	Binomialkoeffizient
`sy.factorial(num)`	Fakultät
`sy.fibonacci(n)`	Fibonacci-Folge (n-tes Element)
`sy.log(x)`	Natürlicher Logarithmus (Basis e)
`sy.log(x, a)`	Logarithmus zur Basis a

Das besondere an den Sympy-Funktionen ist, dass diese nicht nur eine einzelne Zahl bzw. eine Variable als Argument akzeptieren, sondern auch auf so genannte „Symbole“ angewendet werden können. Mit diesem Datentyp werden in Sympy die in der Mathematik für Variablennamen genutzten Buchstaben dargestellt. Beispielsweise kann eine in der Mathematik typischerweise mit x bezeichnete Variable folgendermaßen als Sympy-Symbol definiert werden:

```
x = sy.S('x')
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
type(x)
# Ergebnis: sympy.core.symbol.Symbol
```

Möchte man mehrere „Symbole“ – also mehrere „mathematische“ Variablen – auf einmal definieren, so kann die `symbols()`-Funktion genutzt werden, um eine durch Leerzeichen getrennte Zeichenkette als Liste von Symbol-Bezeichnungen zu interpretieren:

```
x,y,z = sy.symbols('x y z')
```

Bei der Festlegung von Symbolen mittels `sy.S()` oder `sy.symbols()` kann auch als Option `positive=True` angegeben werden, um nicht-negative mathematische Variablen zu definieren.

Mit diesen Symbolen kann nun gerechnet werden, ohne ihnen einen expliziten Wert zuweisen zu müssen.

Ausmultiplizieren und Vereinfachen

Um mathematische Terme umzuformen oder zu vereinfachen, gibt es in Sympy unter anderem die Funktionen `expand()`, `factor()`, `ratsimp()` und `simplify()`.

Mit Hilfe der `expand()`-Funktion lassen sich beispielsweise binomische Formeln explizit berechnen:

```
x = sy.S('x')

sy.expand( (x + 2)**5 )
# Ergebnis: x**5 + 10*x**4 + 40*x**3 + 80*x**2 + 80*x + 32
```

Die `expand()`-Funktion kann mittels der Optionen `frac=True`, `log=True` oder `trig=True` auch zum Erweitern von Bruchtermen, Logarithmen oder trigonometrischen Ausdrücken verwendet werden:

```
x = sy.S('x')
x1, x2 = sy.symbols('x1 x2')

sy.expand( ((x+3)/x) / (x+1) , frac=True)
# Ergebnis: (x + 3)/(x**2 + x)

sy.expand( sy.log(x**5) , log=True, force=True)
# Ergebnis: 5*log(x)

sy.expand( sy.sin(x1+x2) , trig=True)
# Ergebnis: sin(x1)*cos(x2) + sin(x2)*cos(x1)
```

Im letzten Beispiel wurde die Erweiterung durch die Option `force=True` erzwungen, da Sympy in diesem Fall die angegebene Umformung des Terms als ungünstig einstuft.

Umgekehrt können beispielsweise Polynome mittels der Funktion `factor()` in einzelne Faktoren oder Binome zerlegt werden:

```
x = sy.S('x')

sy.factor( 3*x**5 + 7*x**2 )
# Ergebnis: x**2*(3*x**3 + 7)

sy.factor( x**2 + 2*x + 1 )
# Ergebnis: (x + 1)**2
```

Bruchterme lassen sich mittels der Funktion `ratsimp()` vereinfachen:

```
x = sy.S('x')
x1, x2 = sy.symbols('x1 x2')

sy.ratsimp( (x**2 - 9) / (x-3) )
# Ergebnis: x + 3

sy.ratsimp( 1/x1 + 1/x2 )
# Ergebnis: (x1 + x2) / (x1 * x2)
```

Weitere Vereinfachungen von Termen sind mit der Funktion `simplify()` möglich:

```
x = sy.S('x')

sy.simplify( sy.sin(x)**2 + sy.cos(x)**2 )
# Ergebnis: 1

sy.simplify( 3*sy.log(x) + 2 * sy.log(5*x) )
# Ergebnis: 5*log(x) + log(25)
```

Die Funktion `simpify()` kann auch genutzt werden, um die Äquivalenz zweier Terme T_1 und T_2 zu überprüfen. Dies ist nicht zuletzt deshalb von Bedeutung, da die mathematische Äquivalenz in Sympy nicht mit dem Vergleichsoperator als `T1 == T2` geprüft werden kann. Stattdessen kann aber geprüft werden, ob `simplify(T1 - T2)` den Wert Null ergibt:

```
x1, x2 = sy.symbols('x1 x2')

sy.sin(x1 + x2) == sy.sin(x1) * sy.cos(x2) + sy.cos(x1) * sy.sin(x2)
# Ergebnis: False

sy.simplify(
    sy.sin(x1 + x2) - ( sy.sin(x1) * sy.cos(x2) + sy.cos(x1) * sy.sin(x2) )
    )
# Ergebnis: 0
```

Für trigonometrische Vereinfachungen kann zudem die Funktion `trigsimp()` genutzt werden.

Gleichungen und Ungleichungen

Sympy kann insbesondere zum Lösen von Gleichungen, Gleichungssystemen und Ungleichungen genutzt werden. Eine Gleichung kann in Sympy folgendermaßen mittels der Funktion `Equation()` beziehungsweise der Kurzform `Eq()` definiert werden:

```
x = sy.S('x')

sy.Eq(x**2 +1, 3*x -1)
# Ergebnis: x**2 + 1 == 3*x - 1
```

Das Ergebnis von `Eq()` ist ein Gleichungs-Objekt. Dieses kann wahlweise in eine Variable gespeichert oder an die Funktion `solve()` übergeben werden, um die Lösung(en) der Gleichung zu bestimmen:

```
sy.solve( sy.Eq(x**2 +1, 3*x -1) )
# Ergebnis: [1, 2]
```

Gleichungen lassen sich auch mit mehreren Parametern a_i formulieren, die bei Bedarf mittels der Funktion `subs()` durch konkrete Werte ersetzt werden können:

```
x = sy.S('x')
a, b, c = sy.symbols("a b c")

eq = sy.Eq( a*x**2 + b*x + c, 0)

# Gleichung allgemein mit x als Variable lösen:

sy.solve( eq, x )
# Ergebnis: (-b + sqrt(-4*a*c + b**2))/(2*a), -(b + sqrt(-4*a*c + b**2))/(2*a)]

# Gleichung mit Parametern a=1, b=3, c=2 lösen:

sy.solve( eq.subs( {a:1, b:-3, c:2} ) )
# Ergebnis: [1, 2]
```

Die Funktion `solve()` kann auch verwendet werden, um Gleichungssysteme zu lösen. Hierzu empfiehlt es sich, die einzelnen Gleichungen zunächst zu einer Liste zusammenzufassen:

```
x1, x2, x3 = sy.symbols("x1 x2 x3")

equations = [
    sy.Eq( 8*x1 + 2*x2 + 3*x3 ,  15 ),
    sy.Eq( 6*x1 - 1*x2 + 7*x3 , -13 ),
    sy.Eq(-4*x1 + 5*x2 - 3*x3 ,  21 ),
]

sy.solve(equations)
# Ergebnis: {x2: 4, x1: 2, x3: -3}
```

Zum Formulieren von Ungleichungen mit einer einzelnen Variablen zu formulieren, können die folgenden Funktionen in gleicher Weise wie die Funktion `Eq()` genutzt werden:

`Ne()`	Ungleich	(„not equal“)
`Lt()`	Kleiner als	(„less than“)
`Le()`	Kleiner gleich	(„less or equal“)
`Gt()`	Größer als	(„greater than“)
`Ge()`	Größer gleich	(„greater or equal“)

Gegeben sei beispielsweise folgende Ungleichung:

$$x^2 - 8 \cdot x + 15 \leq 2$$

In Sympy lautet die Ungleichung etwa so:

```
sy.Le(x**2 - 8*x + 15, 2)
```

Um die Ungleichung zu lösen, wird der obige Ausdruck wiederum an die Funktion `solve()` übergeben:

```
sy.solve( sy.Le(x**2 - 8*x + 15, 2) )
# Ergebnis: And(-sqrt(3) + 4 <= re(x), im(x) == 0, re(x) <= sqrt(3) + 4)
```

Man erhält also die Schnittmenge („And“) von $[-\sqrt{3}+4\,;\,+\infty[$ und $]-\infty\,;\,\sqrt{3}+4]$ als Ergebnis, also das Intervall $[-\sqrt{3}+4\,;\,+\sqrt{3}+4]$. Die zusätzliche Angabe von `im(x) == 0` bedeutet lediglich, dass es sich bei der Lösung um eine reellwertige Lösung handelt.[1]

... to be continued ...

Links

- Sympy Projektseite

Beispielaufgaben für Scipy, Sympy und Pandas

Diese Sammlung an Übungsaufgaben und Lösungen (mit Quellcode) stellt eine in Python 3 geschriebene Version einer Vielzahl an ursprünglich für Maxima entwickelten Beispielaufgaben dar (siehe Original-Quelle).

Ziffernsumme

Bei dieser Aufgabe handelt es sich um eine Knobelaufgabe bzw. um eine einfache Übungsaufgabe für lineare Gleichungssysteme.

[1] Eine komplexe Zahl z, deren Imaginärteil $\mathrm{Im}(z)$ gleich Null ist, hat nur einen Realteil $\mathrm{Re}(z)$. Sie ist damit mit einer reellen Zahl x identisch, für die $x = \mathrm{Re}(z)$ gilt.

Aufgabe:

Die Aufgabe besteht darin, eine dreistellige Zahl zu finden, die folgende Bedingungen erfüllt:

1. Die Ziffernsumme einer dreistelligen Zahl ist gleich 18.
2. Die Hunderterstelle ist um 6 größer als das 2-fache der Zehnerstelle.
3. Die Einerstelle ist um 6 größer als das 3-fache der Zehnerstelle.

Lösung:

Definiert man die Variablen h als die Hunderterziffer, z als die Zehnerziffer und e als die Einerziffer, so ist die Ziffernsumme gleich $z+h+e$. Aus der Aufgabenstellung lassen sich damit folgende drei Gleichungen aufstellen:

1. Die Ziffernsumme einer dreistelligen Zahl ist 18:

$$z + h + e = 18$$

2. Die Hunderterstelle ist um 6 größer als das 2-fache der Zehnerstelle.

$$h - 6 = 2 \cdot z$$

3. Die Einerstelle ist um 6 größer als das 3-fache der Zehnerstelle.

$$e - 6 = 3 \cdot z$$

Dieses Gleichungsystem kann mittels *Sympy* gelöst werden. Der Code dazu lautet beispielsweise:

```
import sympy as sy

# Sympy-Variablen initiieren:
h, z, e = sy.S( 'h z e'.split() )

# Gleichungssystem formulieren:
equations = [
    sy.Eq( z + h + e , 18  ),
    sy.Eq( h - 6     , 2*z ),
    sy.Eq( e - 6     , 3*z ),
]

# Gleichungssystem lösen:
sy.solve(equations)

# Ergebnis: {h: 8, z: 1, e: 9}
```

Die Hunderterziffer ist gleich 8, die Zehnerziffer gleich 1 und die Einerziffer gleich 9. Die gesuchte Zahl lautet somit 819.

Zahlenrätsel (Division)

Bei dieser Aufgabe handelt es sich um eine Knobelaufgabe bzw. um eine einfache Übungsaufgabe für lineare Gleichungssysteme.

Aufgabe:

Die Problemstellung lautet eine Zahl x zu finden, die, wenn sie durch $(x-a)$ geteilt wird, eine gegebene Zahl b ergibt (a sei ebenfalls ein bekannter, konstanter Wert). Die Aufgabe soll für $a = 10$ und $b = 1\frac{5}{21}$ gelöst werden.

Lösung:

Es muss prinzipiell folgende Gleichung gelöst werden:

$$\frac{x}{x-a} = b$$

Für $a = 10$ und $b = 1\frac{5}{21}$ lautet die Gleichung konkret:

$$\frac{x}{x-10} = 1\frac{5}{21}$$

Diese Gleichung kann bereits ohne weitere Vereinfachungen mittels *Sympy* gelöst werden. Der Code dazu lautet folgendermaßen:

```
import sympy as sy

# Sympy-Variablen initiieren:
x    = sy.S( 'x' )
a, b = sy.S( [10, 1+5/21] )

# Gleichung formulieren:
equation = sy.Eq( x/(x-a) , b )

# Gleichung lösen:
sy.solve(equation)

# Ergebnis: [52.0000000000000]
```

Das Ergebnis der Aufgabe lautet somit $x = 52$.

Volumen bei Rotation um die x-Achse

Bei dieser Aufgabe geht es um eine Volumensberechnung beziehungsweise um die Berechnung eines bestimmten Integrals

Aufgabe:

Das Volumen eines Rotationskörpers soll berechnet werden. Gegeben sind die erzeugende Funktion, die Untergrenze und die Obergrenze des Intervalls. Die Rotation soll um die x-Achse erfolgen. Die Aufgabe soll für die Funktion $f(x) = x^2 + x + \frac{1}{x}$ gelöst werden, wobei die Untergrenze bei $a = 1$ und die Obergrenze bei $b = 5$ liegen soll.

Lösung:

Der Graph der Funktion $f(x) = x^2 + x + \frac{1}{x}$ kann mit Hilfe von numpy und der matplotlib geplottet werden:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

# Wertereihen erzeugen:
x = np.arange(start=1,stop=5, step=0.01)
y = x**2 + x + 1/x

# Funktion plotten:
plt.plot(x,y)

# Layout anpassen:
plt.axis([0,6,0,35])
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.grid(True)

plt.text(3.5, 30, "$f(x)=x^2 + x + 1/x$")

plt.show()
```

Das Ergebnis sieht folgendermaßen aus:

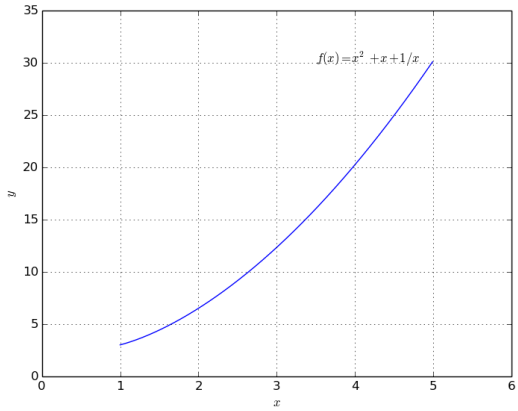


Abb. 8: Graph der Funktion $f(x) = x^2 + x + \frac{1}{x}$.

Um das Volumen des aus $f(x)$ bei Rotation um die x-Achse entstehenden Rotionskörpers zu berechnen, kann man sich diesen aus lauter winzig schmalen (Zylinder-)Scheiben zusammengesetzt denken. Jede dieser Scheiben hat als Radius den Wert $f(x)$, als Höhe den Wert $\mathrm{d}x$ und als Volumen somit $\pi \cdot r^2 \cdot h = \pi \cdot f(x)^2 \cdot \mathrm{d}x$. Alle diese infinitesimalen Volumina müssen aufsummiert werden, was folgendem Integral entspricht:

$$V = \int_a^b \left(\pi \cdot f^2(x)\right) \cdot \mathrm{d}x = \int_1^5 \pi \cdot \left(x^2 + x + \frac{1}{x}\right)^2 \cdot \mathrm{d}x$$

Dieses Integral kann mittels *Sympy* berechnet werden. Der Code dazu lautet folgendermaßen:

```
import sympy as sy

# Sympy-Variablen initiieren:
x    = sy.S( 'x' )
a, b = sy.S( [1, 5] )

# Bestimmtes Integral berechnen:
sy.integrate( sy.pi * (x**2 + x + 1/x) , (x,a,b) )

# Ergebnis: 15164*pi/15

# Alternativ: Ergebnis als Fließkommazahl ausgeben:
sy.integrate( sy.pi * (x**2 + x + 1/x) , (x,a,b) ).evalf()

# Ergebnis: 3175.94073326904
```

Das Volumen des Rotationskörpers beträgt somit rund $3174,94$ Volumeneinheiten.

Geradliniger Flug

Bei dieser Aufgabe geht es darum, eine physikalische Bewegungsgleichung zu lösen.

Aufgabe:

Zwei Flugzeuge verlassen einen Flughafen zur selben Zeit in entgegengesetzte Richtungen mit den Geschwindigkeiten $v_1 = 490\,\mathrm{km/h}$ beziehungsweise $v_2 = 368\,\mathrm{km/h}$. Nach welcher Zeit t haben sie einen Abstand von $s = 3805\,\mathrm{km}$ erreicht?

Lösung:

Die beiden Flugzeuge bewegen sich mit der Summe ihrer Geschwindigkeiten, also mit $v = v_1 + v_2 = 858\,\mathrm{km/h}$ auseinander. Aus der Weg-Zeig-Formel $s = v \cdot t$ für Bewegungen mit konstanter Geschwindigkeit lässt sich die gesuchte Größe t berechnen. Der *Sympy*-Code dazu lautet:

```
import sympy as sy

# Sympy-Variablen initiieren:
s = sy.S( 3805 )
v = sy.S(  858 )
t = sy.S(  't' )

# Gleichung formulieren:
equation = sy.Eq( s , v * t )

# Gleichung lösen:
result = sy.solve(equation)
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
# Ergebnis: [3805/858]

# Ergebnis als Fließkommazahl ausgeben:
float(result[0])

# Ergebnis: 4.434731934731935
```

Die gesuchte Zeit beträgt somit rund $4,435$ Stunden. Etwas eleganter ist allerdings die Angabe in Stunden und Minuten. Sie kann aus dem obigen Ergebnis folgendermaßen berechnet werden:

```
import math

result_f = float(result[0])

hours = math.floor(result_f)
# Ergebnis: 4.0

minutes = (result_f - math.floor(result_f)) * 60
# Ergebnis: 26.083916083916083
```

Die gesuchte Zeit beträgt somit rund 4 Stunden und 26 Minuten.

Leistungsaufgabe

Bei dieser Aufgabe handelt es sich um ein einfaches Dreisatzproblem.

Aufgabe:

Eine Anzahl von $n_1 = 8$ Bauarbeitern, alle mit gleicher Leistung, benötigt $t_1 = 87$ Tage, um ein Haus zu bauen. Wie viele Tage t_2 sind zum Hausbau nötig, wenn $n_2 = 24$ Bauarbeiter mit der selben Leistung daran arbeiten?

Lösung:

Diese Dreisatzaufgabe lässt sich als einfach Verhältnisgleichung darstellen. Da die insgesamt benötigte Zeit als indirekt proportional zur Anzahl der Arbeiter angenommen wird, ist das Verhältnis $\frac{t_1}{t_2}$ der benötigen Zeiten gleich dem Verhältnis $\frac{n_2}{n_1}$ der Arbeiterzahlen:

$$\frac{t_1}{t_2} = \frac{n_1}{n_2}$$

Diese Gleichung lässt sich auch ohne Computer-Algebra-System leicht nach t_2 auflösen (insbesondere, wenn man auf beiden Seiten die Kehrwerte bildet, d.h. die Gleichung $\frac{t_2}{t_1} = \frac{n_1}{n_2}$ betrachtet). Dennoch soll an dieser Stelle die Aufgabe als Beispiel für vielfach vorkommende Dreisatzaufgaben mit *Sympy* gelöst werden:

```
import sympy as sy
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
# Sympy-Variablen initiieren:
n1, n2 = sy.S( [8, 24] )
t1 = sy.S( 87 )
t2 = sy.S( 't2' )

# Gleichung formulieren:
equation = sy.Eq( t1/t2 , n2/n1 )

# Gleichung lösen:
result = sy.solve(equation)

# Ergebnis: [29]
```

Die gesuchte Zeit beträgt somit $t_2 = 29\,\text{Tage}$.

Aufeinander folgende Zahlen

Bei dieser Aufgabe geht es um das Lösen einer Summengleichung.

Aufgabe:

Betrachtet wird eine Folge x_n von aufeinander folgenden, ganzzahligen Werten ($x, n \in \mathbb{N}$).

Die Summe $\sum_{i=1}^{m_1} x_\text{i}$ der ersten m_1 Zahlen sei um einen Differenzwert d kleiner als die Summe $\sum_{i=m_1+1}^{n} x_\text{i}$ der restlichen Zahlen. Wie lässt sich diese Aufgabe mathematisch formulieren? Welche Lösung ergibt sich für $n = 5$, $m_1 = 2$ und $d = 42$?

Lösung:

Damit die obige Bedingung erfüllt ist, muss folgende Gleichung gelten:

$$\sum_{i=1}^{m_1}(x_i) + d = \sum_{i=m_1+1}^{n}(x_i)$$

Um diese Aufgabe mit `sympy` zu lösen, können beide Summen in der obigen Gleichung auch in folgender Form dargestellt werden:

$$\sum_{i=1}^{m_1}(x+i) + d = \sum_{i=m_1+1}^{n}(x+i)$$

Hierbei ist der zu bestimmende Initialwert x um 1 kleiner als der erste Wert der Zahlenreihe. Diese Darstellung hat den Vorteil, dass die Summen leichter formuliert werden können und die Gleichung nur noch eine Unbekannte aufweist. Der Quellcode zur Lösung der Gleichung mittels *Sympy* kann beispielsweise so aussehen:

```
import sympy as sy

# Sympy-Variablen initiieren:
n,m1,d = sy.symbols('n m1 d')
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
x,i = sy.symbols('x i')

# Terme festlegen
s1 = sy.summation(x + i, (i,1,m1))
s2 = sy.summation(x + i, (i,m1+1,n))

# Gleichungen formulieren:
equation = sy.Eq( s1 + d , s2)
equation_concrete = equation.subs({n:5,m1:2,d:42})

# Gleichung(en) lösen:
sy.solve(equation, x, 'dict')
sy.solve(equation_concrete, x, 'dict')

# Ergebnisse:

# Allgemein:
# [{x: (-d - m1**2 - m1 + n**2/2 + n/2)/(2*m1 - n)}]

# Konkret:
# [{x:33}]
```

Beim Lösen der Gleichung wurde hierbei explizit die Variable x als gesuchte Variable angegeben; ebenso könnte die Gleichung beispielsweise nach d für einen gegebenen Startwert x aufgelöst werden.

Das Ergebnis für die konkrete Aufgabe lautet $x = 33$. In diesem Fall sind die $n = 5$ Zahlen also gleich $(34, 35, 36, 37, 38)$, die Summe der ersten $m_1 = 2$ Zahlen ist $s_1 = 34 + 35 = 69$, die Summe der weiteren Zahlen ist gleich $36 + 37 + 38 = 111$.

Altersaufgabe

Bei dieser Aufgabe geht es um die Formulierung und das Lösung einer Funktionsgleichung.

Aufgabe:

Wenn K. heute $m = 3$ mal so alt wäre wie vor $n = 6$ Jahren, dann wäre K. nun $j = 38$ Jahre älter. Wie alt ist K. heute?

Lösung:

Die gesuchte Variable x gebe das heutige Alter von K. an. Dann lässt sich aus den obigen Bedingungen folgende Gleichung aufstellen:

$$m * (x - n) = x + j$$

Diese Gleichung kann folgendermaßen mit *Sympy* gelöst werden:

```
import sympy as sy
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
# Sympy-Variablen initiieren:
x = sy.S( 'x' )
m,n,j = sy.S([3, 6, 38] )

# Gleichung formulieren:
equation = sy.Eq( m * (x-n) , x + j )

# Gleichung lösen:
sy.solve(equation)

# Ergebnis: [28]
```

K. ist somit heute 28 Jahre alt.

Diskriminante

Bei dieser Aufgabe geht es darum, die Diskriminante einer quadratischen Gleichung zu bestimmen.

Aufgabe:

Gegeben sei die quadratische Gleichung $f(x) = x^2 - 8 \cdot x - 15$. Wie lautet die Diskriminante dieser Gleichung?

Lösung:

Die Diskriminante D einer quadratischen Gleichung $f(x) = a \cdot x^2 + b \cdot x + c$ lässt sich in Abhängigkeit der Parameter a, b und c folgendermaßen bestimmen:

$$D = b^2 - 4 \cdot a \cdot c$$

Die Aufgabe kann folgendermaßen mit *Sympy* gelöst werden:

```
import sympy as sy

# Sympy-Variable initiieren:
x = sy.S( 'x' )

# Gleichung formulieren:
f = x**2 - 8*x - 15

# Koeffizienten a, b und c bestimmen:

a = f.coeff(x, n=2)
b = f.coeff(x, n=1)
c = f.coeff(x, n=0)

D = b**2 - 4*a*c

# Ergebnis: 124
```

Die Diskriminante ist positiv, somit hat die Gleichung die zwei Lösungen $x_{1,2} = \frac{-b \pm \sqrt{D}}{2 \cdot a}$:

```
x1 = ( -b + sy.sqrt(D) ) / (2 * a)
x2 = ( -b - sy.sqrt(D) ) / (2 * a)

x1.evalf()
# Ergebnis:  9.56776436283002

x2.evalf()
# Ergebnis: -1.56776436283002
```

Der Funktionsgraph als Ganzes kann mittels numpy und matplotlib folgendermaßen erzeugt werden:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

# Wertereihen erzeugen:
x = np.arange(-3,11,0.01)
y = x**2 - 8*x - 15

# Funktion plotten:
plt.plot(x,y)

# Layout anpassen:
plt.axis([-3,11,-35,35])
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.grid(True)

plt.text(7, 15, "$f(x)=x^2 - 8x - 15$")

plt.show()
```

Das Ergebnis sieht so aus:

Quadratfläche aus Diagonale

Bei dieser Aufgabe geht es um die Formulierung und das Lösung einer geometrischen Funktionsgleichung.

Aufgabe:

Gegeben ist die Diagonale $d = 3 \cdot \sqrt{2}$ eines Quadrats. Wie lässt sich daraus die Fläche A des Quadrats berechnen?

Lösung:

Allgemein gilt für die Diagonale d eines Quadrats in Abhängigkeit von der Seitenlänge a:

$$d = \sqrt{2} \cdot a$$

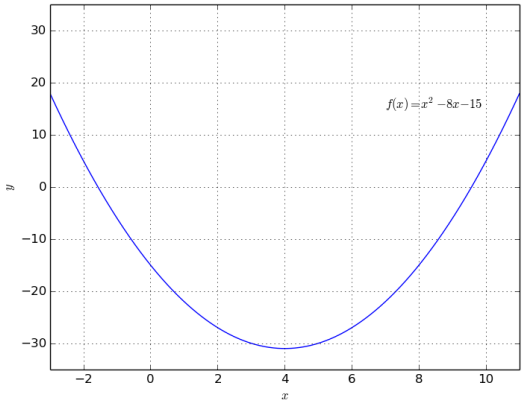


Abb. 9: Graph der Funktion $f(x) = x^2 - 8 \cdot x - 15$.

Umgekehrt gilt somit für die Seitenlänge a in Abhängigkeit von d:

$$a = \frac{d}{\sqrt{2}}$$

Somit lässt sich aus d die Seitenlänge a und damit die Fläche $A = a^2$ des Quadrats berechnen. Ein Beispielcode mit *Sympy* kann beispielsweise so aussehen:

```
import sympy as sy

# Sympy-Variablen initiieren:
d = sy.S( 3 * sy.sqrt(2) )
a = sy.S( 'a' )

# Seitenlänge berechnen:
a = d / sy.sqrt(2)

# Fläche berechnen:
A = a**2

# Ergebnis: 9
```

Die Fläche des Quadrats beträgt somit 9 Flächeneinheiten.

Quaderliste – Volumen und Oberfläche

Diese Aufgabe war im Original dafür vorgesehen, um in Maxima die Iteration über Listenelemente zu demonstrieren. Bei der Verwendung von `python` und `sympy` als Computer-Algebra-System hingegen genügt dafür bereits die Python-Standardbibliothek.

Aufgabe:

Gegeben ist die Liste `[(3,4,5), (10,8,6), (25,21,12)]`, deren Einträge jeweils die Maße eines Quaders angeben (Länge, Breite und Höhe). Berechne das Volumen sowie die Oberfläche der einzelnen Quader.

Lösung:

Das Volumen eines Quaders ist gleich dem Produkt aus Länge l, Breite b und Höhe h; seine Oberfläche ist gleich der doppelten Summe aller Flächen, die sich aus je zwei der drei Größenangaben berechnen lassen:

$$V_{\text{Quader}} = h \cdot b \cdot l$$

$$A_{\text{Quader}} = 2 \cdot (h \cdot b + b \cdot l + l \cdot h)$$

In Python können die gesuchten Größen (auch ohne `sympy`) beispielsweise durch Definition von geeigneten Funktionen berechnet werden:

```
# Variablen initiieren:
cuboid_dimensions = [(3,4,5), (10,8,6), (25,21,12)]
cuboid_surfaces = []
cuboid_volumes = []

# Funktionen definieren:
def get_cuboid_surface(length, width, height):
    """ Calculate the surface of a cuboid."""

    return 2 * (height * width + width * length +  length * height)

def get_cuboid_volume(length, width, height):
    """ Calculate the volume of a cuboid."""

    return length * width * height


# Funktionen auf Liste anwenden:
for c in cuboid_dimensions:

    cuboid_surfaces.append( get_cuboid_surface(*c) )
    cuboid_volumes.append( get_cuboid_volume(*c) )

# Ergebnis:

# cuboid_surfaces: [94, 376, 2154]
# cuboid_volumes:  [60, 400, 6300]
```

In der Hauptschleife werden beide Funktionen für die jeweiligen Größenangaben aufgerufen; die Hilfsvariable `c` wird dabei, da es sich um ein um eine Sequenz handelt, mit einem davor stehenden `*` an die Funktion übergeben. Dies bewirkt, dass nicht das Zahlentripel als eigenes Objekt verwendet wird, sondern vielmehr dessen Inhalte „ausgepackt“ und der Reihenfolge nach an die Funktion übergeben werden.

Quaderliste - minimales oder maximales Volumen

Diese Aufgabe lässt sich – ebenfalls wie die vorherige Aufgabe *Quaderliste – Volumen und Oberfläche* mit der Python-Standardbibliothek lösen. Zu bestimmen ist das Minimum bzw. Maximimum einer Liste an Werten.

Aufgabe:

Gegeben ist folgende Liste an Quadergrößen (Länge, Breite, Höhe):

[(3,4,5),(6,8,10),(1,2,4),(12,13,32), (14,8,22),(17,3,44),(12,5,3),(10,9,11)]

Bestimme das Minumum und Maximum der Quadervolumina, die sich anhand der obigen Längenmaße ergeben.

Lösung:

Das Volumen der einzelnen Quader lässt sich als Produkt der drei Längenangaben berechnen:

$$V_{\text{Quader}} = l \cdot b \cdot h$$

```
# Variablen initiieren:
cuboid_dimensions = [(3,4,5), (6,8,10), (1,2,4), (12,13,32), (14,8,22),
    (17,3,44), (12,5,3), (10,9,11)
    ]

# Volumina berechnen:
cuboid_volumina = [ l * b * h for l,b,h in cuboid_dimensions]

# Minimum und Maximum bestimmen:
cuboid_volume_min = min(cuboid_volume_max)
cuboid_volume_max = max(cuboid_volume_max)

# Ergebnis: 8 bzw. 4992
```

Zur Erstellung der Volumina-Liste wurden hierbei eine so genannte List Comprehension genutzt. Möchte man wissen, welches Listenelement zum Wert des minimalen bzw. maximalen Volumens gehört, so kann man die Listen-Index-Funktion nutzen:

```
# Position des Minimums bzw. Maximums in der Liste bestimmen:
cuboid_volumina.index(cuboid_volume_min)
cuboid_volumina.index(cuboid_volume_min)

# Ergebnis: 2 bzw. 3
```

Python beginnt mit der Indizierung von Listenelementen bei Null, so dass die Ergebniswerte dem dritten und vierten Listenelement entsprechen.

Quader

Bei dieser einfachen Aufgabe soll anhand der Länge, Breite und Höhe eines Quaders dessen Volumen, Oberfläche und Raumdiagonale bestimmt werden.

Aufgabe:

Bestimme zu den Massen $l = 10$, $b = 8$ und $c = 6$ das Volumen, die Oberfläche und die Raumdiagonale eines Quaders.

Lösung:

Die gesuchten Größen lassen sich folgendermaßen berechnen:

$$V_{\text{Quader}} = h \cdot b \cdot l$$

$$A_{\text{Quader}} = 2 \cdot (h \cdot b + b \cdot l + l \cdot h)$$

$$d_{\text{Quader}} = \sqrt{l^2 + b^2 + h^2}$$

Die rechte Seite der letzten Gleichung entspricht dem Betrag eines dreidimensionalen Vektors.

Zur Berechnung der Quaderdiagonale kann die Funktion `sqrt()` aus dem math-Modul genutzt werden:

```
import math as m
import functools as ft

cuboid_dimensions = [10,8,6]

cuboid_volume = ft.reduce(lambda x,y: x*y , cuboid_dimensions)

cuboid_surface = lambda
cuboid_surface = 2 * (
    cuboid_dimensions[0] * cuboid_dimensions[1] +
    cuboid_dimensions[0] * cuboid_dimensions[2] +
    cuboid_dimensions[1] * cuboid_dimensions[2]
)

cuboid_diagonal = m.sqrt(
    cuboid_dimensions[0]**2 +
    cuboid_dimensions[1]**2 +
    cuboid_dimensions[2]**2
)

# Ergebnisse:

# cuboid_volume:   480
# cuboid_surface:  376
# cuboid_diagonal: 14.142135623730951
```

Bei der Berechnung des Quadervolumens wurde, um Schreibarbeit zu sparen, die Funktion `reduce()` aus dem *functools*-Modul verwendet.

Anstelle des Lambda-Ausdrucks (quasi einer Funktion ohne Namen) kann auch die Funktion `mul()` aus dem Modul operator verwendet werden. Diese wertet das Produkt aller Werte einer (als eingigem Argument übergebenen) Liste aus:

```
import operator as op

ft.reduce(op.mul, [1,2,3]) == ft.reduce(lambda x,y: x*y, [1,2,3])

# Ergebnis: True
```

Zudem könnte die Schreibarbeit durch die Definition von Funktionen entsprechend Aufgabe *Quaderliste – Volumen und Oberfläche* weiter reduziert werden.

Punktspiegelung

Bei sollen für einem gegebenen Punkt die Koordinaten eines neuen Punktes bestimmt werden, der sich durch Spiegelung an der x- oder y-Achse ergibt.

Aufgabe:

Eine bestimmter Punkt P soll entweder um die x- oder y-Achse gespiegelt werden. Die Koordinaten des Punktes sowie die Wahl der Spiegelungsachse sollen durch den Benutzer eingegeben werden.

Lösung:

Die Koordinaten des gespiegelten Punkts lassen sich einfach berechnen, indem die jeweilige Koordinate mit `(-1)` multipliziert wird.

Um eine Eingabe von einem Benutzer zu erhalten, kann in Python die `input()`-Funktion genutzt werden, welche die eigegebene Textzeile als String-Variable speichert. Diese wird im folgenden Beispiels schrittweise überarbeitet, indem zunächst mögliche runde Klammern mit der String-Funktion `strip` entfernt werden, der String anschließend am Kommazeichen in eine Liste zweier Strings zerlegt wird, und zuletzt eine neue Liste mittels einer List Comprehension anhand der Elemente der bestehenden Liste erstellt wird. Dieser Schritt ist notwendig, um aus den als Strings gespeicherten Zahlenwerten Float-Variablen zu erzeugen.

```
# Koordinaten des ursprünglichen Punktes einlesen:
original_point = input("Bitte die Koordinaten des Punktes eingeben \
                        [ beispielsweise (3.0, -1.5) ]: ")
axis = input("Um welche Achse soll gespiegelt werden [x oder y]? ")

# Eingabe-String in Liste von Float-Werten umwandeln:
opoint = [float(num) for num in original_point.strip('()').split(',')]

new_point = []

if axis == 'x':

    new_point.append( opoint[0] * (-1) )
    new_point.append( opoint[1] )

if axis == 'y':

    new_point.append( opoint[0] )
    new_point.append( opoint[1] * (-1) )

print("Der an der %s-Achse gespiegelte Punkt hat die \
        Koordinaten %s" % (axis, tuple(new_point)) )
```

Der obige Code zeigt nur die grundlegende Logik auf. In einem „richtigen“ Programm sollte die Eingabe nach falscher Syntax hin untersucht und gegebenenfalls ein entsprechendes Exception-Handling vorgenommen werden.

Parabel – Scheitel und Öffnung

Bei dieser Übungsaufgabe geht es um das Differenzieren einer quadratischen Funktion.

Aufgabe:

Der Scheitel einer Parabel ist zu bestimmen. Es ist zu entscheiden, ob die Parabel nach unten oder nach oben geöffnet ist.

Lösung:

Die Orientierung einer Parabel kann man am Koeffizienten ihres quadratischen Terms ablesen; ist dieser positiv, so ist die Parabel nach oben, andernfalls nach unten geöffnet.

Der Scheitelpunkt ist das einzige Extremum einer Parabel. Um ihn zu bestimmen, bildet man daher die erste Ableitung der quadratischen Funktion und setzt diese gleich Null. So erhält man den x-Wert des Scheitelpunktes. Den zugehörigen y-Wert erhält man, indem man den x-Wert des Scheitelpunktes in die ursprüngliche Funktionsgleichung einsetzt.

Mit *Sympy* kann die Aufgabe folgendermaßen gelöst werden:

```
import sympy as sy

# Funktionsgleichung als String einlesen:
parable_function_input = input("Bitte die Funktionsgleichung einer Parabel \
                               eingeben (beispielsweise 3*x**2 - 5*x +7): ")

# Eingelesenen String in Sympy-Ausdruck umwandeln:
parable_function = sy.S(parable_function_input)

# Orientierung der Parabel bestimmen:
if parable_function.coeff(x, n=2) > 0:
    orientation = "up"
else:
    orientation = "down"

# Erste und zweite Ableitung bilden:
parable_function_diff_1 = sy.diff(parable_function, x, 1)
parable_function_diff_2 = sy.diff(parable_function, x, 2)

# Extremstelle bestimmen (liefert Liste an Ergebnissen):
extremes = sy.solve(sy.Eq(parable_function_diff_1, 0))

# Der gesuchte x_0-Wert ist der einzige Eintrag der Ergebnisliste:
x_0 = extremes[0]

# Zugehörigen Funktionswert bestimmen:
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
y_0 = parable_function.subs(x, x_0)

print("Die Orientierung der Parabel ist \"%s\", ihr Scheitel liegt bei \
      (%.2f , %.2f)." % (orientation, x_0, y_0) )
```

Lage eines Kreises und einer Gerade

Bei dieser Übungsaufgabe geht es das Gleichsetzen zweier geometrischer Gleichungen.

Aufgabe:

Für eine Kreis- und eine Geradengleichung ist die Lagebeziehung (Sekante, Tangente, Passante) der beiden geometrischen Objekte zueinander zu bestimmen.

Lösung:

- Wenn eine Gerade einen Kreis in zwei Punkten schneidet, dann ist die Gerade eine Sekante.
- Wenn eine Gerade einen Kreis in einem Punkt berührt, dann ist die Gerade eine Tangente.
- Wenn die Gerade und der Kreis keinen Punkt gemeinsam haben, dann ist die Gerade eine Passante.

Mit *Sympy* kann die Aufgabe folgendermaßen gelöst werden:

```
import sympy as sy

# Funktionsgleichungen als Strings einlesen:
circle_equation_input = input("Bitte die Funktionsgleichung eines Kreises \
                              eingeben (beispielsweise x**2 + y**2 = 9): ")
linear_equation_input = input("Bitte die Funktionsgleichung einer Geraden \
                              eingeben (beispielsweise 3*x - 2*y   = 4): ")

# Eingelesenen Strings in Sympy-Ausdrücke umwandeln:
circle_equation_elements = [sy.S(item) for item in \
                            circle_equation_input.replace("=", ",").split(",")]
linear_equation_elements = [sy.S(item) for item in \
                            linear_equation_input.replace("=", ",").split(",")]

# Gleichungssystem aus Sympy-Ausdrücken erstellen:
equations = [
    sy.Eq(*circle_equation_elements),
    sy.Eq(*linear_equation_elements)
]

# Gleichungssystem lösen
solutions = sy.solve(equations)
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
if len(solutions) == 2:
    print("Die Gerade ist eine Sekante.")
elif len(solutions) == 1:
    print("Die Gerade ist eine Tangente.")
else:
    print("Die Gerade ist eine Passante.")
```

Beim Erstellen des Gleichungssystems wurde bei der Übergabe der einzelnen Sympy-Elemente der Stern-Operator `*` vorangestellt, um nicht die Liste, sondern deren Inhalt an die `Eq()`-Funktion zu übergeben.

`Bottle` – Ein Mikro-Framework für interaktive Webseiten

Das bottle-Modul bietet eine einfache Möglichkeit zum schnellen Erstellen von WSGI-basierten Webseiten („Web Server Gateway Interface“). Die eigentliche Anwendung kann dabei aus einer einzigen Datei bestehen.

Das `bottle`-Modul lässt sich unter Linux folgendermaßen installieren:

```
sudo aptitude install python3-bottle
```

Alternativ hierzu kann man `bottle` auch, sofern man das Paket `python3-setuptools` via `aptitude` installiert hat, mittels `pip3 install bottle` installieren.

Ein „Hallo Welt“-Beispiel

Um mittels Bottle eine einfache Webanwendung zu programmieren, genügt es, das gleichnamige `bottle`-Paket oder einzelne Funktionen daraus in eine Python-Datei zu importieren. Ein einfaches Code-Beispiel sieht somit etwa folgendermaßen aus:

```
#!/usr/bin/env python3

from bottle import route, debug, run

@route('/hallo/<name>')
def hallo(name):
    return 'Hallo {0}!'.format(name)

debug(True)
run()
```

Speichert man dieses Programm beispielsweise als Datei `hallo-welt.py` und ruft es aus einer Shell heraus mittels `python3 hallo-welt.py` auf, so kann man sich das Ergebnis im Webbrowser unter der Adresse `http://localhost:8080/hallo/Welt` anzeigen lassen. Gibt man in diesem Pfad einen anderen Namen als „Welt“ an, so bekommt man im Webbrowser eine entsprechend andere Begrüßung angezeigt.

Die Funktionsweise der Bottle-Anwendung liegt darin, einen Browserpfad über die `route()`-Funktion mit einer gewöhnlichen Python-Funktion zu verbinden. Über die

`return`-Anweisung kann wahlweise ein einfacher Text im Browser ausgegeben oder auch eine andere Funktion aufgerufen werden, die dann beispielsweise ein HTML-Template rendert und mit Text füllt.

Die `run()`-Funktion startet den von Bottle ohne weitere Abhängigkeiten bereitgestellten WSGI-Server mit dem üblichen HTML-Standard-Port `8080`; man kann auch mittels beispielsweise `run(port=8081)` einen anderen Localhost-Port vorgeben. Ruft man die Funktion `run()` mit der Option `reloader=True` auf, so werden Änderungen unmittelbar, also auch ohne Neustart des WSGI-Servers übernommen.

HTML-Templates

Möchte man nicht nur reinen Text im Webbrowser anzeigen, sondern eine Ausgabe in HTML-Form erreichen, so kann wahlweise die im `bottle`-Modul bereits integrierte SimpleTemplate-Engine genutzt werden; als Alternative dazu können auch Jinja2, ‘ oder Mako eingesetzt werden, welche mittels `pip3` und den gleichen Paketnamen nachinstalliert werden können (`pip3 install Jinja2`).

. . . to be continued . . .

https://www.fullstackpython.com/wsgi-servers.html

Links:

- Developing with Bottle (RealPython)

Kivy - ein Toolkit für GUI-Programme

Kivy ist eines von mehreren Toolkits, mit denen sich in Python Programme mit einer graphischen Bedienoberfläche (Graphical User Interface, kurz: GUI) programmieren lassen.[1] Um Kivy mit Python3 nutzen zu können, müssen die folgenden Pakete installiert werden:

```
# Aktuelle Paketquellen hinzufügen:
sudo add-apt-repository ppa:kivy-team/kivy

# Paketquellen auf mögliche Updates prüfen:
sudo aptitude update

# Kivy-Pakete installieren:
sudo aptitude install cython3 python3-kivy python3-kivy-bin python3-kivy-common

# Kivy-Beispiele installieren (optional):
sudo aptitude install kivy-examples
```

Installiert man auch das Paket `kivy-examples`, so können im Verzeichnis `/usr/share/kivy-examples` einige fertige Beispiel-Programme ausprobiert und als Vorlage genutzt werden.

Ein „Hallo Welt“-Beispiel

Mit Kivy geschriebene Programme haben stets eine `App`-Klasse, die am einfachsten mittels *Vererbung* auf der Standard-App-Klasse aufbauen kann. Diese Standard-Klasse stellt unter anderem eine `run()`- und eine `build()`-Methode bereit, wobei letztere automatisch aufgerufen wird, wenn das Programm mittels der `run()`-Methode gestartet wird. Die `build()`-Funktion wiederum startet automatisch das graphische Bedienfenster, und kann zugleich weitere Elemente in diesem Fenster platzieren; sie sollte als Ergebnis stets das „Root“-Widget der Anwendung liefern.

Im einfachsten Fall soll ein schlichter „Button“ mittig im Fenster platziert werden, der die Bezeichnung „Hallo Welt!“ trägt. Ein erstes vollständiges Programm, das wiederum auf ein bereits vordefiniertes Button-Element zurückgreift, kann damit folgendermaßen aussehen:

[1] Weiter GUI-Toolkits für Python sind das in der Standard-Installation enthaltene Tkinter, sowie PyGTK, PyQT und wxPython (siehe Übersicht). Kivy ist allerdings das einzige in Python selbst geschriebene Toolkit und bietet somit eine sehr python-typische Syntax.

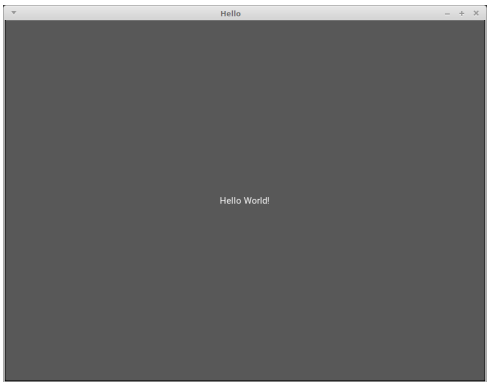
```
# Datei: hello.py

import kivy
kivy.require('1.9.0') # Mindest-Version von Kivy

from kivy.app import App
from kivy.uix.button import Button


class HelloApp(App):

    def build(self):
        return Button(text='Hallo Welt!')

if __name__== "__main__":
    HelloApp().run()
```

Das `uix`-Modul stellt allgemeine GUI-Elemte bereit, wie beispielsweise Widgets und Layouts.

Bei einem Aufruf der Programmdatei wird durch die *if-main-Abfrage* eine Instanz der App erzeugt und diese gestartet, wobei wiederum die `build()`-Funktion aufgerufen wird:

```
python3 hello.py
```

Damit erscheint folgendes „Anwendungs“-Fenster:



Dieses erste Programm hat keine weitere Funktion; es kann beendet werden, indem auf das `X`-Symbol geklickt wird oder in der Shell die Anwendung mittels `Ctrl c` unterbrochen wird.

Design-Anpassungen mittels einer `.kv`-Datei

Das Layout einer Anwendung sollte für eine bessere Übersichtlichkeit, Anpassungsfähigkeit und Wiederverwertbarkeit von der eigentlichen „Logik“ des Programms getrennt sein („Model-View-Controller“). Mit Kivy wird dieser Grundsatz in einer Art und Weise verfolgt, der stark an die Anpassung des Layouts einer Webseite mittels CSS erinnert: Das

Aussehen der einzelnen graphischen Elemente eines Programms wird über eine entsprechende `.kv`-Datei festgelegt werden.

Beim Aufruf der Programm-Datei wird automatisch diejenige `.kv`-Datei im gleichen Verzeichnis geladen, deren Namen mit (in Kleinbuchstaben) dem Namen der App-Klasse übereinstimmt: Hat die App-Klasse beispielsweise den Namen `HelloWorldApp`, so heißt die zugehörige `.kv`-Datei `helloworld.kv`. In dieser Datei kann mit einer YAML-artigen Syntax das Design einzelner „Widgets“ (Teilbereiche des Hauptfensters) sowie deren Anordnung festgelegt werden.

... to be continued ...

Links

- Kivy Documentation (pdf, en.)
- API Referenz (en.)
- Kivy Language (en.)
- Kivy Beispiele (en.)

Anhang

Schlüsselwörter

In Python3 sind folgende Schlüsselwörter vordefiniert:

False	assert	del	for	in	or	while
None	break	elif	from	is	pass	with
True	class	else	global	lambda	raise	yield
and	continue	except	if	nonlocal	return	
as	def	finally	import	not	try	

Die in der obigen Tabelle angegebenen Wörter können nicht als Variablen- oder Funktionsnamen verwendet werden. Mittels der Funktion `iskeyword()` aus dem Modul `keyword` kann getestet werden, ob eine Zeichenkette ein Schlüsselwort ist:

```
from keyword import iskeyword()

iskeyword("lambda")
# Ergebnis: True
```

Standardfunktionen

Die im folgenden Abschnitt beschriebenen Funktionen (Builtin-Funktionen) sind standardmäßig in Python definiert, ohne dass ein zusätzliches Modul geladen werden muss.

abs()

Die Funktion abs(x) gibt den Absolutwert einer Zahl `x` als Ergebnis zurück.

Beispiel:

```
abs( -5.7 )
# Ergebnis: 5.7

abs( +5.7 )
# Ergebnis: 5.7
```

all()

Die Funktion all(sequenz) kann auf ein beliebiges iterierbares Objekt (Listen oder Mengen) angewendet werden. Als Ergebnis wird `True` zurückgegeben, wenn alle Elemente den Wahrheitswert `True` besitzen; andernfalls wird `False` als Ergebnis zurückgegeben.

Beispiel:

```
all( [1,3,5,0,7] )
# Ergebnis: False

all( [3,7,9,5,2] )
# Ergebnis: True
```

any()

Die Funktion any(sequenz) kann auf ein beliebiges iterierbares Objekt (Listen oder Mengen) angewendet werden. Als Ergebnis wird `True` zurückgegeben, wenn zumindest ein Element den Wahrheitswert `True` besitzt; andernfalls wird `False` als Ergebnis zurückgegeben.

Beispiel:

```
any( [0,0,0,0,0] )
# Ergebnis: False

any( [0,0,0,1,0] )
# Ergebnis: True
```

ascii()

Die Funktion ascii(objekt) gibt ebenso wie die Funktion *repr()* als Ergebnis eine Zeichenkette zurück, die eine kurze charakteristische Beschreibung des Objekts beinhaltet; häufig entspricht dies einer Angabe der Objekt-Klasse, des Objekt-Namens und der Speicheradresse.

Beispiel:

```
ascii(print)
# Ergebnis: '<built-in function print>'
```

Ist in der Klasse des angegebenen Objekts eine `__repr__()`-Methode definiert, so ist `repr(objekt)` identisch mit `objekt.__repr__()`. Als Zeichensatz wird für die Ausgabe des Strings allerdings der ASCII-Zeichensatz verwendet, so dass darin nicht enthaltene Symbole durch Zeichen mit vorangestelltem `\x`, `\u` oder `\U` gekennzeichnet werden.

bin()

Die Funktion bin(x) gibt eine Zeichenkette mit der Binärdarstellung einer einer Integer-Zahl als Ergebnis zurück. Eine solche Zeichenkette wird mit `0b` eingeleitet, gefolgt von der eigentlichen Binärzahl.

Beispiel:

```
bin(42)
# Ergebnis: '0b101010'
```

bool()

Die Funktion bool(ausdruck) gibt den Wahrheitswert eines logischen Ausdrucks an; dieser kann entweder `True` oder `False` sein. Als Argument kann entweder ein mittels :ref`Vergleichsoperatoren <Operatoren>` erzeugter logischer Ausdruck oder auch ein einzelnes Objekt übergeben werden.

- Listen, Tupel und Zeichenketten haben den Wahrheitswert `True`, wenn sie nicht leer sind beziehungsweise mindestens ein Zeichen enthalten.
- Zahlen haben dann den Wahrheitswert `True`, wenn sie nicht gleich Null sind.
- `bool(None)` liefert den Wahrheitswert `False`.

Beispiel:

```
bool(-3)
# Ergebnis: True
```

callable()

Die Funktion callable(objekt) gibt in Form eines booleschen Wahrheitswertes an, ob das als Argument übergebene Objekt (wie eine Funktion oder Methode) aufrufbar ist oder nicht.

Beispiel:

```
callable(5)
# Ergebnis: False

callable(print)
# Ergebnis: True
```

chr()

Die Funktion chr(zahl) gibt zu einem angegebenen Ganzzahl-Wert mit positivem Vorzeichen das entsprechende Unicode-Zeichen aus.

Beispiel:

```
chr(65)
# Ergebnis: 'A'

chr(97)
# Ergebnis: 'a'
```

Für viele Programme reichen die *ASCII-Codes* als Teilmenge des Unicode-Zeichensatzes bereits aus.

classmethod()

Die Funktion classmethod(methode) macht die angegebene Methode zu einer so genannten Klassen-Methode. Üblicherweise wird die `classmethod()`-Funktion als *Funktionsdekorator* verwendet:

```
class C():

    @classmethod
    def my_class_method(cls, arguments):

        pass
```

Bei einer so definierten Methode wird die als erstes Argument der Name der Klasse angegeben, von der aus die Methode aufgerufen wird. Die Klassen-Methode des obigen Beispiels kann dann wahlweise mittels `C.my_class_method()` oder ausgehend von einer Instanz der Klasse, also mittels `C().my_class_method()` aufgerufen werden; im letzteren Fall wird beim Aufruf nur der Name der Instanzklasse, nicht die Instanz selbst als erstes Argument an die Methode übergeben.

Wird eine Klassen-Methode von einer Instanz einer Klasse aufgerufen, welche die Methode lediglich über eine *Vererbung* erhalten hat, so wird beim Aufruf dennoch der Name der konkreten Instanzklasse (und nicht der Basis-Klasse) übergeben.

compile()

Die Funktion compile(code, file, mode) übersetzt den als erstes Argument angegebenen Code-String in ein ausführbares, in Maschinensprache geschriebenes Bytecode-Objekt. Als zweites Argument muss der Pfad einer Datei angegeben werden, in die gegebenenfalls auftretende Fehler geschrieben werden sollen. Als drittes Argument muss entweder zum Kompilieren genutzte Modus angegeben werden:

- `single`, wenn es sich bei dem angegebenen Code um eine einzelne Aussage-Komponente (beispielsweise den Wert einer Variablen) handelt;
- `eval`, wenn der angegebene Code eine einzelne Aussage darstellt;
- `exec`, wenn der angegebene Code aus einer oder mehreren Aussagen besteht und als Ergebnis `None` liefern soll.

Der compilierte Bytecode kann anschließend mittels *eval()* beziehungsweise *exec()* ausgeführt werden.

Beispiel:

```
# Bytecode erzeugen:

a = 5

compile('a', 'tmp.txt', 'single')
# Ergebnis: <code object <module> at 0x7f38edc91f60, file "tmp.txt", line 1>

compile('print("Hallo Welt!")', 'tmp.txt', 'eval')
# Ergebnis: <code object <module> at 0x7f38edc91c00, file "tmp.txt", line 1>

compile('for i in range(3): print(i)', 'tmp.txt', 'exec')
# Ergebnis: <code object <module> at 0x7f38edc94780, file "tmp.txt", line 1>

# Bytecode ausführen:

eval( compile('a', 'tmp.txt', 'single') )
# Rückgabewert / Ergebnis: 5

eval( compile('print("Hallo Welt!")', 'tmp.txt', 'eval') )
# Rückgabewert / Ergebnis: Hallo Welt!

exec( compile('for i in range(3): print(i)', 'tmp.txt', 'exec') )
# Rückgabewert: None
# Ergebnis (auf dem Bildschirm):
# 0
# 1
# 2
```

complex()

Die Funktion complex() erstellt eine neue Instanz einer *komplexen Zahl* aus zwei angegebenen Zahlen oder einem angegebenen String.

Beispiel:

```
complex(1.5, 2)
# Ergebnis: (1.5+2j)
```

Wird ein String als Argument angegeben, so muss darauf geachtet werden, dass kein Leerzeichen zwischen dem Realteil, dem Pluszeichen und dem Imaginärteil steht; `complex()` löst sonst einen `ValueError` aus.

delattr()

Die Funktion delattr(objekt, attributname) löscht ein angegebenes Attribut beziehungsweise einen angegebenen Funktionsnamen (eine Zeichenkette) aus dem als erstes Argument angegebenen Objekt; dies ist formal identisch mit `del objekt.attributname`.

```
import math as m

# Attribut löschen:
delattr(m, 'cos')

# Test:
m.cos( m.pi/4 )
# Ergebnis: 'module' object has no attribute 'cos'
```

dict()

Die Funktion dict() erzeugt eine neue Instanz eines *dict*-Objekts, also ein Dictionary. Formal ist `d = dict()` somit identisch mit `d = {}`.

Beispiel:

```
# Neues dict erzeugen:
d = dict()

# Schlüssel-Wert-Paar hinzufügen:
d['test'] = 'Hallo Welt!'

d
# Ergebnis: {'test': 'Hallo Welt!'}
```

dir()

Die Funktion dir() gibt, wenn sie ohne ein angegebenes Argument aufgerufen wird, eine Liste mit den Namen aller in der aktuellen Python-Sitzung definierten Objekt-Namen (als Strings) zurück.

Wird als Argument ein beliebiges Objekt angegeben, so werden die Attribute und Methoden des jeweiligen Objekts in Form einer String-Liste ausgegeben.

Beispiel:

```
import math as m

dir(m)
# Ergebnis:
# ['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
# 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
# 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
# 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf',
# 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi',
# 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

divmod()

Die Funktion divmod(zahl1, zahl2) dividiert die als erstes Argument angegebene Zahl durch die zweite Zahl. Als Ergebnis gibt die Funktion ein Tupel zweier Werte zurück, wobei der erste Wert das ganzzahlige Ergebnis der Division und der zweite Wert den Divisionsrest angibt.

Beispiel:

```
divmod(14,5)
# Ergebnis: (2, 4)
```

enumerate()

Die Funktion enumerate(sequenz) ermöglicht es, die Einträge einer Liste oder eines Tupels zu nummerieren. Damit lassen sich beispielsweise `for`-Schleifen über die Elemente einer Liste konstruieren, in denen beide Informationen verwendet werden.

Beispiel:

```
liste = [5, 6, 7, 8, 9]

for i, num in enumerate(liste):
    print( "Der {}. Eintrag in der Liste ist {}".format(i, num) )

# Ergebnis:
# Der 0. Eintrag in der Liste ist 5
# Der 1. Eintrag in der Liste ist 6
# Der 2. Eintrag in der Liste ist 7
# Der 3. Eintrag in der Liste ist 8
# Der 4. Eintrag in der Liste ist 9
```

eval()

Die Funktion eval(zeichenkette) erstellt aus der angegebenen Zeichenkette den entsprechenden Python-Ausdruck und wertet diesen aus; es darf sich dabei allerdings nur um einen einzelnen Ausdruck, nicht um ein aus vielen einzelnen Aussagen zusammengesetztes Code-Stück handeln.

Der Rückgabewert von `eval()` entspricht dabei dem Ergebnis des ausgewerteten Ausdrucks.

Beispiel:

```
x = 1

eval('x+1')
# Rückgabewert / Ergebnis: 2

eval('for i in range(3): print(i)')
# Ergebnis:
# for i in range(3): print(i)
#   ^
# SyntaxError: invalid syntax
```

Die Funktion `eval()` kann ebenso verwendet werden, um einen mittels `compile()` erzeugten Ausdruck auszuwerten. Wurde als Compilier-Modus hierbei `'single'` oder `eval` angegeben, so entspricht der Rückgabewert wiederum dem Ergebnis des Ausdrucks; bei der Angabe von `exec()` als Compilier-Modus liefert `eval()` als Ergebnis stets den Wert `None`.

exec()

Die Funktion exec(zeichenkette) führt – ähnlich wie `eval()` – einen (beispielsweise mittels *compile()* konstruierten) Python-Ausdruck aus; es kann sich dabei auch um eine beliebig lange Zusammensetzung einzelner Python-Ausdrücke handeln. Als Ergebnis wird stets `None` zurückgegeben.

Beispiel:

```
exec('print("Hallo Welt!")')
# Rückgabewert: None
# Ergebnis (Auf dem Bildschirm):
# Hallo Welt!

exec('for i in range(3): print(i)')
# Rückgabewert: None
# Ergebnis (Auf dem Bildschirm):
# 0
# 1
# 2

exec('42')
# Rückgabewert / Ergebnis: None
```

Die Funktion `exec()` kann ebenso verwendet werden, um einen mittels `compile()` erzeugten Ausdruck auszuwerten; auch hierbei ist der Rückgabewert stets `None`.

filter()

Die Funktion filter(funktionsname, objekt) bietet die Möglichkeit, eine Filter-Funktion auf alle Elemente eines iterierbaren Objekts (beispielsweise einer Liste) anzuwenden. Als Ergebnis gibt die `filter()`-Funktion ein iterierbares Objekt zurück. Dieses kann beispielsweise für eine `for`-Schleife genutzt oder mittels `list()` in eine neue Liste umgewandelt werden.

Beispiel:

```
my_list = [1,2,3,4,5,6,7,8,9]

even_numbers = filter(lambda x: x % 2 == 0, my_list)

list(even_numbers)
# Ergebnis: [2,4,6,8]
```

Oftmals kann anstelle der `filter()`-Funktion allerdings auch eine (meist besser lesbare) *List-Comprehension* genutzt werden. Im obigen Beispiel könnte auch kürzer `even_numbers = [x for x in my_list if x % 2 == 0]` geschrieben werden.

float()

Die Funktion float() gibt, sofern möglich, die zur angegebenen Zeichenkette oder Zahl passende Gleitkomma-Zahl als Ergebnis zurück; wird eine `int`-Zahl als Argument übergeben, so wird die Nachkommastelle `.0` ergänzt.

Beispiel:

```
float(5)
# Ergebnis: 5.0

float('3.2')
# Ergebnis: 3.2

float('1e3')
# Ergebnis: 1000.0
```

format()

Die Funktion format(wert, formatangabe) formatiert die Ausgabe des angegebenen Werts. Hierzu können als Format-Angabe die für die *Formatierung von Zeichenketten* üblichen Symbole verwendet werden. Wird kein Format angegeben, so wird die in der Objektklasse des Werts definierte Funktion `wertklasse.__format__()` aufgerufen.

Beispiel:

```
# Zeichenkette zentriert ausgeben (Gesamtbreite 20):
format('Hallo Welt!', '^20')
# Ergebnis: '    Hallo Welt!     '

# Zeichenkette rechtsbündig ausgeben (Gesamtbreite 20):
format('Hallo Welt!', '>20')
# Ergebnis: '         Hallo Welt!'

# Zahl Pi mit drei Stellen Genauigkeit ausgeben:
format(m.pi, '.3')
# Ergebnis: 3.14
```

frozenset()

Die Funktion frozenset(sequenz) erzeugt aus der angegebenen Sequenz (beispielsweise einer Liste oder einer Zeichenkette) eine neue Instanz eines *frozenset*-Objekts, also eine unveränderliche Menge.

Beispiel:

```
frozenset( [1, 3, 5, 7, 9, 9] )
# Ergebnis: frozenset({1, 3, 5, 7, 9})

frozenset( "Hallo Welt!" )
# Ergebnis: frozenset({' ', '!', 'H', 'W', 'a', 'e', 'l', 'o', 't'})
```

getattr()

Die Funktion getattr(objekt, attributname) gibt als Ergebnis den Wert von `objekt.attributname` zurück. Als drittes Argument kann optional ein Standard-Wert angegeben werden, der als Ergebnis zurück gegeben wird, wenn das angegebene Attribut nicht existiert.

Beispiel:

```
# Beispiel-Klasse:
class Point():

    x = 5
    y = 4

# Punkt-Objekt erzeugen:
p = Point()

getattr(p, 'x')
# Ergebnis: 5
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
getattr(p, 'y')
# Ergebnis: 4

getattr(p, 'z', 0)
# Ergebnis: 0
```

Wird kein Standard-Wert angegeben und das Attribut existiert nicht, so wird ein `AttributeError` ausgelöst.

globals()

Die Funktion globals() liefert als Ergebnis ein `dict` mit den Namen und den Werten aller zum Zeitpunkt des Aufrufs existierenden globalen, das heißt programmweit sichtbaren Variablen.

Beispiel:

```
globals()
# Ergebnis:
# {'__doc__': None, '__spec__': None, '__name__': '__main__',
# '__package__': None, # '__loader__': <class '_frozen_importlib.BuiltinImporter
↪'>,
# '__builtins__': <module 'builtins' (built-in)>}
```

hasattr()

Die Funktion hasattr(objekt, attributname) gibt als Ergebnis den Wahrheitswert `True` zurück, falls für das angegebene Objekt ein Attribut mit dem angegebenen Namen existiert, andernfalls `False`.

Beispiel:

```
# Beispiel-Klasse:
class Point():

    x = 5
    y = 4

# Punkt-Objekt erzeugen:
p = Point()

hasattr(p, 'x')
# Ergebnis: True

hasattr(p, 'y')
# Ergebnis: True
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
getattr(p, 'z')
# Ergebnis: False
```

Mittels der Funktion `hasattr()` kann somit geprüft werden, ob die Funktion `getattr()` beim Aufruf einen `AttributeError` auslösen wird oder nicht.

hash()

Die Funktion hash(unveraenderliches-objekt) liefert zu beliebigen nicht veränderlichen Python-Objekten (beispielsweise Zeichenketten oder Tupeln) einen eindeutigen Integer-Wert als Ergebnis zurück; dieser ist nicht abhängig von der aktuellen Python-Sitzung. Identische Objekte werden durch die `hash()`-Funktion also auf identische ganzzahlige Werte abgebildet.

Beispiel:

```
hash("Hallo Welt!")
# Ergebnis: -2446188496090613429

hash( (1, 3, 5, 7, 9) )
# Ergebnis: -4331119994873071480
```

Die Umkehrung ist leider nicht zwingend eindeutig: Zu einem Hash-Wert können unterschiedliche Objekte gehören.

help()

Die Funktion help(objekt) blendet im Interpreter eine Hilfe-Seite zum angegebenen Objekt ein, sofern eine Dokumentation zum angegebenen Objekt vorhanden ist.

Beispiel:

```
# Hilfe zu Zeichenketten (str) anzeigen:
help(str)

# Hilfe zur Funktion print() anzeigen:
help(str)
```

hex()

Die Funktion hex(int-wert) gibt eine Zeichenkette mit der Hexadezimal-Darstellung einer Integer-Zahl als Ergebnis zurück. Eine solche Zeichenkette wird mit `0x` eingeleitet, gefolgt von der eigentlichen Binärzahl.

Beispiel:

```
hex(42)
# Ergebnis: '0x2a'
```

id()

Die Funktion id(objekt) liefert für beliebige Python-Objekte, abhängig von der aktuellen Python-Sitzung, einen eindeutigen Integer-Wert als Ergebnis zurück; dieser Wert entspricht der Adresse, an der das Objekt im Speicher abgelegt ist.

Beispiel:

```
id("Hallo Welt!")
# Ergebnis: 139882484400688
```

Mittels der Funktion `id()` können somit zwei Objekte auf Gleichheit getestet werden.

input()

Die Funktion input() dient zum Einlesen einer vom Benutzer eingegebenen Zeichenkette. Beim Aufruf kann dabei optional ein String angegeben werden, der dem Benutzer vor dem Eingabe-Prompt angezeigt wird:

```
answer = input("Bitte geben Sie Ihren Namen an: ")

print("Ihr Name ist %s." % answer)
```

Soll eine Zahl eingelesen werden, so muss die Benutzerantwort mittels `int()` bzw. `float()` explizit von einem String in eine solche umgewandelt werden.

int()

Die Funktion int() gibt, sofern möglich, die zur angegebenen Zeichenkette oder Gleitkomma-Zahl passende Integer-Zahl als Ergebnis zurück; wird eine `float`-Zahl als Argument übergeben, so werden mögliche Nachkommastellen schlichtweg ignoriert, beispielsweise ergibt `int(3.7)` den Wert `3`.

Beispiel:

```
int('5')
# Ergebnis: 5

int(3.14)
# Ergebnis: 3
```

isinstance()

Die Funktion isinstance(objekt, klassen-name) gibt als Ergebnis den Wahrheitswert `True` zurück, wenn das angegebene Objekt eine Instanz der als zweites Argument angegebenen Klasse (oder einer *Subklasse*) ist; ist dies nicht der Fall, wird `False` als Ergebnis zurückgegeben.

Gibt man als zweites Argument eine Liste mit Klassennamen an, so wird geprüft, ob das angegebene Objekt eine Instanz *einer* der in der Liste angegebenen Klassen ist:

Beispiel:

```
isinstance("Hallo Welt", str)
# Ergebnis: True

isinstance(3.14, [int, float])
# Ergebnis: True
```

issubclass()

Die Funktion issubclass(cls1, cls2) gibt als Ergebnis den Wahrheitswert `True` zurück, wenn die als erstes Argument angegebene Klasse eine *Subklasse* der als zweites Argument angegebenen Klasse ist; ist dies nicht der Fall, wird `False` als Ergebnis zurückgegeben.

Beispiel:

```
isinstance(str, object)
# Ergebnis: True
```

iter()

Die Funktion iter(sequenz) erstellt eine neue Instanz eines Iterator-Objekts aus einer listen-artigen Sequenz (genauer: einem Objekt mit einer `__iter__()`-Methode). Dieser Iterator kann beispielsweise verwendet werden, um eine `for`-Schleife über die in der Sequenz vorkommenden Elemente zu konstruieren:

Beispiel:

```
# Iterator generieren:
iterator = iter( ['Hallo', 'Welt'] )

# Elemente des Iterator-Objekts ausgeben:
for i in iterator:
    print(i)

# Ergebnis:
# Hallo
# Welt
```

Die einzelnen Elemente eines Iterator-Objekts können auch schrittweise mittels `iteratorname.__next__()` aufgerufen werden; ist man am Ende der Sequenz angekommen, so wird ein `StopIteration`-Error ausgelöst.

Eine zweite Verwendungsmöglichkeit der `iter()`-Funktion besteht darin, als erstes Objekt einen Funktions- oder Methodennamen und als zweites Argument eine Integer-Zahl als „Grenzwert“ anzugeben. Wird ein solcher „aufrufbarer“ Iterator mit `iteratorname.__next__()` aufgerufen, so wird die angegebene Funktion so lange aufgerufen, bis diese einen Rückgabewert liefert, der mit dem angegebenen Grenzwert identisch ist. Wird der Grenzwert nicht erreicht, so kann der Iterator beliebig oft aufgerufen werden.

Beispiel:

```
import random

# Aufrufbaren Iterator generieren:
iterator = iter( random.random, 1 )

# Zufallszahlen ausgeben:

iterator.__next__()
# Ergebnis: 0.17789467192460118

iterator.__next__()
# Ergebnis: 0.7501975823469289
```

len()

Die Funktion len() gibt die Länge einer Liste oder Zeichenkette als `int`-Wert an. Bei einer Liste wird die Anzahl an Elementen gezählt, bei einer Zeichenkette die einzelnen Textzeichen, aus denen die Zeichenkette besteht.

Beispiel:

```
len('Hallo Welt!')
# Ergebnis: 11

len( str(1000) )
# Ergebnis: 4

len( [1,2,3,4,5] )
# Ergebnis: 5
```

list()

Die Funktion list() erzeugt eine neue Instanz eines *list*-Objekts, also eine (veränderliche) Liste. Formal ist `l = list()` somit identisch mit `l = [ ]`.

Wird beim Aufruf von `list()` eine Sequenz angegeben, so wird die Liste mit den in der Sequenz vorkommenden Einträgen gefüllt.

Beispiel:

```
# Leere Liste erzeugen:
l1 = list()

# Liste mit Zahlen 0 bis 9 erzeugen:
l2 = list( range(10) )

l2
# Ergebnis: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

locals()

Die Funktion locals() liefert als Ergebnis ein `dict` mit den Namen und den Werten aller zum Zeitpunkt des Aufrufs existierenden lokalen, das heißt im aktuellen Codeblock sichtbaren Variablen.

map()

Die Funktion map(function, object) wendet eine Funktion auf alle Elemente eines iterierbaren Objekts (beispielsweise einer Liste) an. Als Ergebnis liefert `map()` ein neues iterierbares Objekt, dessen Elemente den einzelnen Ergebniswerten entsprechen.

Beispiel:

```
my_list = [3, 5, -10.2, -7, 4.5]
map(abs, my_list)
# Ergebnis: [3, 5, 10.2, 7, 4.5]
```

Oftmals wird anstelle der `map()`-Funktion eine (meist besser lesbare) *List-Comprehension* genutzt. Im obigen Beispiel könnte auch `[abs(x) for x in my_list]` geschrieben werden.

max()

Die Funktion max() gibt das größte Element einer Liste als Ergebnis zurück.

Beispiel:

```
max( [5,1,3,9,7] )
# Ergebnis: 9
```

min()

Die Funktion min() gibt das kleinste Element einer Liste als Ergebnis zurück.

Beispiel:

```
min( [5,1,3,9,7] )
# Ergebnis: 1
```

next()

Die Funktion `next(iterator)` bewirkt einen Aufruf von `iterator.__next__()`, gibt also das nächste Element der Iterator-Sequenz aus. Ist der Iterator am Ende der Sequenz angelangt, so wird von `next(iterator)` ein `StopIteration`-Error ausgegeben.

Beispiel:

```
# Iterator generieren:
iterator = iter( ['Hallo', 'Welt'] )

next(iterator)
# Ergebnis: 'Hallo'

next(iterator)
# Ergebnis: 'Welt'

next(iterator)
# Ergebnis:
# --> 1 next(iterator)
# StopIteration
```

object()

Die Funktion object() erzeugt eine Instanz eines neuen `object`-Objekts. Ein `objekt` ist die Basisklasse aller Objekte, hat allerdings keine besonderen Attribute oder Methoden.

Beim Aufruf von `object()` dürfen keine weiteren Argumente angegeben werden; zudem verfügt ein `object`-Objekt über kein `__dict__`, so dass der Instanz keine weiteren Attribute oder Methoden hinzugefügt werden können.

oct()

Die Funktion oct(int-wert) gibt eine Zeichenkette mit der Oktaldarstellung einer `int`-Zahl als Ergebnis zurück. Eine solche Zeichenkette wird mit `0o` eingeleitet, gefolgt von der eigentlichen Oktalzahl.

Beispiel:

```
oct(42)
# Ergebnis: '0o52'
```

open()

Die Funktion open(dateiname) gibt ein zum angegebenen Pfad passendes Datei-Objekt als Ergebnis zurück, das zum Lesen oder Schreiben von Dateien verwendet wird.

Die Funktion `open()` ist im Abschnitt *Dateien* näher beschrieben.

ord()

Die Funktion ord(zeichen) gibt die Unicode-Zahl (ein `int`-Wert) eines angegebenen Zeichens (Buchstabe, Zahl, oder Sonderzeichen) aus.

Beispiel:

```
ord('A')
# Ergebnis: 65

ord('a')
# Ergebnis: 97
```

Für viele Programme reichen die *ASCII-Codes* als Teilmenge des Unicode-Zeichensatzes bereits aus.

pow()

Die Funktion pow(zahl1, zahl2) gibt beim Aufruf von `pow(x,y)` den Wert von `x ** y`, also `x` hoch `y` aus (Potenz).

Beispiel:

```
pow(10, 3)
# Ergebnis: 1000

pow(10, -3)
# Ergebnis: 0.001
```

print()

Die Funktion print(zeichenkette) gibt die angegebene Zeichenkette auf dem Bildschirm aus; dabei können unter anderem mittels einer geeigneten *Formatierung* auch Werte von Variablen ausgegeben werden.

Beispiel:

```
print("Die Antwort lautet %d.", % 42)
# Ergebnis: Die Antwort lautet 42.
```

property()

Die Funktion property() wird verwendet, um auf ein Attribut einer Klasse nicht direkt, sondern mittels einer Methode zuzugreifen. Hierzu wird in der Klasse des Objekts je eine *Setter- und Getter*-Methode definiert, die zum Zuweisen und Abrufen des Attributs verwendet werden. Anschließend kann mittels `my_attribute = property(fget=getterfunction, fset=setterfunction)` ein Property-Attribut erzeugt werden.

„Klassisch“ kann die `property()`-Funktion folgendermaßen verwendet werden:

```
# Testklasse definieren:
class C():

    # Normales Klassen-Attribut anlegen:
    foo = 1

    # Getter-Funktion für 'bar' definieren:
    def get_bar(self):
        return self.foo

    # Setter-Funktion für 'bar' definieren:
    def set_bar(self, value):
        self.foo = value

    # 'bar' zu einer Property machen:
    bar = property(get_bar, set_bar)
```

Häufiger wird die `property()`-Funktion allerdings als *Funktionsdekorator* genutzt. Die Bedeutung bleibt dabei gleich, doch ist die Schreibweise etwas „übersichtlicher“:

```
# Testklasse definieren:
class C():

    # Normales Klassen-Attribut anlegen:
    foo = 1

    # Property 'bar' definieren:
    @property
    def bar(self):
        return self.foo

    # Setter für 'bar' definieren:
    @bar.setter
    def bar(self, value):
        self.foo = value
```

Erzeugt man mittels `c = C()` ein neues Objekt der obigen Beispielklasse, so kann auch mittels `c.bar` auf das Attribut `c.foo` zugegriffen werden:

```
# Instanz der Beispiel-Klasse erzeugen:
c = C()

c.bar
# Ergebnis: 1

# Wert der Property 'bar' ändern:
c.bar = 2

c.foo
# Ergebnis: 2
```

Üblicherweise erhält die Zielvariable, die von der Property verändert wird, den gleichen Namen wie die Property selbst, jedoch mit einem `_` zu Beginn des Variablennamens. Hierdurch wird ausgedrückt, dass die Variable nicht direkt verändert werden sollte (obgleich dies möglich wäre). In der Setter-Funktion kann dann beispielsweise explizit geprüft werden, ob eine vorgenommene Wertzuweisung überhaupt zulässig ist.

range()

Die Funktion range() erzeugt eine Sequenz ganzzahliger Werte. Sie kann wahlweise in folgenden Formen benutzt werden:

```
range(stop)
range(start, stop)
range(start, stop, step)
```

Wird der `range()`-Funktion nur ein einziger Wert n als Argument übergeben, so wird eine Zahlensequenz von 0 bis $n-1$ generiert; Werden zwei Werte m und n übergeben, so wird eine Zahlensequenz von m bis $n-1$ erzeugt. Allgemein ist bei Verwendung von `range()` die untere Schranke im Zahlenbereich enthalten, die obere hingegen nicht.

Wird eine dritte Zahl $i \neq 0$ als Argument angegeben, so wird nur jede i-te Zahl im angegebenen Zahlenbereich in die Sequenz aufgenommen. Ist der Startwert des Zahlenbereichs größer als der Stopwert und i negativ, so wird eine absteigende Zahlensequenz generiert.

Beispiel:

```
range(10)
# Ergebnis: range(0,10)

list( range(0, 10) )
# Ergebnis: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

list( range(0, 10, 2) )
# Ergebnis: [0, 2, 4, 6, 8]
```

(continues on next page)

(Fortsetzung der vorherigen Seite)

```
list( range(10, 0, -1) )
# Ergebnis: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

repr()

Die Funktion repr(objekt) gibt als Ergebnis eine Zeichenkette zurück, die eine kurze charakteristische Beschreibung des Objekts beinhaltet; häufig entspricht dies einer Angabe der Objekt-Klasse, des Objekt-Namens und der Speicheradresse.

Beispiel:

```
repr(print)
# Ergebnis: '<built-in function print>'
```

Ist in der Klasse des angegebenen Objekts eine `__repr__()`-Methode definiert, so ist `repr(objekt)` identisch mit `objekt.__repr__()`.

Als Zeichensatz wird für die Ausgabe des Strings Unicode verwendet, so dass beispielsweise auch Umlaute im Ausgabe-String enthalten sein können.

reversed()

Die Funktion reversed(sequenz) kann auf eine iterierbares Objekt (beispielsweise ein Tupel oder eine Liste) angewendet werden; sie gibt einen Iterator mit den gleichen Elementen, aber in der umgekehrten Reihenfolge zurück.

Beispiel

```
liste = [1,5,2,3]

liste_rev = reversed(liste)

liste_rev
# Ergebnis: <builtins.list_reverseiterator at 0x7f38edce2278>

for i in liste_rev:
    print(i)

# Ergebnis:
# 3
# 2
# 5
# 1
```

round()

Die Funktion round() rundet eine `float`-Zahl auf die nächste `int`-Zahl auf beziehungsweise ab und gibt diese als Ergebnis zurück. Wird zusätzlich zur Zahl eine zweite Integer-Zahl als Argument angegeben, also `round(a, n)`, so wird die Zahl `a` auf `n` Stellen gerundet als Ergebnis zurück gegeben.

```
round(15.37)
#Ergebnis: 15

round(15.37, 1)
#Ergebnis: 15.4
```

set()

Die Funktion set() erzeugt ein neues *set*-Objekt, also eine Menge.

Wird optional beim Aufruf von `set()` eine Sequenz als Argument angegeben, so wird das Mengen-Objekt mit den Einträgen dieser Menge gefüllt (doppelte Einträge bleiben ausgeschlossen).

Beispiel:

```
# Leeres Set-Objekt erstellen:
s1 = set()

# Set-Objekt aus einer Liste erstellen:
s2 = set( [1, 3, 5, 7, 9, 9] )

s2
# Ergebnis: set({1, 3, 5, 7, 9})

# Set-Objekt aus einer Zeichenkette erstellen:
s3 = set( "Hallo Welt!" )

s3
# Ergebnis: set({' ', '!', 'H', 'W', 'a', 'e', 'l', 'o', 't'})
```

setattr()

Die Funktion setattr(objekt, attributname, wert) weist dem angegebenen Attribut des als erstes Argument angegebenen Objekts den als drittes Argument angegebenen Wert zu (sofern dies möglich ist); formal ist `setattr(objekt, attributname, wert)` somit identisch mit `objekt.attributname = wert`.

Beispiel:

```
# Beispiel-Klasse:
class Point():

    x = 5
    y = 4

# Punkt-Objekt erzeugen:
p = Point()

# Attribut ändern:
setattr(p, 'x', 3)

# Attribut abrufen:
getattr(p, 'x')
# Ergebnis: 3

# Attribut neu zuweisen:
setattr(p, 'z', 2)

# Attribut abrufen:
getattr(p, 'z')
# Ergebnis: 2
```

slice()

Die Funktion slice(startwert, stopwert, stepwert) erstellt eine neue Instanz eines Slice-Objekts. Dieses Objekt repräsentiert einen Satz an Indizes, der durch die angegebenen Werte unveränderbar festgelegt ist.

Beispiel

```
s = slice(0,10,2)

s.start
# Ergebnis:0

s.stop
# Ergebnis:10

s.step
# Ergebnis:2
```

Beim Aufruf von `slice()` kann als Wert für die Argumente `start` und `stop` auch `None` angegeben werden. Das Slice-Objekt enthält dann nur `step` als unveränderlichen Wert. Wird das Slice-Objekt mit `s` bezeichnet, so kann in diesem Fall beispielsweise mittels `s.indices(100)` ein neues Slice-Objekt als Ergebnis geliefert werden, das den angegebenen Wert als `stop`-Wert hat.

Slice-Objekte werden selten direkt verwendet. Allerdings werden bei Datentypen wie *Zeichenketten* oder *Listen* Slicings gerne zur Auswahl von Elementen genutzt; ebenso kön-

nen bei Verwendung von Modulen wie *numpy* oder *pandas* Slicings eingesetzt werden, um mittels den dabei resultierenden Indizes Teilbereiche aus Zahlenlisten zu selektieren. Die Syntax lautet dabei etwa:

```
a = numpy.arange(10)

# Als Zahlenbereich die dritte bis zur siebten Zahl selektieren:

a[3:8]
# Ergebnis: array([3, 4, 5, 6, 7])

# Dabei nur jede zweite Zahl selektieren:

a[3:8:2]
# Ergebnis: array([3, 5, 7])
```

Verwendet man die gleichnamige Funktion `slice()` aus dem `itertools`-Modul, so wird als Ergebnis statt einem Slice ein entsprechendes Iterator-Objekt zurückgegeben.

sorted()

Die Funktion sorted(sequenz) kann auf eine iterierbares Objekt (beispielsweise ein Tupel oder eine Liste) angewendet werden; sie gibt eine Liste mit den entsprechenden Elementen in sortierter Reihenfolge zurück.

Beispiel

```
sorted([1,5,2,3])
# Ergebnis: [1, 2, 3, 5]
```

staticmethod()

Die Funktion staticmethod(methode) macht die angegebene Methode zu einer so genannten statischen Methode. Üblicherweise wird die `staticmethod()`-Funktion als *Funktionsdekorator* verwendet:

```
class C():

    @staticmethod
    def my_static_method(arguments):

        pass
```

Bei einer so definierten Methode wird weder der Name der Klasse noch der Name der Instanz angegeben, von der aus die Methode aufgerufen wird.

Die statische Methode des obigen Beispiels kann wahlweise mittels `C.my_class_method()` oder ausgehend von einer Instanz der Klasse, also mittels `C().my_class_method()` aufgerufen werden.

str()

Die Funktion str(objekt) gibt eine String-Version des als Argument angegebenen Objekts aus. Hierbei wird die Methode `objekt.__str__()` der jeweiligen Klasse aufgerufen.

Beispiel:

```
str( [1,2,3,4,5] )
# Ergebnis: '[1, 2, 3, 4, 5]'
```

sum()

Die Funktion sum(sequenz) gibt die Summe eines iterierbaren Objekts (beispielsweise einer Liste) als Ergebnis zurück.

Beispiel:

```
sum( [1,2,3,4,5] )
# Ergebnis: 15

sum( range(100) )
# Ergebnis: 4950
```

super()

Die Funktion super() gibt, ausgehend von der Klasse des aufrufenden Objekts, die in der Objekt-Hierarchie nächst höher liegende Klasse an; dies wird insbesondere bei der *Vererbung* von Methoden genutzt.

Die Objekt-Hierarchie gibt eine Art Stammbaum für die Klasse an. Über das Attribut `__mro__` einer Klasse („Method Resolution Order“) kann abgefragt werden, in welcher Reihenfolge Klassen bei einem Methodenaufruf nach einer entsprechend benannten Methode durchsucht werden.

Beispiel:

```
# Objekt-Hierarchie einer abgeleiteten Klasse anzeigen:

import enum

enum.OrderedDict.__mro__
# Ergebnis: (collections.OrderedDict, builtins.dict, builtins.object)
```

Wird beispielsweise beim Aufruf von `obj.eine_methode()` die Methode nicht im Namensraum des Objekts gefunden, so wird entlang der Method Resolution-Order geprüft, ob eine gleichnamige Methode in einer übergeordneten Klasse definiert ist. Ist dies der Fall, so wird die Methode dieser Klasse aufgerufen, da die konkrete Klasse des Objekts die Methoden „geerbt“ und nicht überschrieben hat. Den Zugriff auf die jeweils nächste Klasse der Method Resolution Order bietet gerade die Funktion `super()`.

Beim Programmieren kann die Funktion `super()`, die in Python3 fast immer ohne Argumente aufgerufen wird, genutzt werden, um eine Methode der übergeordneten Klasse aufzugreifen und gleichzeitig zu modifizieren.

tuple()

Die Funktion tuple(sequenz) erzeugt aus der angegebenen Sequenz (beispielsweise einer Liste oder einer Zeichenkette) eine neue Instanz eines *tuple*-Objekts, also eine unveränderliche Liste.

Beispiel:

```
tuple('Hallo Welt!')
# Ergebnis: ('H', 'a', 'l', 'l', 'o', ' ', 'W', 'e', 'l', 't')

tuple( range(10) )
# Ergebnis: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

type()

Die Funktion type(objekt) gibt als Ergebnis den Namen der Klasse des angegebenen Objekts zurück; dies ist identisch mit einem Aufruf von `objekt.__class__`.

Beispiel:

```
type("Hallo Welt!")
# Ergebnis: builtins.str
```

Eine zweite Verwendungsmöglichkeit der `type()`-Funktion liegt darin, sie als `type(objektname, basisklasse, attribute-dict)` aufzurufen, um ein neues Objekt zu erstellen. Dieses erbt alle Eigenschaften der angegebenen Basisklasse (oder mehrerer als Liste angegebener Basisklassen); zudem können für das Objekt in form eines `dict` weitere Attribute definiert werden.

Die folgenden beiden Code-Varianten erzeugen jeweils ein Objekt mit gleichen Eigenschaften:

```
# Beispielklasse definieren:
class C(object):
    x = 1

# Beispiel-Objekt generieren:
c1 = C()

# Type-Objekt mit gleichen Eigenschaften generieren:
c2 = type('C', (object,), dict(x=1) )
```

vars()

Die Funktion vars() gibt, sofern sie ohne Argument aufgerufen wird, als Ergebnis ein `dict` mit den Namen und den Werten aller zum Zeitpunkt des Aufrufs existierenden lokalen, das heißt im aktuellen Codeblock sichtbaren Variablen zurück (ebenso wie *locals()*).

Wird beim Aufruf von `vars()` als Argument ein beliebiges Objekt angegeben, so wird der Inhalt von `objekt.__dict__` als Ergebnis zurückgegeben.

zip()

Die Funktion zip() verbindet – ähnlich wie ein Reißverschluss – Elemente aus verschiedenen iterierbaren Objekten (beispielsweise Listen) zu einem neuen Iterator-Objekt, dessen Elemente Zusammensetzungen der ursprünglichen Elemente sind.

Beispiel:

```
zip( ['a', 'b', 'c'], [1, 2, 3, 4] )
# Ergebnis: <builtins.zip at 0x7f39027b22c8>

list( zip( ['a', 'b', 'c'], [1, 2, 3, 4] ) )
# Ergebnis: [('a', 1), ('b', 2), ('c', 3)]
```

Wichtige Standard-Module

Die im folgenden Abschnitt beschriebenen Module sind standardmäßig in Python enthalten, ohne dass zusätzliche Software-Pakete installiert werden müssen:

cmath – Mathe-Modul für komplexe Zahlen

Das cmath-Modul umfasst viele Funktionen des `math`-Moduls, die allerdings komplexe Zahlen als Argumente zulassen.

copy – Kopien von Objekten erstellen

Das copy-Modul stellt insbesondere die Funktion `deepcopy()` bereit, mit der 1 : 1-Kopien von existierenden Objekten gemacht werden können.

Erstellt man eine Kopie eines Objekts mittels `objekt2 = objekt1.copy()`, so wird genau genommen nur eine neue Referenz auf das bestehende Objekt angelegt. Hat `objekt1` beispielsweise ein Attribut `x` mit dem Wert `5`, so würde durch eine Eingabe von `objekt2.x = 7` auch der Attribut-Wert von `objekt1` geändert. Ein solches Verhalten ist beispielsweise bei der Übergabe von Objekten an Funktionen erwünscht, entspricht allerdings nicht der klassischen Vorstellung einer Kopie. Eine solche kann folgendermaßen erstellt werden:

```
import copy

# Kopie eines Objekts erzeugen:
objekt2 = copy.deepcopy(objekt1)
```

Werden nun die Attribut-Werte von `objekt2` geändert, so bleiben die Werte des Original-Objekts unverändert.

`cProfile` – Profiler

Mittels des Pakets cProfile und der darin definierten Funktion `run()` kann ermittelt werden, wie viel Zeit für einen Aufruf einer Funktion benötigt wird. Bei einer Funktion, die weitere Unterfunktionen aufruft, wird zudem angezeigt, wie viel Zeit auf die einzelnen Schritte entfällt:

```
import cProfile
cProfile.run('sum( range(10000000) )')

# Ergebnis:
# 4 function calls in 0.321 seconds

# Ordered by: standard name

# ncalls  tottime  percall  cumtime  percall filename:lineno(function)
#      1    0.000    0.000    0.321    0.321 <string>:1(<module>)
#      1    0.000    0.000    0.321    0.321 {built-in method exec}
#      1    0.321    0.321    0.321    0.321 {built-in method sum}
#      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.
↪Profiler' objects}
```

Mit dem Profiler können in verschachtelten Funktionen schnell „Bottlenecks“ gefunden werden, also Programmteile, die sehr rechenintensiv sind und daher bevorzugt optimiert werden sollten.

`functools` – Funktionen für aufrufbare Objekte

Das functools-Modul stellt einige Funktionen bereit, mit denen sich beispielsweise mathematische Funktion oder Lambda-Ausdrücke auf mehrere Elemente einer Liste anwenden lassen

- Die Funktion `functools.reduce()` führt die durch das erste Argument angegebene Funktion schrittweise von links nach rechts auf alle Elemente einer als zweites Argument übergebenen Sequenz aus; ein Aufruf von `ft.reduce(lambda x,y: x*y, [1,2,3,4,5])`. würde beispielsweise `((((1*2)*3)*4)*5)` berechnen.

`logging` – Logger-Modul

Das logging-Modul stellt einfache Funktionen bereit, mit denen ein einfaches Aufzeichnen verschiedener Informationen im Verlauf eines Programms ermöglicht wird.

Das `logging`-Modul ist im Abschnitt *Arbeiten mit Logdateien* näher beschrieben.

`math` – Mathematische Funktionen

Das math-Modul stellt eine Vielzahl häufig vorkommender mathematischer Funktionen bereit. Unter anderem sind folgende Funktionen nützlich:

- Mit `math.pi` und `math.e` können die Naturkonstanten $\pi = 3,1415\ldots$ und $e = 2,7182\ldots$ ausgegeben werden.
- Mit `math.floor(zahl)` wird der nächst kleinere `int`-Wert zur angegebenen Zahl ausgegeben, mit `math.ceil(zahl)` der nächst größere `int`-Wert.
- Mit `math.factorial(n)` wird die Fakultät $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$ einer positiven ganzzahligen Zahl n ausgegeben.
- Mit `math.sqrt(zahl)` wird die Wurzel einer positiven Zahl ausgegeben.
- Mit `math.pow(zahl, n)` wird die `n`-te Potenz der angegebenen Zahl ausgegeben. Für n kann auch eine `float`-Zahl kleiner als `1` angegeben werden; beispielsweise wird durch `math.pow(8, 1/3)` die dritte Wurzel von `8` berechnet.
- Mit `math.exp(x)` wird der Wert der Exponentialfunktion e^x ausgegeben.
- Mit `math.log(x, a)` wird der Wert der Logarithmusfunktion $\log_a(x)$ ausgegeben. Wird kein Basis-Wert angegeben, so wird der natürliche Logarithmus, also der Logarithmus zur Basis e berechnet.
- Mit `math.radians(winkelwert)` kann der angegebene Winkel (im Gradmaß) ins Bogenmaß, mittels `math.degrees(zahl)` das angegebene Bogenmaß ins Gradmaß Winkelwert umgerechnet werden.
- Mit `math.sin(x)`, `math.cos(x)` und `math.tan(x)` können die trigonometrischen Funktionen Sinus, Cosinus und Tangens zu den angegeben Werten berechnet werden; diese müssen im Bogenmaß, also in `rad` angegeben werden.
- Mit `math.asin(x)`, `math.acos(x)` und `math.atan(x)` können die Umkehrfunktionen zu den jeweiligen trigonometrischen Funktionen berechnet werden; die Ergebnisse werden im Bogenmaß, also in `rad` angegeben.

Das `math`-Modul ist für die Berechnung einzelner Werte vorgesehen. Für die Berechnung von Zahlenreihen stellt das Zusatz-Modul *numpy* gleichnamige Funktionen bereit. Diese können jeweils nicht nur einen einzelnen Wert, sondern jeweils auch eine Liste von entsprechenden Zahlenwerten als Argument effizient auswerten.

os – Interaktion mit dem Betriebsystem

Das os-Modul stellt einige nützliche Funktionen und Konstanten bereit, um gewöhnliche Aufgaben auf der Ebene des Betriebsystems durchführen zu können.

- Mit `os.chdir(pfad)` wird das als Argument angegebene Verzeichnis zum aktuellen Arbeitsverzeichnis.
- Mit `os.getcwd()` wird der Pfad des aktuellen Arbeitsverzeichnisses ausgegeben.
- Mit `os.listdir(pfad)` wird eine Liste aller Dateinamen des als Argument angegebenen Verzeichnisses ausgegeben.
- Mit `os.mkdir(verzeichnisname)` wird das als Argument angegebene Verzeichnis neu erstellt.
- Mit `os.rmdir(verzeichnisname)` wird das als Argument angegebene Verzeichnis gelöscht.
- Mit `os.remove(dateiname)` wird die als Argument angegebene Datei gelöscht.
- Mit `os.rename(alt, neu)` wird einer Datei oder einem Verzeichnis ein neuer Name zugewiesen.

Mit der Funktion `os.popen()` ist es zudem möglich, ein Programm in einer gewöhnlichen Shell aufzurufen. Hierzu wird der Funktion `os.popen()` als Argument eine Zeichenkette angegeben, deren Inhalt an den Shell-Interpreter weitergereicht wird. Die Ausgabe des Programms wird in eine Pipe geschrieben, die wie ein *Datei*-Objekt wahlweise zeilenweise mittels `readline()` oder als Ganzes mittels `read()` ausgelesen werden kann:

```
import os

# Shell-Anweisung festlegen:
command = 'ls -l'

# Shell-Anweisung ausführen:
# (Der Rückgabewert ist ein Filepointer auf die Pipe)
fp = os.popen(command)

# Das Ergebnis der Shellanweisung (Pipe) auslesen:
ergebnis = fp.read()

# Pipe schließen:
# (Status == None bedeutet fehlerfreies Schließen)
status = fp.close()
```

os.path – Pfadfunktionen

Das os.path-Modul stellt einige nützliche Funktionen bereit, die bei der Arbeit mit Datei- und Verzeichnisnamen hilfreich sind:

- Mit `os.path.exists(pfad)` kann geprüft werden, ob der als Argument angegebene Dateiname als Pfad im Dateisystem existiert; als Ergebnis gibt die Funktion `True` oder `False` zurück.
- Mit `os.path.isdir(pfad)` kann geprüft werden, ob der als Argument angegebene Pfad ein Verzeichnis ist; als Ergebnis gibt die Funktion `True` oder `False` zurück.
- Mit `os.path.isfile(pfad)` kann geprüft werden, ob der als Argument angegebene Pfad eine Datei ist; als Ergebnis gibt die Funktion `True` oder `False` zurück.
- Mit `os.path.getsize(pfad)` kann der vom als Argument angegebenen Pfad belegte Speicherplatz ausgegeben werden.

Um nicht nur relative, sondern auch absolute Pfadangaben nutzen zu können, kann die Funktion `os.path.abspath(pfad)` genutzt werden; diese gibt zu einem angegebenen (relativen) Dateinamen den zugehörigen absoluten Pfad an.

`pickle` – Speichern von Python-Objekten

Das pickle-Modul ermöglicht es, während einer Python-Sitzung existierende Objekte in Byte-Strings umzuwandeln und diese auf einer Festplatte zu speichern; ebenso können auf diese Art festgehaltene Daten mittels `pickle` zu einem späteren Zeitpunkt (sogar nach einem Neustart des Systems) auch wieder gelesen werden.

Um ein beliebiges Python-Objekt mittels `pickle` als Zeichenkette zu codieren, gibt man folgendes ein:

```
import pickle

liste_original = [1,2,3,4,5]

# Objekt als Byte-String ausgeben:
storage = pickle.dumps(liste_original)
b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05e.'
```

Hierbei steht `dumps` für „dump string“. Der erzeugte Byte-String ist zwar für Menschen nicht unmittelbar lesbar, kann aber vom Computer effizient geschrieben und auch mittels `pickle.loads()` („load string“) wieder ausgelesen werden:

```
# Byte-String zurückübersetzen:
liste_neu = pickle.loads(storage)

liste_neu
# Ergebnis: [1,2,3,4,5]
```

Das wieder geladene Objekt ist inhaltlich mit dem Original identisch, wird vom Interpreter jedoch als neues Objekt gehandhabt.

Soll das Ablegen eines Objekts unmittelbar in eine Datei erfolgen, so kann anstelle von `pickle.dumps()` die Funktion `pickle.dump()` verwendet und dabei als Argument ein existierender File-Pointer angegeben werden. Umgekehrt kann mittels `pickle.load()` wieder unmittelbar aus dem als Argument angegebenen Datei-Objekt gelesen werden.

`random` – Zufallsgenerator

Das random-Modul stellt Funktion zum Erzeugen von Zufallszahlen, für das Auswählen eines zufälligen Elements aus einer Liste sowie für das Umsortieren von Listen bereit.

Zu Beginn sollte zunächst stets eine neue Basis für die Erzeugung von Zufallszahlen in der aktuellen Python-Sitzung erstellt werden:

```
import random

# Zufallszahlen initiieren:
random.seed()
```

Anschließend können folgende Funktionen genutzt werden:

- Die Funktion `random.random()` liefert als Ergebnis eine Zufallszahl zwischen `0.0` und `1.0` (einschließlich dieser beiden Werte).
- Die Funktion `random.randint(min,max)` liefert als Ergebnis eine ganzzahlige Zufallszahl zwischen `min` und `max` (einschließlich dieser beiden Werte).
- Die Funktion `random.choice(sequenz)` wählt ein zufälliges Element aus einer Sequenz (beispielsweise einer Liste oder einem Tupel) aus.
- Die Funktion `random.shuffle(liste)` ordnet die Elemente einer Liste auf zufällige Weise neu an; dabei wird das Original verändert.

Beispielsweise kann also mittels `random.randint(1,6)` das Verhalten eines gewöhnlichen sechsflächigen Würfels imitiert werden.

`sys` – Systemzugriff

Das sys-Modul stellt Variablen und Funktion bereit, die in unmittelbarem Zusammenhang mit dem Python-Interpreter selbst stehen. Hilfreich sind unter anderem:

- Mit `sys.exit()` kann die aktuelle Interpreter-Sitzung beziehungsweise das aktuelle Programm beendet werden. Bei einem gewöhnlichen Beenden ohne Fehler wird dabei üblicherweise der Wert `0` als Argument angegeben, bei einem fehlerhaften Beenden der Wert `1`.
- Mit `sys.modules` erhält man eine Liste aller Module, die in der aktuellen Interpreter-Sitzung beziehungsweise im laufenden Programm bereits geladen wurden.
- Mit `sys.path` erhält man eine Liste mit Pfadnamen, in denen beim Aufruf von `import` nach Modulen gesucht wird.
- Mit `sys.stdin`, `sys.stdout` und `sys.stderr` hat man Zugriff zu den drei gewöhnlichen Shell-Kanälen (Eingabe, Ausgabe, Fehler). In Python werden diese wie gewöhnliche *Datei*-Objekte behandelt.
- Mit `sys.version` wird die Versionsnummer des Python-Interpreters ausgegeben.

`timeit` – Laufzeitanalyse

Mittels des Moduls timeit und der gleichnamigen Funktion aus diesem Paket kann einfach ermittelt werden, wieviel Zeit eine Funktion für einen Aufruf benötigt:

```
import timeit

timeit.timeit("x = 2 ** 2")
# Ergebnis: 0.027617342071607 71
```

ASCII-Codes

Dez	ASCII	Dez	ASCII	Dez	ASCII	Dez	ASCII	Dez	ASCII	Dez	ASCII	Dez	ASCII	Dez	ASCII
0	NUL	16	DLE	32	SP	48	0	64	@	80	P	96	‘	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Links

Offizielle Seiten

- Python Projektseite (en.)
- Python Dokumentation (en.)

Bücher und Tutorials

- Das Python-Tutorial (de, Python3-Version)
- Python 2 und 3 Tutorial
- Learning with Python 3 (en.)
- Think Python (en.)
- Think Complexity (en.)
- Python Vortrags-Folien

Dokumentationen von hilfreichen Werkzeugen

- Sphinx: Dokumentation von Projekten (en.)
- Scipy: Wissenschaftliche Werkzeuge (en.)

Code-Suche

- Nullege Python Quellcode-Suchmaschine

Private Python-Seiten

- Mathematik mit Python

Tips und Tricks

- Python bei Stack Overflow
- Python Code-Beispiel-Suchmaschine

Stichwortverzeichnis

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z