Python Skript

Clelia Albrecht, Felix Boes, Johannes Holke

10. April 2017 — 13. April 2017

Für Mama und Papa

Für Mamma und Papa

Für Mama und Papa

Python ist ja ganz nett, aber ist C nicht besser und überhaupt was soll dieses Einrücken?

Guido von Rossum

Inhaltsverzeichnis

1. Einleitung **7**

2. Das Python Datenmodell **9**
2.1. Wiederholung: Daten in C und C++ . . . 9
2.2. Daten in Python . . . 9
2.2.1. Objekte . . . 9
2.2.2. Variablen . . . 10
2.3. Speichermanagement . . . 10

3. Crashkurs **11**
3.1. Kommentare . . . 11
3.2. Print . . . 11
3.3. Typ und Identität . . . 11
3.4. Zahlen . . . 11
3.5. Strings . . . 12
3.6. Bedingungen . . . 12
3.7. Listen . . . 13
3.8. Schleifen . . . 13
3.9. Funktionen . . . 15
3.10. Module . . . 15
3.11. Workflow . . . 17

4. Einige Standarddatentypen **18**
4.1. Zahlen . . . 18
4.2. bool . . . 19
4.3. Sequenzen . . . 19
4.3.1. Gemeinsamkeiten . . . 19
4.3.2. Oberflächeliche und tiefe Kopien . . . 20
4.3.3. tuple . . . 21
4.3.4. list . . . 22
4.3.5. range . . . 23
4.3.6. list comprehension . . . 23
4.3.7. str . . . 24
4.4. None . . . 24
4.5. Dateien . . . 25
4.6. Mengen . . . 26
4.6.1. Mengen erzeugen . . . 26
4.6.2. Operationen für set und frozenset . . . 26
4.6.3. Operationen für set . . . 27
4.7. Verzeichnisse . . . 28
4.7.1. Verzeichnisse erstellen . . . 28
4.7.2. Verzeichnisse nutzen . . . 28

5. Einschub: Namespacing 30

6. Funktionen 31
6.1. Funktionen definieren und aufrufen . . . 31
6.2. Wo ist die main-Funktion? . . . 34
6.3. Funktionen sind auch nur Objekte . . . 34

7. Klassen 36
7.1. Klassen definieren und nutzen . . . 36
7.2. Eine sehr naive Matrixklasse . . . 37
7.3. Spezielle Funktionen . . . 38
7.3.1. Duck-Typing . . . 38
7.3.2. Spezielle Memberfunktionen . . . 39
7.4. Eine naive Matrixklasse . . . 43
7.5. Hashbare Objekte . . . 43

8. Ausnahmen 46
8.1. Ausnahmen behandeln . . . 46
8.2. Ausnahmen auslösen und weitergeben . . . 49
8.3. Bereits definierte und eigens definierte Ausnahmen . . . 51

9. Module 53
9.1. Module einbinden . . . 53
9.2. Eigene Module in Python bereitstellen . . . 55
9.2.1. Skripte als Module einbinden . . . 55
9.2.2. Ordner als Module einbinden . . . 56
9.3. Empfohlene Module . . . 58
9.3.1. re . . . 58
9.3.2. copy . . . 58
9.3.3. math . . . 58
9.3.4. cmath . . . 59
9.3.5. random . . . 59
9.3.6. itertools . . . 59
9.3.7. pickle . . . 60
9.3.8. os . . . 61
9.3.9. time . . . 61
9.3.10. argparse . . . 62
9.3.11. multiprocessing . . . 63
9.3.12. subprocess . . . 64
9.3.13. sys . . . 64
9.3.14. Weitere Python Module . . . 64

10. Python + C = ♡ **66**
10.1. Extension Modules 66
10.1.1. Die Ausgangssituation 66
10.1.2. Schnelle C-Funtkion implementieren 66
10.1.3. Jede schnelle C-Funktion braucht eine Hilfsfunktion 67
10.1.4. Die Hilfsfunktion werden in einem Modul zusammengefasst 68
10.1.5. Das Modul kompileren und benutzen 70
10.2. CTypes 70

A. Installation **72**
A.1. Python3 installieren 72
A.2. PyCharm Community Edition installieren 72

Literatur **73**

1. Einleitung

Bei vielen Anwendungen in der wissenschaftlichen Programmierung sind übersichtlicher, handlicher Code und eine schnell mögliche Implementierung, für die man auf viele verschiedene, bereits vorhandene, gute Bibliotheken zugreifen kann, wichtiger als Schnelligkeit und optimierter Speicherverbrauch.

Wer sehr viel Wert auf die letzten beiden Eigenschaften legt, ist zum Beispiel mit **C/C++** als Programmiersprache der Wahl gut beraten (siehe auch die erste Hälfte dieses sehr guten Kurses [ABH17]). In den meisten Fällen bringt einen **Python** jedoch schneller ans Ziel.

Eine der Hauptmotivationen von **Python** ist es, eine besonders übersichtliche, gut lesbare und einfache Programmiersprache zu sein. Die Syntax ist deshalb sehr reduziert und **Python** kommt mit wenigen Schlüsselwörtern aus. Die überschaubare Standardbibliothek ist leicht zu erweitern – tatsächlich ist einer der größten Vorteile von **Python** die große Fülle an bereits existierenden Modulen und Bibliotheken, die einem als Nutzer sehr viel Zeit bei der Programmierung ersparen können (mehr dazu kann in **Abschnitt 9** nachgelesen werden).

Diese Erweiterbarkeit von **Python** sorgt auch dafür, dass Nachteile (wie zum Beispiel die im Vergleich zu maschienennäheren Sprachen eher langsame Performance) ausgeglichen werden können. Beispielsweise können performancekritische Routinen in **C** implementiert und in **Python** eingebunden werden um die Vorteile beider Sprachen verbinden zu können (zusammengefasst in **Tabelle 1**).

	Code schnell schreiben	Schnellen Code schreiben
Python	+	-
C/C++	-	+
Python + **C** = ♡	+	+

Tabelle 1: Die awesome Effizienzmatrix

Python ist keine kompilierte, sondern eine interpretierte Sprache. Der aus **C/C++** bekannte Ablauf Code schreiben – kompilieren – ausführen entfällt also in der Form. Stattdessen wird der Programmcode dem sogenannten **Interpreter** übergeben. Hierbei hat man zwei Möglichkeiten zur Auswahl: entweder man schreibt den Code direkt in den Interpreter, der die einzelnen Codeblöcke daraufhin sofort ausführt, oder man übergibt ihm den Code gebündelt in einer Datei (man hat zudem noch die Möglichkeit, diese Datei direkt ausführbar zu machen). Mehr zur Nutzung und Installation von **Python** findet sich in **Anhang A**.

Diese Vorlesung und das begleitende Skript haben zum Ziel, in die Programmiersprache **Python3** einzuführen. Dabei richten wir uns vor allem an Programmiererinnen, die bereits (fortgeschrittene) Erfahrungen in **C/C++** haben. Es ist jedoch auch möglich, die Vorlesung ohne weitreichende Programmierkenntnisse zu besuchen. Der Kurs richtet sich an Bachelorstudenten der Mathematik, die begleitenden Übungen sind daher meist mathematisch motiviert.

Aufgrund der knappen Zeit besprechen wir in diesem Skript nur einige Grundlagen von **Python**. Der interessierten Leserin legen wir den **Python**-Standard [Pytb] und [Pytc] aufgrund der hohen Präzision sehr ans Herz. Das Buchprojekt [Lot10], welches sich "nur" auf **Python2** bezieht, ist ebenso (vor allem für weniger erfahrene Programmiererinnen) zu empfehlen, da es ausführlich ist, viele gute Beispiele enthält und kaum Programmierkenntnisse voraussetzt.

Dieses Skript ist wie folgt aufgebaut:
In **Abschnitt 2** erklären wir den wesentlichen konzeptionellen Unterschied zwischen **Python** und **C/C++**, das **Python**-Datenmodel, bevor wir in **Abschnitt 3** die grundlegenden Bausteine von **Python** zusammenfassen. Dieser Crashkurs hat vor allem zum Ziel, Besonderheiten der Syntax aufzuzeigen um bereits erfahreneren Programmiererinnen den Einstieg in **Python** zu erleichtern. Grundlegende Konzepte wie Bedingungen, Schleifen, Ausgabe und Funktionen werden nicht konzeptionell wiederholt.

Die Datentypen, mit denen man in **Python** am häufigsten arbeitet, werden in **Abschnitt 4** behandelt. Diese umfassen unter anderem Zahlen, Sequenzen, Mengen, Verzeichnisse und den Umgang mit Dateien. Zu jedem Datentyp listen wir die wichtigsten Operationen und Funktionen auf.

Einige Erklärungen zum Namespace befinden sich in **Abschnitt 5**. Diese sind notwendig um den Umgang mit Variablen in den folgenden Abschnitten besser verstehen zu können.

Abschnitt 6 widmet sich der Syntax von Funktionen in **Python**. Neben Funktionsdefinitionen und -aufrufen gibt es noch ein paar Bemerkungen zur Erstellung einer `main`-Funktion in **Python**, sowie den Vorteilen die sich ergeben, wenn man Funktionen als Objekte auffasst.

Neben Funktionen sind Klassen eine weitere Möglichkeit, Code übersichtlicher zu gestalten und einfacher nutzbar zu machen. Genauere Erklärung zur Syntax, Aufbau und Nutzung von Klassen finden sich in **Abschnitt 7**.

Ein wichtiger Aspekt des Programmierens ist die Fehlerbehandlung. Diese wird in **Python** durch Ausnahmebehandlung vereinfacht, die in **Abschnitt 8** ausführlich besprochen wird.

Wie bereits in der Einleitung erwähnt ist einer der größten Vorteile von **Python** die riesige Vielfalt an bereits vorhandenen Modulen. Einen Überblick über der nützlichsten Module die der **Python** Standard für Mathematiker bereit hält sowie Anmerkungen zur Nutzung von Modulen finden sich in **Abschnitt 9**.

Zum Abschluss, in **Abschnitt 10** gehen wir noch auf die Möglichkeit ein, bereits vorhandenen **C/C**++-Code und sogar bereits kompilierte Bibliotheken mit **Python** zu verbinden und somit das beste aus beiden Welten zu vereinen.

In **Anhang A** finden sich zum Schluss noch einige Hinweise zur Installation von **Python** unter verschiedenen Betriebssystemen.

2. Das Python Datenmodell

Ein wesentlicher Unterschied zwischen **C/C++** und **Python** ist das Verhalten von Daten im Speicher und den Zugriff darauf. Um den Unterschied zu **C/C++** klarer zu machen, beginnen wir mit einer kurzen Wiederholung

2.1. Wiederholung: Daten in C und C++

In **C/C++** gibt es Variablen. Diese bestehen aus einer "Speicheradresse", einem "Typ" und dem dort gespeicherten "Wert". Sie verhalten sich also wie ein zusammenhängendes Stück Speicher, indem ein Wert gespeichert wird und wir wissen, wie wir diesen Wert zu interpretieren haben. Nehmen wir beispielsweise folgende Codezeile.

```
1 int32_t a = 0;
2 int32_t b = 3;
3 a = b;
```

In der ersten Zeile erstellen wir eine Variable vom Typ `int32_t`, d.h. wir haben nun ein zusammenhängendes Stück Speicher indem der Wert einer ganzen 32-bit-Zahl gespeichert werden kann, und legen dort den Wert der Expression `0` ab. In der zweiten Zeile erstellen wir eine weitere Variable vom Typ `int32_t` und legen dort den Wert der Expression `3` ab. In der dritten Zeile legen wir den Wert der Expression `b` in den zu `a` gehörigen Speicher ab. Nun ist der in `a` gespeicherte Wert `3` und der in `b` gespeicherte Wert ebenfalls `3`. Wenn `b` stattdessen vom Typ `double *` wäre, führte Zeile 3 zu einem Compilerfehler.

2.2. Daten in Python

Das **Python** Datenmodell ist intelektuell nicht so einfach zu fassen wir das Datenmodell von **C/C++**. Es ist zwar keine Magie, dennoch kann es einige Tage oder gar Wochen dauern kann, bis man sich daran gewöhnt hat.

In **Python** gibt es "Objekte" und "Variablen". Objekte repräsentieren Daten und Variablen referenzieren auf Objekte. Zunächst besprechen wir Objekte genauer und kümmer uns im Anschluss um Variablen.

2.2.1. Objekte

Objekte bestehen aus einer "Identität", einem "Typ" und einem "Wert". Sie verhalten sich wie ein zusammenhängendes Stück Speicher, in dem ein Wert gespeichert wird, also ganz genauso wie sich Variablen in **C/C++** verhalten. Die Identität eines (existierenden) Objekts ändert sich nie und bestimmt ein Objekt (während der Laufzeit) eindeutig. Wenn man mag, kann man sich die Identität wie die Speicheradresse des Objekts vorstellen (auch wenn man in **Python** keine Pointer verwendet).

Ein jedes Objekt ist entweder "mutable" und kann nach seiner Erzeugung verändert werden oder es ist "immutable" und ist nach seiner Erzeugung konstant. Dieses Konzept gibt es ebenfalls in **C/C++**: Alle Typen in **C/C++** sind ohne weiteres "mutable",

es sei denn, sie bekommen das Präfix `const` bei ihrer Definition, denn dann sind sie “immutable”.

In **Python** sind folgende Objekte immutable: die ganze Zahl `3` (siehe Abschnitt 4.1), das Tupel `(1, 'a')` (siehe Abschnitt 4.3) und der leere Typ `None` (siehe Abschnitt 4.4). Die folgenden Objekte sind mutable: Die Liste `[1, 'a']` (siehe Abschnitt 4.3), das Verzeichnis `{ 'Felix': 405, 'Jonathan': 599 }` (siehe Abschnitt 4.7) und die meisten Klassen (siehe Abschnitt 7).

2.2.2. Variablen

In **Python** haben Variablen keinen Typ. Wir wiederholen nochmal: In **Python** haben Variablen keinen Typ. Und ja, in der Tat, wir wiederholen nochmal: In **Python** haben Variablen keinen Typ. Variablen referenzieren auf Objekte; sie sind sozusagen die (temporären) Namen von Objekten. Wenn man mag, kann man sich Variablen wie (`void`)-Pointer vorstellen (auch wenn man in **Python** keine Pointer verwendet).

Variablen beziehen sich immer auf ein Objekt. Beim Erstellen einer Variable muss ihr konsequenterweise direkt ein Objekt zugewiesen werden. Betrachten wir ein einfaches Beispiel.

```
1 a = 'Hallo' # Erstelle a; a referenziert das Objekt 'Hallo'.
2 b = 'Welt'  # Erstelle b; b referenziert das Objekt 'Welt'.
3 a = b       # a bezieht sich nun auf das Objekt "Welt".
4 b = 3.141   # b bezieht sich nun auf das Objekt 3.141
```

In der ersten Zeile erstellen wir eine Variable `a`, die sich auf das Objekt `'Hallo'` bezieht. Das Objekt `'Hallo'` hat beispielsweise die Identität `1000`. In der zweiten Zeile erstellen wir eine Variable `b` die sich auf das Objekt `'Welt'` bezieht. Das Objekt `'Welt'` hat beispielsweise die Identität `2000`. In der dritten Zeile weisen wir `a` die Referenz auf das Objekt zu, auf das sich `b` bezieht. In der vierten Zeile weisen wir `b` die Referenz auf das Objekt `3.141` zu. Das Objekt `3.141` hat beispielsweise die Identität `3000`. Danach bezieht sich `a` auf `'Welt'` und `b` auf `3.141`.

2.3. Speichermanagement

Die meisten Objekte können genau solange verwendet werden, wie es Variablen gibt, die auf sie referenzieren. (Einige Ausnahmen sind beispielsweise Zahlen und das Objekt `None`.) Objekte, die nicht mehr verwendet werden können, werden automatisch dem “Garbage Collector” übergeben. Dieser entfernt die Objekte nach eigenem Bedarf aus dem Speicher. Das ist in den meisten Fällen bequemer als in **C/C++**, wo man seinen Speicher selbst verwalten muss. In der Zeit wo der **C/C++** Programmierer seinen Speicher aufräumt, trinken wir lieber GLUMP[1].

[1] GLUMP Alkoholcola vereint das erweckende Prickeln von koffeinhaltiger Cola mit der betäubenden Wirkung von Alkohol.

3. Crashkurs

Wir beginnen mit einem Überblick der grundlegende Bausteine. In den nachfolgenden Kapiteln besprechen wir einige der Bausteine nochmal im Detail.

3.1. Kommentare

Kommentare in **Python** beginnen mit # und enden am Ende der Zeile. Sie verhalten sich also ganz genau wie Kommentare in **C/C++**, die mit // beginnen[2]. Gut platzierte Kommentare dürfen in keinem Programm fehlen. Ein schlecht kommentiertes Programm ist ein Anzeichen dafür, dass es von einem schlechten Programmierer geschrieben wurde.

3.2. Print

So gut wie alle Objekte können durch die **Python** Funktion `print` auf der Standardausgabe ausgegeben werden. Das liegt daran, dass die meisten Objekte Klassen sind und die Funktion `__str__` implementieren (auch wenn wir das jetzt noch nicht verstehen; siehe Abschnitt 7.3). Die Funktion `print` ist für **Python2** und **Python3** verschieden.

```
1 print 'Hallo', 'Welt'    # Python2
2 print ('Hallo', 'Welt') # Python3
```

In diesem Skript nutzen wir die von **Python3** bereitgestellte Version von `print`. Will man in **Python2** die `print`-Funktion aus **Python3** verwenden, kann man folgende Zeile am Anfang seines Skript einfügen.

```
1 from __future__ import print_function # Nutze Python3-print in Python2
```

3.3. Typ und Identität

Um die Identiät eines Objekts `x` zu bestimmen, nutzen wir die Funktion `id(x)`. Sie gibt einen String zurück, den wir mit `print` ausdrucken können. Ganz ähnlich bekommen wir den Typ eines Objekts mit `type(x)`.

3.4. Zahlen

In **Python** gibt es die Zahlentypen `int`, `long`, `float` und `complex`. Sie verhalten sich (fast) genauso wie die aus **C/C++** bekannten Typen `int`, `long`, `float` und `complex`[3]. Man erstellt ein Objekt vom Typ `int`, durch Angabe einer ganzen Zahl oder durch Aufrufen der Funktion `int`. Ähnlich erstellt man ein Objekt vom Typ `float`, durch Angabe einer Kommazahl oder durch Aufrufen der Funktion `float`.

```
1 print( 2 )          # 2    typ: int
2 print( int(2.1) )   # 2    typ: int
3 print( 2.0 )        # 2.0  typ: float
4 print( float(2) )   # 2.0  typ: float
```

[2] Die Kommentare // wurden in **C99** (siehe [ISO99]) eingeführt.
[3] Der Typ `complex` wurde in **C99** (siehe [ISO99]) eingeführt.

Man kann zwei verschiedene oder gleiche Zahlentypen mit mathematischen Operatoren verbinden. Das Ergebnis hat den “genaueren” Typ. Des Weiteren beschreibt `//` die ganzzahlige Division ohne Rest.

```
1 print( 2 + 3.3 )   # 5.3   typ: int +  float -> float
2 print( 5 // 2 )    # 2     typ: int // int   -> int
3 print( 5.3 // 2 )  # 2.0   typ: int // float -> float
```

Die Division von Ganzzahlen unterscheidet sich in **Python2** und **Python3**.

```
1 4 /  2 # Python2: 2 vom Typ int;   Python3: 2.0 vom Typ float;
2 4 // 2 # Python2: 2 vom Typ int;   Python3: 2   vom Typ int;
```

3.5. Strings

Strings die eine Zeile lang sind, werden entweder mit `'` oder mit `"` begonnen und beendet. Soll ein String mehre Zeilen lang sein, kann man ihn mit `"""` beginnen und beenden. Genau wie in **C/C++** gibt es gewisse Zeichen, die nicht direkt geschrieben werden können und mit `\` beginnen müssen (zum Beispiel werden neue Zeilen durch `\n` ausgedrückt). Folgender Code erstellt den String “Hallo[Neue Zeile]Welt”.

```
1 print( 'Hallo\nWelt' ) # Hallo[neue Zeile]Welt
```

Es gibt eine große Auswahl an Funktionen, um aus einem gegebenen String einen neuen zu erstellen. Beispielsweise verknüpft man zwei Strings mit `+` und die Funktion `lower` ersetzt alle Großbuchstaben durch Kleinbuchstaben.

```
1 print( 'Hallo' + '␣' + 'Welt' ) # Hallo Welt
2 print( 'Hallo␣Welt'.lower() )   # hallo welt
```

3.6. Bedingungen

In **Python** gibt es genau zwei Objekte `True` und `False` von Typ `bool`. (Fast) alle Typen können mit der Funktion `bool` in einen der beiden Objekte umgewandelt werden. Den Kontrollfluss des Programms steuert man im einfachsten Fall wir folgt.

```
1 if bedingung:
2   ausdruck
3   ...
```

Das heißt, wir brauchen zunächst eine Expression `bedingung`, die als `True` oder `False` interpretiert werden kann. Wertet sie als `True` aus, so wird ein Block von Ausdrücken ausgeführt. Anders als in **C/C++** werden in **Python** Blöcke nicht durch Klammern, sondern durch konsistente Einrückung definiert.

```
1 x = 101//3+46//5
2 if x == 42:
3   print('Die␣Antwort')
```

Da gut eingerückter Code (also Code, in dem jeder Block konsistent eingerückt ist) deutlich lesbarer sind als nicht eingerückter Code, ist er in jedem Fall – unabhängig von der Sprache – ein Zeichen für eine gute Programmiererin.

Da es jedoch trotzdem immer noch genug Leute gibt, die hässlichen Code produzieren, wird man in **Python** dazu gezwungen, auf konsistente Einrückung zu achten: zu einem Block gehören genau die nachfolgenden Zeilen, die genauso weit eingerückt sind wie die erste Zeile des Blocks. Bei nicht konsistenter Einrückung gibt es Syntaxfehler. Dies macht es sehr schwer, hässlichen Code in **Python** zu schreiben.

Aus eigener Erfahrung können wir sagen, dass man sich als erfahrene **C/C++** Programmiererin recht schnell an diese Konvention gewöhnen kann, wenn man auch schon vorher auf sauber geschriebene Programme geachtet hat.

Optional besteht eine Bedingung aus keinem oder mehreren `elif bedingung:`[4] und keinem oder einem `else`: Die Konstrukte `else:` und `elif bedingung:` in **Python** verhalten sich wie die Konstrukte `else` und `else if` in **C/C++**.

```
1 x = 101//3+46//5                 # x = ?
2 if x == 42:                      # Falls x den Wert 42 hat:
3   print('Die Antwort')           # Drucke 'Die Antwort'
4 elif x == 84:                    # Sonst, falls x den Wert 84 hat:
5   print('Zweimal die Antwort')   # Drucke 'Zweimal die Antwort'
6 elif x == 106:                   # Sonst, falls x den Wert 106 hat:
7   print('Dreimal die Antwort')   # Drucke 'Dreimal die Antwort'
8 else:                            # In allen anderen Faellen:
9   print('Versuchs nochmal')      # Drucke 'Versuchs nochmal'
```

3.7. Listen

In **Python** gibt es verschiedene "Containertypen" und hier besprechen wir den Typ `list`. Eine Liste `thor` ist eine linear geordnete Menge von Variablen, auf die man mit `thor[0], thor[1], ..., thor[i]` zugreift. Um auf Elemente von hinten zuzugreifen, sind negative Werte für `i` erlaubt.

```
1 thor = [2, 'Hallo', 33.3, 'Welt', [] ] # Erzeugt eine Liste
2 print( thor[1], thor[-1] )             # Gibt folgendes aus: Hallo []
```

Die Länge einer Liste ist `len(thor)`. Listen sind mutable, das heißt wir können sie verändern. Mit `thor.append(x)` erweitert man die Liste `thor` um die Variable `x` und mit `thor.remove(x)` entfernt man die erste Variable mit dem Wert `x`. Ob ein Objekt mit Wert `x` in `thor` enthalten ist, testet man mit `x in thor`. Mit `thor+asgard` verknüpft man die beiden Listen `thor` und `asgard`.

3.8. Schleifen

In **Python** gibt es zwei Arten von Schleifen. Die `while`-Schleife führt einen Codeblock aus, solange die vor der Ausführung des Blocks geprüfte Bedingung als `True` auswertet.

[4] sprich "else if"

```
while bedingung:
  ausdruck
  ...
```

Da uns dieses Konzept aus **C/C++** sehr vertraut ist, brauchen wir hier kein Beispiel.

Die `for`-Schleife führt einen Codeblock für jedes Element einer gegebenen Sequenz (siehe Abschnitt 4.3) aus. Eine Sequenz kann zum Beispiel eine Liste oder ein String sein.

```
for i in s: # der Ausdruck s muss als Sequenz interpretiert werden koennen
  ausdruck
  ...
```

Dabei referenziert die Laufvariable `i` jedes Element der Sequenz `s` nacheinander einmal. Durch Verwendung von `i` innerhalb des folgenden Codeblocks kann dann auf den Wert des jeweiligen Objekts zugegriffen werden. Nach Ausführung des Codeblocks der `for`-Schleife referenziert `i` dann das nächste Element von `s`, bis das Ende der Sequenz erreicht ist.

Dies wird durch das folgende Beispiel veranschaulicht:

```
s = 'Hallo Welt' # Die Sequenz 'Hallo Welt'
for i in s:      # Nimm alle i = H,a,l,l,o, ,W,e,l,t
  print(i)       # Drucke i aus
```

Hier ist unsere Sequenz `s` ein String mit dem Wert `Hallo Welt`. In der `for`-Schleife nimmt die Laufvariable nacheinander die Werte der einzelnen Zeichen von `s`, also `H,a,l, l,o, ,W,e,l` und `t`. Im zur Schleife gehörenden Block werden diese Zeichen dann einzeln ausgegeben.

Wenn der Codeblock für eine aufsteigende Folge von Zahlen `from, from+1, ..., to-1` ausgeführt werden soll, nimmt man die Sequenz `range(from, to)`. Wichtig ist, dass wir das halboffene Interval betrachten, also nur bis `to-1` und nicht bis `to` gehen. Damit verhält sich der **Python**-Code `for i in range(from, to):` ganz genau wie der **C/C++**-Code `for(i = from; i < to; ++i) { ... }`. Hier ein Beispiel

```
sum = 0                # Setze sum auf 0
for i in range(1,5):   # Nimm alle i = 1, 2, 3, 4
  sum += 3*i           # Addiere 3*i zu sum
print( sum )           # Drucke sum aus
```

Will man eine absteigende Folge ganzer Zahlen, oder allgemeiner eine Folge ganzer Zahlen mit Schrittweite `step > 0` oder `step < 0` haben, nimmt man die Sequenz `range(from,to,step)`. Also können wir obiges Beispiel wie folgt umformulieren.

```
sum = 0                    # Setze sum auf 0
for i in range(3,15, 3):   # Nimm alle i = 3, 6, 9, 12
  sum += i                 # Addiere i zu sum
print( sum )               # Drucke sum aus
```

3.9. Funktionen

Meistens will man sich wiederholende Codeblöcke auslagern und natürlich kann man auch in **Python** Funktionen definieren. Jede Funktionsdefinition erstellt ein Objekt, welches der Funktion im Speicher entspricht. Außerdem wird eine Variable definiert, welche auf die Funktion im Speicher referenziert.

```
1 def funktionsname(parameter):
2   ausdruck
3   ...
```

Die Variable ist hier `funktionsname` und zeigt auf das Objekt, welches der definierten Funktion entspricht.

Anders als in **C**/**C**++ hat eine Funktion keine Signatur und keinen definierbaren Rückgabetyp. Mit `return` können kein, ein oder mehrere Objekte zurückgegeben werden. Wird kein Objekt zurückgegeben, so ist die Rückgabe automatisch `None`. Wird ein Objekt `x` zurückgegeben, so ist der Rückgabetyp automatisch `type(x)`. Werden mehrere Objekte zurückgegeben, so werden diese automatisch in einem `tuple` zusammengefasst, der Rückgabetyp ist also `tuple`. Ein Tuple ist eine Liste, die nicht geändert werden kann (siehe Abschnitt 4.3).

```
1  def f(i):
2    if i == 1:
3      return 1            # Rueckgabetyp: int,   Rueckgabewert: 1
4    elif i == 2:
5      return 'Hallo', 44 # Rueckgabetyp: tuple, Rueckgabewert: ('Hallo', 44)
6
7  print( type( f ) )                      # <class 'function'>
8  print( type( f(1) ) )                   # <class 'int'>
9  print( type( f(2) ) )                   # <class 'tuple'>
10 print( type( f('Gurkenwasser') ) ) # <class 'NoneType'>
```

Also ist das von `f(3)` oder auch `f('Gurkenwasser')` zurückgegebene Objekt `None`.

Beim Funktionsausruf übergibt man die Parameter entweder in der Reihenfolge wie sie in der Funktionsdefinition spezifiziert sind, oder man übergibt `parametername=ausdruck`. Außerdem kann man den Argumenten einer Funktion Standardwerte übergeben. Beim Funktionsausruf muss man die Argumente mit Standardwerten nicht angeben, kann es aber.

```
1 def plus(a,b=1):
2   return a+b
3
4 plus(4,5)        # = 9
5 plus(b=5, a=4) # = 9
6 plus(3)          # = 4
```

3.10. Module

Was **Python** wirklich mächtig macht, ist nicht die Sprache an sich, sondern die schier unendliche Fülle an Paketen die **Python**-Programmierer bereit stellen. Ein Modul ist so

etwas wie eine “shared library” in **C/C++**. Man kann sie importieren und dann nutzen, die Implementierung haben andere bereits übernommen (siehe Abbildung 1).

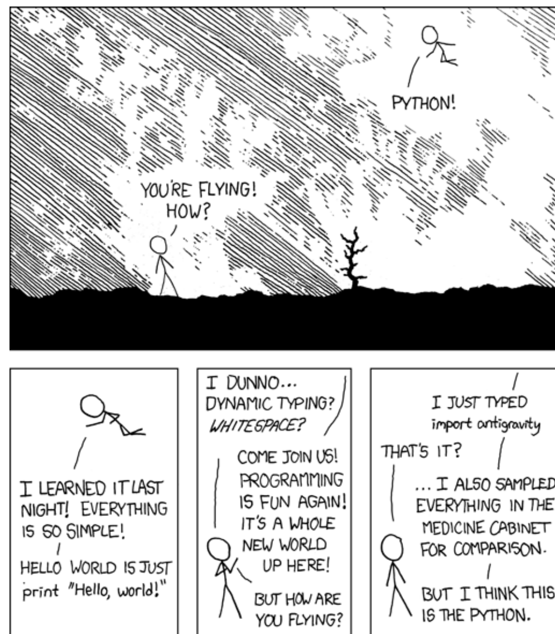

Abbildung 1: “Python” by Randall Munroe [Mun]

Ein Modul bindet man mit `import modulname` in sein Programm ein. Alle Objekttypen (Klassen, Funktionen und andere Objekte) die `modulname` bereitstellt, können durch `modulname.oname` erreicht werden.

```
1 import math
2 print( math.sin(math.pi/3.0) )
```

Wenn man aus `modulname` bestimmte Objekte einbinden möchte, nutzt man folgende Codezeile.

```
1 from modulname import oname1, oname2, ...
```

Jetzt kann man auf die Objekte direkt (also ohne das Präfix `modulname.`) zugreifen. Wir hatten bereits gesehen, wie man die **Python3** Printfunktion in **Python2** einbinden kann.

```
1 from __future__ import print_function
```

Hier noch ein Beispiel.

```
1 from math import sin, cos, pi
2 print( sin(pi/3.0), cos(pi/3.0) )
```

An dieser Stelle machen wir noch auf Abschnitt 9.3 indem wir eine Hand voll **Python**-Pakete erwähnen, die wir oft benutzen.

3.11. Workflow

Wir arbeiten recht erfolgreich mit folgendem Workflow.

Beginne ein Projekt in **Python**. Solange du noch nicht fertig bist: Wenn eine Funktion zu aufwendig zu implementieren ist, suche nach einer bereits vorhandene **Python** Bibliothek (es gibt bestimmt eine). Wenn eine Funktion für dein Projekt zu langsam ist, schreibe sie in **C**/**C**++ und binde sie als Modul ein. Wie genau letzteres geht, lernen wir noch.

4. Einige Standarddatentypen

In diesem Abschnitt besprechen wir einige Standarddatentypen, die **Python** bereit stellt. In den darauf folgenden Abschnitten besprechen wir "Funktionen" und "Klassen" und "Module".

4.1. Zahlen

In **Python** gibt es die numerischen Typen `int`, `bool`, `float` und `complex`. Objekte dieses Typs sind immer "immutable".

Die **Python**-Typen `int`, `float` und `complex` verhalten sich ähnlich wie die **C/C++**-Typen `int`, `float` und `complex`. Hier die wichtigsten Operationen.

```
1  x + y  # Summe von x und y
2  x - y  # Differenz von x und y
3  x * y  # Produkt von x und y
4  x / y  # Quotient von x and y
5  x // y # Abgerundeter Quotient x und y
6  x % y  # Rest von x / y
7  -x     # Negiere x
8  +x     # Tut nichts
9  x ** y # x hoch y
10 abs(x) # Betrag von x
11 int(x) # Erstellt int aus x
12 float(x)        # Erstellt float aus x
13 complex(re, im) # Erstellt complex aus re und im
14 divmod(x, y)    # Das Paar (x // y, x % y)
```

Die Division von zwei Ganzzahlen wieder eine Ganzzahl in **Python2** und eine Gleitkommazahl in **Python3**. Sind die Typen der Operationen verschieden, verhält sich **Python** ebenfalls wie **C/C++** und nimmt immer den genaueren Typ.

```
1 2   /  3   # = 0 in Python2; 0.6666666666666666 in Python3
2 2   /  3.0 # = 0.6666666666666666
3 2.0 /  3   # = 0.6666666666666666
4 2.0 // 3.0 # = 0.0
```

Des Weiteren gibt es für `int`, `float` und `complex` typenspezifische Funktionen. Für `int` gibt es beispielsweise Bitoperationen ~, `|`, `&`, `^`, `<<` und `>>`; für `float` gibt es beispielsweise die Funktion `as_integer_ration`, die `x` als Bruch darstellt; und komplexe Zahlen können beispielsweise mit der Funktion `conjugate` ihr Komplexkonjugiertes zurückgeben. Mehr solcher Funktionen sind hier zu finden [Pytc, Library, Build-in Types, Numeric Types].

Die Funktion `float(x)` erstellt aus `x` ein Objekt vom Typ `float`. Praktischerweise ist der Typ des Objekts `x` fast unbeschränkt. Hier ein Beispiel.

```
1 float(700)            # Erzeugt das Objekt 700.0
2 float('   -12345\n')  # Erzeugt das Objekt -12345.0
3 float('1e-003')       # Erzeugt das Objekt 0.001
4 float('+1E6')         # Erzeugt das Objekt 1000000.0
5 float('-Infinity')    # Erzeugt das Objekt -inf
```

An dieser Stelle sei noch gesagt, dass ein Objekt vom Typ `typ` genau dann als `float` (bzw. `int` oder `complex`) interpretiert werden, wenn die zu `typ` gehörige Klasse über die speziellen Memberfunktion `__float__` (bzw. `__int__` oder `__complex__`) verfügt. Was das genau heißt, wird im Abschnitt 7 erklärt.

4.2. bool

Es gibt genau zwei Objekte vom Typ `bool`. Sie sind durch ihren Wert `True` und `False` eindeutig bestimmt. Außerdem sind sie immutable.

Als `False` interpretiert **Python** die folgende Werte: Zahlen `0` und `0.0`; leere Mengen (ja, leere Menge**n**, i.e. Objekte vom Typ `set`, deren Wert mit dem Wert `{}` übereinstimmt (selbst wenn ihre Identitäten verschieden sind)); leere Sequenzen `''`, `()` und `[]`; und leere Verzeichnisse. Allgemein kann man ein Objekt vom Typ `x` genau dann als `bool` interpretiert werden, wenn die zu `x` gehörige Klasse über eine Memberfunktion `__bool__` verfügt. Was das genau heißt, wird im Abschnitt 7 erklärt.

Es gibt folgende boolschen Operationen.

```
1 x or  y # y wird nur ausgewertet wenn x False ist
2 x and y # y wird nur ausgewertet wenn x True ist
3   not x # False wenn x True ist und True wenn x False ist
```

4.3. Sequenzen

Sequenzen sind sogenannte “Container”, die sich zum Teil ähnlich wie die aus **C/C++** bekannten Arrays verhalten. Die wichtigsten Sequenztypen sind der mutable Typ `list`, sowie die immutable Typen `tuple` und `str`.

4.3.1. Gemeinsamkeiten

Besprechen wir zunächst die wichtigsten Operationen, die sich die drei Typen `list`, `tuple` und `str` teilen.

```
1 a = [x,y,z] # a ist Typ list  mit a[0]=x, a[1]=y, a[2]=z
2 a = (x,y,z) # a ist Typ tuple mit a[0]=x, a[1]=y, a[2]=z
3 a = 'xyz'   # a ist Typ str   mit a[0]=x, a[1]=y, a[2]=z
4 a[i]        # Die Variable a[i]
5 len(a)      # Laenge von a
```

Für nicht-negative Ganzzahlen `i` ist völlig klar was damit gemeint ist. Allerdings sind auch negative Werte für `i` zulässig: hier benutzt **Python** den Index `len(a) + i`. Insgesamt sind nur Werte `-len(a) <= i < len(a)` zulässig.

Um zu überprüfen, ob (der Wert des Objekts der Variable) `x` mindestens einmal in `a` vorkommt oder nicht, gibt es die folgenden Operationen.

```
1 x in a       # True  genau dann wenn x in a enthalten ist
2 x not in a   # False genau dann wenn x in a enthalten ist
```

Um zwei Container desselben Typs (also `list`, `tuple` oder `str`) hintereinander zu heften gibt es folgende Operationen. Dabei entsteht immer ein neues Objekt.

```
a + b # Erstelle die Verkettung von a und b (erst a dann b)
n * a # a + a + ... + a (n mal)
```

Beispielsweise erstellt man eine Liste der Größe `n` mit Einträgen `0` wie folgt.

```
thor = n*[0] # Die Sequenz [0, 0, ...., 0] der Laenge n
```

Einen Teilsequenz kann man wie folgt bekommen.

```
b=a[i:j]   # b[t] = a[i+t]   mit i <= i+t   < j
b=a[i:j:k] # b[t] = a[i+t*k] mit i <= i+t*k < j
```

Für nicht-negative `i` und `j` ist völlig klar was damit gemeint ist. Auch hier sind negative Werte für `i` und `j` zulässig solange `-len(a) <= i,j < len(a)` erfüllt ist.

```
a[2:-4] # Die Variablen a[2], ..., a[n-4] fuer n = len(a)
```

Außerdem können `i` und `j` weggelassen werden. Dann ist `i = 0` bzw. `j = len(a)`.

```
a[ :-4] # Die Variablen a[0], ..., a[n-4] fuer n = len(a)
a[2:  ] # Die Variablen a[2], ..., a[n]   fuer n = len(a)
```

Bei diesen Teilsequenzen ist es wichtig zu wissen, dass hier sogenannte "oberflächeliche Kopien" erstellt werden. Was das ist und welche (ungeahnten) Konsequenzen das haben kann, behandeln wir im Abschnitt 4.3.2.

Weitere nützliche Operationen sind die Folgenden.

```
min(a)      # Kleinstes Element von a, falls sinnvolle Frage
max(a)      # Groesstes Element von a, falls sinnvolle Frage
a.count(x)  # Anzahl der Variablen die gleich x sind.
a.index(x)  # Index von der ersten Variablen, die gleich x ist
a.index(x, i)     # Index von der ersten Variablen, die gleich x ist sowie
                  # gleich oder nach i kommt
a.index(x, i, j)  # Index von der ersten Variablen, die gleich x ist sowie
                  # gleich oder nach i sowie vor j kommt
```

4.3.2. Oberflächeliche und tiefe Kopien

Eine Sequenz ist ein Objekt. Der Wert einer Sequenz ist eine linear geordnete Menge von Variablen. Ja richtig, der Wert einer Sequenz ist eine linear geordnete Menge von Variablen (und nicht von Objekten). Deshalb ist eine sogenannte "oberflächliche Kopie" einer Sequenz `a` ein neues Objekt `b` mit dem selben Wert wie `a`. Das heißt, `a[0]` und `b[0]` zeigen auf dasselbe Objekt. Schauen wir uns das folgende Beispiel an:

```
# In diesem Beispiel bezeichnen wir Listen der Form [i,j] wobei i und j
# Ganzzahlen sind als Vektoren.

# Zuerst erstellen wir eine Liste von drei Vektoren und kopieren die Liste
vekt_lst = [ [0,1], [2,3], [5,1] ] # Liste von drei Vektoren
kopie    = vekt_lst[0:3]           # Oberflaechliche Kopie
```

```
7
8 # Die Listen sind verschieden
9 print(id(vekt_lst)) # 140541651867528
10 print(id(kopie))    # 140541679766536
11
12 # Die Elemente von einer der Listen zu aendern hat folgenden Effekt
13 v = kopie[0] # zeigt auf denselben Vektor wie kopie[0] und vekt_lst[0]
14 v[0] = 100   # v wird geaendert
15 print(v)         # [100,1]
16 print(kopie)     # [[100, 1], [2, 3], [5, 1]]
17 print(vekt_lst) # [[100, 1], [2, 3], [5, 1]]
```

Dieses Verhalten ist in den meisten Fällen gewollt. Wenn man eine "tiefe Kopie" der Sequenz `a` braucht, also eine Sequenz `b`, deren "mutable" Objekte denselben Wert, aber eine verschiedene Identität haben, dann benutzt man das Modul `copy` (siehe auch Abschnitt 9.3.2). Das geschieht wie folgt.

```
1 # In diesem Beispiel bezeichnen wir Listen der Form [i,j] wobei i und j
2 # Ganzzahlen sind als Vektoren.
3
4 import copy
5
6 # Zuerst erstellen wir eine Liste von drei Vektoren und kopieren die Liste
7 vekt_lst = [ [0,1], [2,3], [5,1] ] # Liste von drei Vektoren
8 kopie    = copy.deepcopy(vekt_lst) # Tiefe Kopie
9
10 # Die Listen sind verschieden
11 print(id(vekt_lst)) # 140541651867528
12 print(id(kopie))    # 140541679766536
13
14 # Die Elemente von einer der Listen zu aendern hat folgenden Effekt
15 v = kopie[0] # zeigt auf denselben Vektor wie kopie[0]
16 v[0] = 100   # v wird geaendert
17 print(v)         # [100,1]
18 print(kopie)     # [[100, 1], [2, 3], [5, 1]]
19 print(vekt_lst) # [[  0, 1], [2, 3], [5, 1]]
```

4.3.3. tuple

Der Containertyp `tuple` ist immutable. Das bedeutet, dass weder einzelne Einträge des Tupels verändert werden können, noch kann die Länge des Tupels verändert werden. Das Tupel `(3, 'Harry␣Potter')` beispielsweise wird über den gesamten Verlauf des Programs das Tupel `(3, 'Harry␣Potter')` bleiben und kann nicht zu z.B. `(1, 'LotR')` geändert werden.

Dennoch nutzt man Tupel implizit rechts häufig. Unter anderem können, wie im Crashkurs Abschnitt 3.9 bereits angedeutet, Funktionen mehrere Objekte zurückgeben. Diese werden dann zusammen in ein Tupel gepackt, in der Reihenfolge, in der sie nach dem `return`-Statement aufgezählt sind. Das folgende Beispiel veranschaulicht dies:

```
1 def doppeltupelfkt(x,y)
2   return 2*x, 2*y
```

```
3
4 z = doppeltupelfkt(1,2)
5 print( z )                # gibt (2,4) aus
```

`z` ist nach Aufrufen der Funktion das Tupel `(2,4)`, d.h. wenn eine Funktion mehrere Objekte zurückgibt, werden sie in einem Objekt abgefangen.

Außerdem gibt es für Tupel noch die Möglichkeit der mehrfachen Zuordnung:

```
1 x_1, x_2, ... , x_n = z # z ist Tupel mit len(z) = n
```

Hierbei wird `z[i-1]` der Variable `x_i` zugeordnet. Auf diese Weise können auch die Objekte, die eine Funktion zurückgibt, in mehreren Variablen abgespeichert werden statt in nur einem Tupel. Dabei muss darauf geachtet werden, dass auf der linken Seite der Zuordnung genau so viele Variablen stehen wie das Tupel Elemente hat.

Ansonsten verfügt `tuple` über keine weiteren (für diesen Kurs bedeutsamen) Operationen — abgesehen von den Gemeinsamkeiten mit anderen Sequenzen, die im vorangegangenen Abschnitt 4.3.1 beschrieben wurden.

4.3.4. list

Den mutable Containertyp `list` verwendet man in **Python** recht oft. Neben den Operationen, die alle Sequenzen gemeinsam haben, können wir eine gegebene Liste `a` mit den nachfolgenden Operationen verändern.

Folgende Operationen fügen Objekte hinzu oder entfernen sie.

```
1 a[i] = x          # Ersetze a[i] durch x
2 a.insert(i, x)  # Fuege x hinter a[i] ein
3 a.append(x)      # Fuege x am Ende von a ein
4 x = a.pop([i])  # Erhalte x = a[i] und entferne a[i] aus der list
5 a.remove(x)      # Entferne die erste Variable mit Wert x
```

Folgende Operationen verlängern eine gegebene `list` mit Namen `a` um eine (andere) gegebene `list`.

```
1 a += t # Verlaengert a mit t
2 a *= n # Verlaengert a n mal mit sich selbst
```

Wir machen darauf aufmerksam, dass `a+=t` nicht dasselbe ist wie `a=a+t`. Der erste Ausdruck verlängert `a` um `t`, der zweite Ausdruck erstellt ein neues Objekt mit Wert `a+t` und danach referenziert `a` auf dieses Objekt. Schauen wir uns dazu folgendes Beispiel an.

```
1 thor = ['Valhalla', 42] # Erstellt thor
2 print(id(thor))          # 140541679765448
3 thor += ['Valgrind']     # fuegt 'Valgrind' hinzu
4 print(id(thor))          # 140541679765448
5 thor = thor + ['Suppe'] # erstellt neue Liste und nennt sie thor
6 print(id(thor))          # 140541679767112
```

Listen können auch scheibchenweise mit nur einer Operation verändert werden.

```
a[i:j] = t    # Entferne den slice a[i:j] und fuege dort die Liste t ein
del a[i:j]    # Entferne den slice a[i:j]. Aequivalent zu a[i:j] = []
a[i:j:k] = t  # Ersetze den slice a[i:j:k] durch die Elemente der Liste t
del a[i:j:k]  # Entferne den slice a[i:j:k]
```

Außerdem können wir eine Liste leeren oder eine oberflächliche Kopie anfertigen. Das funktioniert so:

```
a.clear() # Entfernt alle Elemente von a
a.copy()  # Erstellt eine oberflaechliche Kopie von a
```

Eine oberflächliche Kopie erstellt eine neue Liste mit denselben Variablen. Insbesondere sind die referenzierten Objekte der beiden Listen identisch. Das will man oft, da so eine oberflaechliche Kopie sehr schnell erstellt ist. Manchmal möchte man aber, dass die Objekte verschieden sind. Wie das funktioniert, wird im Abschnitt 4.3.2 besprochen.

Für Schleifendurchläufe (oder auch andere Situationen) kann es sinnvoll sein, eine Liste umzudrehen. Außerdem möchte man Listen sortieren können, sofern das Sinn macht. Dafür stehen diese beiden vielsagenden Funktionen bereit.

```
a.reverse() # Kehrt a um
a.sort()    # Sortiert a, falls das Sinn macht
```

Genauer gesagt kann man eine Liste sortieren, sofern je zwei Elemente `x` und `y` der Liste mit `x < y` verglichen werden können. Siehe hierzu auch 7.3.

4.3.5. range

Ein sogenanntes "Rangeobjekt" ist eine immutable Sequenz von aufsteigenden ganzen Zahlen, die sich so ähnlich wie ein Tupel verhält. Es gibt drei einfache Möglichkeiten ein Rangeobjekt zu erstellen. Um ein Rangeobjekt der Zahlen `0 <= i < n` zu erstellen, ruft man die Funktion `range(n)` auf. Um ein Rangeobjekt der Zahlen `m <= i < n` zu erstellen, ruft man die Funktion `range(m,n)` auf. Um ein Rangeobjekt der Zahlen `m <= i*k < n` zu erstellen, ruft man die Funktion `range(m,n,k)` auf.

```
range(n)      # (0, 1, ..., n-1)
range(m,n)    # (m, m+1, ..., n-1
range(m,n,k)  # (m, m+k, ...., m+l*k) mit m+l*k < k <= m+(l+1)*k
```

Für `n`, `m` und `k` sind auch negative Werte möglich, solange dabei eine endliche Sequenz von Zahlen erstellt werden kann.

4.3.6. list comprehension

Die sogenannten *list comprehension* ist eine sehr praktische Möglichkeit um aus bereits vorhandenen Sequenzen neue Listen zu erstellen. Hier ist `a` eine Sequenz und wir wollen all diejenigen `x` aus `a` haben, die eine gewisse Bedingung `cond(x)` erfüllen.

```
[ x for x in a if cond(x) ]
```

Außerdem können wir statt `x` eine (von `x`) abhängende) Expression `expr(x)` in unsere neue Liste stecken.

```
1 [ expr(x) for x in a if cond(x) ]
```

Noch allgemeiner, können wir eine Expression `expra(x)` verwenden, wenn die Bedingung `cond(x)` erfüllt ist und im anderen Fall eine Expression `exprb(x)` verwenden.

```
1 [ expra(x) if cond(x) else exprb(x) for x in a  ]
```

Hier drei Beispiele. Wir wollen eine Liste aller natürlichen Zahlen $0 \leq x < 9$, die nicht durch 4 teilbar sind.

```
1 [ x for x in range(10) if x % 4 != 0 ]
```

Wir bekommen ein Tupel `a` und wollen daraus eine Liste all seiner Strings erstellen.

```
1 a = ('ss', 20 , 'rtt3', 50, 40.2, [344], ('akk', 'er') )
2 [ x for x in a if type(x) is str ]
```

Wir bekommen ein Tupel `a` der nur Strings enthält. Wir wollen eine Liste, die alle Strings aus `a` enthält, außer der String ist `'Anakonda'`, den wollen wir durch `'Python'` ersetzen.

```
1 a = ( 'blau', 'anakonda', 'Strasse', 'Anakonda', '42', 'Anakonda' )
2 [ x if x is not 'Anakonda' else 'Python' for x in a ]
```

4.3.7. str

Die immutable Sequenz `str` verfügt über sehr viele selbsterklärende Funktionen. Zum Beispiel erstellt man aus einem String `s` mit `s.lower()` einen String, der nur aus Kleinbuchstaben besteht. Wir besprechen nun noch kurz die Funktion `format` und verweisen den interessierten Leser auf [Pytc, Library, Build-in Types, Text Sequence Type].

Oft macht es Sinn, einen String durch Einsetzung von Variablen zu erzeugen. Dabei sollen die Variablen von der Beschreibung des zu entstehenden Strings getrennt sein. Dieses Konzept sollte von der **C**-Funktion `printf` bekannt sein. In **Python** kann man das am einfachsten mit der Funktionen `s.format(...)` erreichen. Dabei beschreibt `s` das Aussehen des Strings, also normalen Text sowie Platzhalter `'{}'` und die Parameter von `format` sind die Variablen, die in die Platzhalter eingesetzt werden. Hier ein Beispiel:

```
1 text = 'Hallo␣{}.␣Ich␣bin␣{}␣Jahre␣alt.'.format('Welt', 42)
2 print(text) # Hallo Welt. Ich bin 42 Jahre alt.
```

4.4. None

Es gibt genau ein Objekt `None` vom sogenannte Typ `NoneType`. Man benutzt dieses Objekt um die Abwesenheit eines Werts oder eines Parameters auszudrücken. Zum Beispiel ist `None` der Rückgabetyp einer Funktion, die zurückkehrt ohne ein Objekt mit `return` zurückzugeben. Hier ein Beispiel:

```python
def ich_geb_nix_zurueck():
  print("Ich hab nix")

x = ich_geb_nix_zurueck()
print( type(x) )
```

Ganz oft wird `None` als Standardparameter für optionale Parameter einer Funktion festgelegt. Damit lässt sich mit ganz leicht prüfen, ob der Benutzer einen optionalen Parameter übergeben hat oder nicht. Hier ein simples Beispiel:

```python
def sprich_falls_du_was_zu_sagen_hast( was=None ):
  if was is None: # Behandle als ob nichts uebergeben wurde
    print("Ich hab nix zu sagen!")
  else:           # Behandle als ob etwas uebergeben wurde
    print("Ich hab was zu sagen: '{}'.".format(was))

sprich_falls_du_was_zu_sagen_hast()     # Ich hab nix zu sagen!
sprich_falls_du_was_zu_sagen_hast(None) # Ich hab nix zu sagen!
sprich_falls_du_was_zu_sagen_hast(42)   # Ich hab was zu sagen: '42'.
```

4.5. Dateien

In **Python** ließt und schreibt man in Dateien ganz ähnlich wie in **C/C++**. Man öffnet eine Datei, ließt oder schreibt und schließt die Datei sobald man sie nicht mehr braucht.

Das Öffnen funktioniert durch Aufrufen der Funktion:

```python
open(file, mode='r')
```

Dabei ist `file` der Pfad der zu öffenden Datei und `mode` ist der sogenannte Modus. Die wichtigsten Modi fassen wir hier zusammen.

```python
'r' # Lesenzugriff
'w' # Schreibzugriff: alter Inhalt wird ueberschrieben
'a' # Schreibzugriff: beginne am Ende der alten Datei
```

Will man Lese- und Schreibzugriff haben, hängt man an `'r'`, `'w'` oder `'a'` das Zeichen `'+'` an. Also benutzen wir `'r+'` und `'w+'` für Lese- und Schreibzugriff, wobei wir am Anfang der Datei anfangen und wir benutzen `'a+'` für Lese- und Schreibzugriff, wobei wir am Ende der Datei anfangen.

Eine Datei `datei` schließt man wie folgt:

```python
datei.close() # Schliesse die Datei.
```

Es gibt einfache Möglichkeiten um aus einer geöffneten Datei `datei` zu lesen. Entweder man ließt sie zeilenweise aus `datei.readline(size=1)` oder man ließt sie zeichenweise aus `datei.read(size=1)`. Insbesondere ließt `datei.readline()` eine Zeile und `datei.read()` ließt ein Zeichen. Praktischerweise verhält sich eine geöffnete Datei wie eine Sequenz, das heißt, wir können sie ganz einfach mit in einer Schleife auslesen. In folgendem lesen und drucken wir eine Datei zeilenweise aus.

```
datei = open('dateiname.txt', 'r') # Oeffne die Datei 'dateiname.txt'
for zeile in datei:                # Fuer jede Zeile in der Datei
  print(zeile)                     # Drucke die Zeile aus
datei.close()                      # Schliesse die Datei.
```

In eine Datei schreiben wir wie folgt:

```
datei.write(string)
datei.writelines(liste_von_strings)
```

4.6. Mengen

In Gegensatz zu Sequenzen, die wir im Abschnitt 4.3 behandelt haben, sind Mengen “ungeordnete Container”. Ähnlich wie bei Cantor, verstehen wir unter einer Menge die Zusammenfassung von bestimmten, wohlunterschiedenen Objekten zu einem Ganzen. Allerdings verhalten sich Mengen in **Python** anders als “mathematische Mengen” (sie erfüllen zum Beispiel nicht das Extensionalitätsaxiom, i.e. sie sind nicht eindeutig durch ihre Elemente bestimmt). Das spielt in der Praxis keine Rolle und ist gerüchten zu Folge nur für zwangsgestörte Möchtegernmathematiker ein Problem.

4.6.1. Mengen erzeugen

Eine Menge kann aus einer (möglicherweise) leeren Sequenz erzeugt werden, sofern jedes Element der Sequenz “hashbar” ist. In Abschnitt 7.5 erklären wir genauer, was “hashbare Objekte” sind. Für den Moment begnügen wir uns damit, dass die Standardtypen genau dann hashbar sind, wenn sie immutable sind. Mengen selbst können entweder mutable oder immutable sein. Die mutable Variante ist `set`, die immutable Variante ist `frozenset`.

Um eine Menge zu erzeugen, welche die Objekte `4`, `'Python'` und `None` zu enthält, rufen wir `set` mit einer Liste oder einem Tuple das diese Elemente enthält auf.

```
s = set( [4, 'Python', None] )
```

4.6.2. Operationen für set und frozenset

Folgende Operationen verändern die Menge nicht und stehen somit sowohl `set` als auch `frozenset` zur Verfügung.

```
len(s)          # Gibt die Anzahl der Elemente zurueck
x in s          # True  gdw x in s vorkommt
x not in s      # False gdw x in s vorkommt
s == t          # True gdw ihre Elemente uebereinstimmen
s.isdisjoint(t) # True gdw s und t disjunkt sind
s <= t          # True gdw s eine Teilmenge von t ist
s < t           # True gdw s eine echte Teilmenge von t ist
s >= t          # True gdw t eine Teilmenge von s ist
s > t           # True gdw t eine echte Teilmenge von s ist
s | t | ...     # Die Vereinigung von s und t und ...
```

```
11 s & t & ...       # Der Schnitt von s und t und ...
12 s - t - ...       # Die Differenz von s und t und ...
13 s ^ t             # Die Symmetrische Differenz von s und t
14 s.copy()          # Erstelle eine oberflaechliche Kopie von s
```

Die Konzepte der oberflächlichen und tiefen Kopien sowie die daraus resultierenden Konsequenzen erklären wir in Abschnitt 4.3.2 am Beispiel der Listen. Analoge Aussagen gelten für Mengen.

An dieser Stelle gehen wir noch ein wenig auf die Gleichheit von Mengen ein. Wie gesagt, wertet `s == t` genau dann als `True` aus, wenn die Elemente von `s` und `t` übereinstimmen. Das heißt aber nicht, dass `s` und `t` dieselbe Identität haben, also an der selben Stelle im Speicher stehen. Hier ein Beispiel.

```
1 s = set( range(1,11,1) ) # Die Menge {1,2,...,10}
2 t = set( range(1,11,1) ) # Die Menge {1,2,...,10}
3 print( s == t )          # True
4 print( id(s) )           # 139836709444296
5 print( id(t) )           # 139836709038568
```

4.6.3. Operationen für set

Jede Menge `s` vom Typ `set` ist mutable. Das heißt, `s` ist ein Objekt, der Wert von `s` ist die Menge seiner Elemente und wir können den Wert von `s` verändern. Dazu stehen uns folgende Operationen bereit.

```
1  s |= t | ... # Erweitert die Menge s um t, ...
2  s &= t & ... # Entfernt alle Elmente aus s, die nicht in t,... sind
3  s -= t | ... # Entfernt alle Elemente aus s, die in t, ... sind
4  s ^= t       # Entfernt alle Elemente aus s, die nicht gleichzeitig
5               # in s und t sind
6  add(e)       # Erweitert s um e
7  remove(e)    # Entfernt e aus s; loest KeyError aus, wenn e nicht in s ist
8  discard(e)   # Entfernt e aus s falls e in s ist
9  pop()        # Entfernt ein Element aus s und gibt es zurueck
10 clear()      # Entfernt alle Elemente aus s
```

Wir bemerken hier nochmal, dass die Operationen nur den Wert der Menge `s` verändern, die Identität `id(s)` bleibt bestehen.

```
1 s = set( range(1, 6,1) ) # Die Menge {1,2,...,5}
2 t = set( range(6,11,1) ) # Die Menge {5,6,...,10}
3 print( id(s) )           # 139836709038792
4 print( id(t) )           # 139836709444296
5 s.add(11)                # Fuege zu s das Element 11 hinzu
6 print( id(s) )           # 139836709038792
7 print( id(s |  t) )      # 139836709037448
8 s |= t                   # Fuege zu s die Elemente 5,6,...,10 hinzu
9 print( id(s) )           # 139836709038792
```

Insbesondere sind die Objekte `s |= t` und `s = s | t` verschieden.

4.7. Verzeichnisse

Objekte von Typ `dict` nennt man "Verzeichnisse". Ein Verzeichnis verhält sich wie eine partiell definierte mathematische Funktion, deren Definitionsbereich die Menge der "hashbaren Objekte" ist und deren Wertebereich die Menge aller Objekte ist. In anderen Worten: Genau wie partiell definierte Funktionen, kann man sich ein Verzeichnis `d` wie eine Menge von Tupeln `(key, d[key])` vorstellen, wobei die linke Seite eines jeden Tupels höchstens einmal als linke Seite vorkommt. Wenn man bei dieser Vorstellung bleibt, sagt man dass `d` auf `key` definiert ist, wenn `d` das Tupel `(key, d[key])` enthält. Diejenigen Objekte, auf denen ein Verzeichnis definiert ist, nennt man "Schlüssel". Schlüssel sind immer hashbare Objekte (siehe unten). Ist `key` ein Schlüssel einer Liste `d`, so ist `d[key]` ein "Wert" des Verzeichnisses `d`.

Noch haben wir nicht erklärt, was hashbare Objekte sind. Hier sei gesagt, dass alle vom **Python** Standard definierten Objekte hashbar sind, sofern sie immutable sind (also Objekte vom Typ `int`, `float`, `complex`, `tuple`, `str`, `bool` und `none`, aber nicht `list` und `dict`). Wir erklären hashbare Objekte in Abschnitt 7.5.

4.7.1. Verzeichnisse erstellen

Verzeichnisse können auf folgende Arten erstellt werden:

```
1 # Leere Verzeichnisse
2 verz1 = {}
3 print(verz1)
4 verz2 = dict()
5 print(verz2)
6
7 # Nichtleeres Verzeichnis
8 verz3 = { key1 : value1, key2 : value2, key3 : value3, ... }
9 print(verz3)
```

`verz1` und `verz2` sind leere Verzeichnisse, `verz3` verdeutlicht die Syntax eines nichtleeren Verzeichnisses. Die "Schlüssel"-"Wert"-Paare werden durch Doppelpunkte getrennt, die Paare voneinander durch Kommata. Ein Beispiel für ein nichtleeres Verzeichnis könnte folgendermaßen aussehen:

```
1 #Verzeichnis speichert Telefonnummern:
2 telefonbuch = {'Mama':1234, 'Papa':1234, 'Tina':4242, 'Peter':'Keine Nummer'}
```

Außerdem kann man ein Verzeichnis aus einer Liste von Tupeln erstellen (solange die linke Seite eines jeden Tupels ein hashbares Objekt ist). Hier ein Beispiel:

```
1 verz_eins = { 42 : 'Wahrheit', True : 'Wahrheit' }
2 verz_zwei = dict([(42, 'Wahrheit'), (True, 'Wahrheit')])
3 print(verz_eins, verz_zwei)
```

4.7.2. Verzeichnisse nutzen

Mit den folgenden Funktionen, kann man sich die Größe eines Verzeichnisses `d` ausgeben lassen, auf Werte zugreifen, sowie (neue) Schlüssel-Wert Paare setzen oder löschen.

```
len(d)          # Groesse des Verzeichnisses
d[key]          # Gibt den Wert d[key] zurueck oder loest eine Ausnahme aus
d.get(key, x)   # Gibt den Wert d[key] zurueck falls er existiert. Sonst
                # wird das Paar (key, x) erstellt und x zurueckgegeben.
d[key] = wert   # Setzt d[key] auf wert (auch wenn key noch nicht existiert)
del d[key]      # Entfernt (key, d[key]) oder loest eine Ausnahme aus
d.pop(key, x)   # Entfernt (key, d[key]) und gibt d[key] zurueck falls es
                # existiert. Sonst wird x zurueckgegeben.
d.popitem()     # Entfernt bel. Paar (key, d[key]) und gibt d[key] zurueck
d.clear()       # Entfernt alle Paare aus dem Verzeichniss.
```

Um nachzuschauen, ob ein Schlüssel in einem Verzeichniss `d` definiert ist oder nicht nutzen wir die folgenden Operationen.

```
key in d        # Gibt True  zurueck gdw key ein Schluessel von d ist
key not in d    # Gibt False zurueck gdw key ein Schluessel von d ist
```

Eine oberflächeliche Kopie eines Verzeichnisses `d` zu erstellen, benutzt man die folgende Operation.

```
d.copy()
```

Die Konzepte der oberflächlichen und tiefen Kopien sowie die daraus resultierenden Konsequenzen erklären wir in Abschnitt 4.3.2 am Beispiel der Listen. Analoge Aussagen gelten für Verzeichnisse.

An dieser Stelle gehen wir noch ein wenig auf die Gleichheit von Verzeichnissen ein. Der Vergleich `s == t` wertet genau dann als `True` aus, wenn die Paare von `s` und `t` übereinstimmen. Das heißt aber nicht, dass `s` und `t` dieselbe Identität haben, also an der selben Stelle im Speicher stehen. Hier ein Beispiel.

```
s = {1: 'Eins'}
t = {1: 'Eins'}
print( s == t )           # True
print( id(s) )            # 140601528318536
print( id(t) )            # 140601528344776
```

Um eine Sequenz der Schlüsse, der Werte oder der Schlüssel-Wert-Paare eines gegebenen Verzeichnisses `d` zu erhalten, nutzen wir die folgenden Operationen.

```
d.keys()   # Gibt eine Sequenz der Schluessel zurueck
d.values() # Gibt eine Sequenz der Werte zurueck
d.items()  # Gibt eine Sequenz der Schluessel-Wert-Paare zurueck
```

Zum Beispiel druckt man die Schluessel-Wert-Paare eines Verzeichnisses `d` so aus:

```
d = { 42 : 'Wahrheit', True : 'Wahrheit' }

for key,val in d.items():
  print("Schluessel: '{}', Wert: '{}'".format(key, val))
```

5. Einschub: Namespacing

Wie wir bereits wissen, organisieren wir unseren **Python**-Code in Codeblöcken. Immer wenn wir eine neues **Python**-Skript beginnt, zählen wir das als eigenen Block; Immer wenn wir Code einrücken ist dieser Code ein eigener Block.

In so gut wie allen Blöcken definieren wir viele Variablen, zum Beispiel Klassennamen, Objekte und Funktionen. Es ist unvermeidbar, dass wir Variabelnamen mehrfach vergeben. Zum Beispiel haben viele Module eine Funktion mit dem Namen `main`, die nur dann aufgerufen wird, wenn das Modul das main-Modul ist (vergleiche Abschnitte 6.2 und 9). Bei der Organisation der Variablnamen stellt sich **Python** klüger (wenn auch nicht notwendigerweise effizienter) als andere Sprachen an. In **Python** werden Variablen zuerst lokal gesucht und gesetzt und erst dann global. Insbesondere haben wir nicht ganz viele globale Variablen, die alle `main` heißen.

Genauer legt **Python** die Struktur eines gerichteten Baums an. Dabei wird jeder Modul- Klassen und Funtktionsblock als Knoten verstanden und es verläuft genau dann einen gerichteten Weg von `A` nach `B`, wenn der Codeblock `B` ein Unterblock vom Codeblock `A` ist.

Wird eine Variable in einem Codeblock `A` definiert, so liegt sie am entsprechenden Knoten im Baum. Wir nennen die Variable "lokal bezüglich `A`". Wenn wir zum Beispiel in einem Modul `modul` zwei Funktionen `f` und `g` definieren, so sind `f` und `g` lokal bezüglich `modul`, aber weder `f` ist lokal bezüglich `g` noch anders herum.

Wenn eine Variable `x` im Codeblock `A` ausgeweret werden soll, sucht **Python** den Namen `x` erstmal im Codeblock `A`. Die Variable `x` kann in `A` definiert sein, muss sie aber nicht. Zum Beispiel wollen wir die Möglichkeit haben, Funktionen aufzurufen, die nicht in `A` definiert sind. Wurde die Variable gefunden, so nennen wir die Variable "lokal bezüglich `A`". Wenn sie nicht in `A` gefunden wurde, durchsucht **Python** den genannten Baum skuzessive aufwärts. Wurde die Variable nicht in `A`, aber in einem Knoten über `A` gefunden, so nennen wir die Variable "global bezüglich `A`".

Falls die Variable nicht gefunden werden kann, wird eine Ausnahme ausgelöst (vergleiche Abschnitt 8).

6. Funktionen

Je länger und komplexer Programme werden, desto häufiger kommt es vor, dass Codeblöcke immer wieder vorkommen oder es für die Übersichtlichkeit des Programmcodes gut wäre, wenn Codeabschnitte, die eine einzelne Aufgabe erfüllen, zusammengefasst werden könnten. Meist wird dieser Code dann in Funktionen ausgelagert.

In diesem Kapitel besprechen wir zunächst die Syntax von Funktionen in **Python** und gehen auf die Konstruktion einer "main-Funktion" ein. Zum Abschluss des Kapitels erinnern wir noch einmal daran, dass in Python alle Konstrukte Objekte sind (außer Variablen) und zeigen, welche Konsequenzen das für Funktionen haben kann.

6.1. Funktionen definieren und aufrufen

Funktionsdefinitionen in **Python** haben folgende einfache Syntax:

```
1 def funktionsname(par_1, ..., par_k, rap_1=wert_1, ..., rap_l=wert_l):
2   ausdruck
3   ...
```

Im Crashkurs, Abschnitt 3.9, haben wir bereits eine vereinfachte Version der Funktionsdefinition gesehen und auch, dass man Parametern Standardwerte zuweisen kann. Dies ist die allgemeine Form. Wir betonen noch einmal (weil es nicht häufig genug betont werden kann), dass die Funktionsparameter keinen fest zugewiesenen Datentyp haben, auch nicht die, die einen Standardwert zugewiesen bekommen haben.

Wie immer gilt, dass man eine gute Programmiererin daran erkennt, dass die Funktionen gut auskommentiert und dokumentiert sind. Die sinnvollste Art, eine Funktion (oder auch ein anderes Objekt) in Python zu dokumentieren ist der sogenannte Doc-String, der direkt unter dem Funktionskopf steht und von jeweils drei doppelten Anführungszeichen pro Seite eingerahmt wird:

```
1 def funktionsname(par_1, ..., par_k, rap_1=wert_1, ..., rap_l=wert_l):
2   """ Ausfuehrliche Dokumentation """
3   ausdruck
4   ...
```

Im Doc-String sollten - falls vorhanden - Eingabeparameter und Rückgabewerte der Funktion erklärt werden, sowie eventuelle Ausgaben.

Der Vorteil des Doc-Strings im Vergleich zu konventionellen Kommenaren ist, dass **Python** in der Lage ist, ihn mit Hilfe der `help`-Funktion auszugeben. Dazu ruft man `help(funktionsname)` auf, woraufhin der Interpreter einen Editor öffnet und den Funktionskopf zusammen mit dem Doc-String der Funktion anzeigt. Dieses Feature ist vor allem nützlich, wenn man direkt im Interpreter programmiert und sich daran erinnern möchte, wie die Funktion zu benutzen ist.

Ein kleines Beispiel für eine Funktion, die keine Parameter bekommt und nichts (also immer `None`) zurückgibt, aber deshalb nicht weniger dokumentiert werden sollte:

```
1 import random
2 def moechtegern_mathematiker():
```

```
  """Approxomiert den Output eines moechtegern Mathematikers."""
  if( random.random() < 0.5 ):
    print("Aber das mit dem Einruecken ist doof!")
  else:
    print("Aber C hat eine bessere Laufzeit!")

help(moechtegern_mathematiker)
moechtegern_mathematiker()
```

Führt man diesen Code aus, so wird zunächst `help(moechtegern_mathematiker)` ausgeführt, d.h. es wird ein Editor geöffnet, in dem Funktionskopf und Doc-String der Funktion `moechtegern_mathematiker()` angezeigt werden. Nachdem der Editor geschlossen wurde, wird die Funktion selbst ausgeführt.

Es ist möglich, Funktionen eine variable Anzahl von Parametern zu übergeben. Ein Stern vor einem der Parameternamen signalisiert **Python**, dass es hier eine nicht weiter spezifizierte Anzahl von Elementen zu erwarten hat. Diese Anzahl kann auch Null sein. Man kann es sich so vorstellen, dass eine Liste von Argumenten erwartet wird, die natürlich auch leer sein kann.

```
def funktionsname(parameter, *rest_als_liste):
```

Der `*`-Operator kann auch bei einem Funktionsaufruf verwendet werden. In diesem Fall schreibt man ihn vor eine Sequenz (meistens vor einen Variablennamen) und signalisiert damit, dass man die Sequenz gerne "entpacken" möchte, also ihre einzelnen Einträge der Funktion als Parameter übergeben möchte.

Das folgende Beispiel zeigt beide Nutzvarianten des `*`-Operators in Funktionsdefinitionen und -aufrufen:

```
import random
def moechtegern_mathematiker( *gejammer ):
  """Approxomiert den Output des lieblings moechtegern Mathematikers.
     gejammer ist das lieblings Gejammer."""
  print( random.choice(gejammer) )

moechtegern_mathematiker("Aber C ist besser!", "Aber C++ ist besser!")

lieblings_gejammer = (
  "Aber das mit dem Einruecken ist doof!",
  "Aber C hat eine bessere Laufzeit!",
  "Aber wir koennen doch schon C++!",
  "Aber wenn man reine Mathematik macht, braucht man keine Computer!",
  "Aber im Internet steht, dass Python doof ist!",
  "gcc Warnungen kann ich ignorieren, ich weiss ja was ich tue!",
  "Aber ich kann doch schon Python, nur was ist dieses Objektorientiert?!"
)

moechtegern_mathematiker( *lieblings_gejammer )
```

Die Funktion `moechtegern_mathematiker` erwartet eine beliebige Anzahl an Parametern, bzw. sie erwartet als Parameter eine entpackte Sequenz mit einer beliebigen Anzahl

Einträge, die unter `gejammer` zusammengefasst sind. Aus diesen wählt sie einen Eintrag zufällig aus und gibt ihn auf der Standardausgabe aus.

Im ersten Aufruf der Funktion werden ihr zwei Strings als Parameter übergeben. Für den zweiten Funktionsaufruf wird zunächst die Liste `lieblings_gejammer` definiert, die eine größere Anzahl an Strings enthält. Danach rufen wir die Funktion mit `*lieblings_gejammer` auf, d.h. die Einträge der Liste werden entpackt und der Funktion als Argumente übergeben.

Man kann den `*`-Operator auch zum Entpacken von Sequenzen verwenden, wenn die Funktion eine feste Anzahl von Parametern erwartet. Natürlich muss man dann aufpassen, dass die Länge der Sequenz mit der Anzahl der erwarteten Parameter übereinstimmt.

Es ist auch möglich, eine beliebige Anzahl von Schlüssel-Parameter-Paaren (also ein entpacktes Verzeichnis) zu übergeben. In der Funktionsdefinition wird dies mit `**` vor dem Parameternamen angekündigt:

```
1 def funktionsname(parameter, **rest_als_verzeichnis):
```

Wie auch mit Sequenzen (s.o.) hat man jetzt zwei Möglichkeiten, eine solche Funktion aufzurufen, nämlich einmal indem man die Parameter direkt übergibt oder indem man ein Verzeichnis mit dem `**`-Operator entpackt. Das folgende Beispiel zeigt dies etwas genauer:

```
1  import random
2  def moechtegern_mathematiker( **gejammer ):
3    """Approxomiert den Output des lieblings moechtegern Mathematikers.
4       gejammer ist das lieblings Gejammer."""
5    eigenschaft, adjektiv = random.choice(list(gejammer.items()))
6    print( "Aber " + eigenschaft + " ist " + adjektiv + "!" )
7
8  moechtegern_mathematiker(C="besser", Python="doof")
9
10 lieblings_gejammer = {
11   "Einruecken" : "doof",
12   "Laufzeit" : "doof",
13   "angewante Mathematik" : "auch doof",
14   "Freizeit" : "ganz doof",
15   "Algebra" : "besser"
16 }
17
18 moechtegern_mathematiker( **lieblings_gejammer )
```

Wenn man die Parameter direkt übergibt, erfolgt die Zuordung Schlüssel-Parameter nicht über den Doppelpunkt wie wenn man ein Verzeichnis erstellt, sondern über ein Gleichheitszeichen.

Wie auch bei Sequenzen ist es möglich, ein entpacktes Verzeichnis an eine Funktion zu übergeben, die eine feste Anzahl an Argumenten erwartet. In diesem Fall muss man nicht nur darauf aufpassen, dass die Anzahl der Paare des Verzeichnisses mit der Anzahl der erwarteten Argumente der Funktion übereinstimmt, sondern auch darauf, dass die Schlüssel im Verzeichnis mit den Parameternamen übereinstimmen.

6.2. Wo ist die main-Funktion?

In **Python** gibt es keine main-Funktion sondern ein main-Modul, siehe Abschnitt 9. Optimalerweise sieht ein **Python**-Skript so aus:

```python
# Module importieren

# Klasse definieren

# Funktionen definieren

def main_funktion():
  # Hier eine Art main-Funktion definieren.

if __name__ == '__main__':
  main_funktion()
```

6.3. Funktionen sind auch nur Objekte

Wir wiederholen hier nochmal, dass Funktionen Objekte sind. Insbesondere können sie Argumente einer anderen Funktion `F` sein.

Schauen wir uns ein einfaches Beispiel an. Angenommen, wir haben eine aufwendige Funktion `aufwendige_funk`, die `k` Parameter braucht. Daraus wollen wir eine neue Funktion bauen, die genau dasselbe macht und genau dasselbe zurückgibt, aber noch zusätzlich die CPU-Zeit misst und ausdruckt. Dazu schreiben wir eine neue Funktion `zeit_messen`, die als erstes Argument eine Funktion `f` bekommt und als zweites Argument ein Verzeichniss `vargs` welches den Parametern der Funktion `f` entsprechen soll. Innerhalb von `zeit_messen` nehmen wir erstmal die CPU-Zeit. Dann rufen wir `f` mit den Parametern `vargs` auf und speichern die Rückgabe in `rueckgabe`. Dann nehmen wir wieder die CPU-Zeit und drucken die Differenz aus. Zu guter Letzt geben wir `rueckgabe` zurück. Der Code sieht also so aus:

```python
import time

def aufwendige_funk(par_1, ... par_k):
  # Aufwendige Funktion

def zeit_messen( f, **vargs ):
  start = time.process_time ()
  rueckgabe = f( **vargs )
  dauer = time.process_time() - start
  print("Die Funktion '{}' hat {} Sekunden CPU-Zeit verbraucht".format(
    f.__name__, dauer))
  return rueckgabe

parameter = {'par_1' : ... }
zeit_messen(aufwendige_funk, parameter)
```

Ein ganz wesentlicher Unterschied zwischen **C** und **Python** ist, dass man in **Python** beliebige Funktionen zur Laufzeit erstellen kann. Das ist in **C** nicht möglich und in **C++**

erst ab **C++11**. Schauen wir uns ein eifnaches Beispiel an. Aus der Mathematik kennen wir die Ableitungsfunktion:

$$\frac{d}{dt} : C^\infty(\mathbb{R}) \to C^\infty(\mathbb{R}) \qquad \frac{df}{dt}(x) = \lim_{t\to 0} \frac{f(x+t) - f(x)}{(x+t) - x}$$

Diese nimmt eine glatte Funktion f und definiert eine neue Funktion df/dt. Das können wir in **Python** ganz genauso machen:

```
def ableiten(f, dt=1e-4):
  dt = float(dt)          # Interpretiert dt als float
  def df_dt(x):           # Definiert df/dt
    df = f(x+dt) - f(x)
    return df / dt
  return df_dt            # Gibt df/dt zurueck

def g(t):                 # Definiert g(x) = x*x/2
  return t*t/2.0

h = ableiten(g)           # Approx. die Ableitung von g, also h(x) ~ x
```

Unsere Funktion `ableiten` dient hier nur zur Demonstration. Für numerische Berechnungen taugt sie nichts, da wir uns nicht um Auslöschung kümmern.

Für den Fall, dass wir eine Funktion definieren möchten, die nur aus einem einzigen Ausdruck besteht (der dann zurückgegeben wird), können wir folgende Abkürzung benutzen. Genauer, gehen wir davon aus, dass wir folgende Funktion haben:

```
def f(par_1, ..., par_k):
  return ausdruck
```

Dann ist diese Funktionsdefinition äquivalent zu:

```
f = lambda par_1, ..., par_k : ausdruck
```

So definierte Funktionen nennt man “Lambdafunktionen”. Man soll sie dann und nur dann benutzen, wenn der Code dadurch leserlicher wird. Unser Ableitungsbeispiel sieht mit Lambdafunktionen so aus:

```
def ableiten(f, dt=1e-4):
  dt = float(dt)          # Interpretiert dt als float
  return lambda x : (f(x+dt)-f(x))/dt

def g(t):                 # Definiert g(x) = x*x/2
  return t*t/2.0

h = ableiten(g)           # Approx. die Ableitung von g, also h(x) ~ x
```

7. Klassen

Man benutzt Klassen, um Code übersichtlicher zu gestalten. Immer wenn wir einen (mathematischen oder realen) Gegenstand modelieren wollen, dessen Zustand

(K1) durch ein Verzeichniss von Eigenschaften beschrieben wird und

(K2) der Zustand durch eine oder mehrere feste Regeln verändert wird

benutzen wir dazu sogenannte Klassen.

Bevor wir besprechen wie man Klassen in **Python** definiert und nutzt, schauen wir uns ein Beispiel für eine Klasse an. Der Zustand einer Matrix M wird durch ihre Größe und ihre Koeffizienten bestimmt. Damit erfüllen Matrizen das erste Kriterium **(K1)**. Es macht Sinn die Einträge einer Matrix abzufragen, die Einträge zu ändern und Matrizen zu addieren oder zu multiplizieren (sofern sie die richtigen Größen haben). Damit erfüllen Matrizen das zweite Kriterium **(K2)**. Wie man eine Matrix modelliert hängt von **(K1)** und **(K2)** ab. Allerdings legen **(K1)** und **(K2)** nicht völlig fest, wie die Modellierung aussehen muss. Zum Beispiel können Matrizen durch zwei natürliche Zahlen (welche die Anzahl der Zeilen und Spalten festlegt) sowie die Liste der Einträge beschrieben werden, man kann aber auch das "Compressed Sparse Row Storage" Format verwenden. Für die reine Benutzung von Matrizen spielt diese Designentscheidung keine Rolle.

An dieser Stelle machen wir noch darauf aufmerksam, wie man Klassen konzeptionell in **C** umsetzen würde. Da man aus **C** an `struct` gewöhnt ist, kann man eine Klasse durch ein `struct K` zusammen mit einer Menge von Funktionen realisieren, deren erstes Argument vom Typ `struct K*` ist. Genau so haben wir Vektoren und Matrizen in der Bibliothek `JoelixBlas` realisiert (siehe [ABH17]).

7.1. Klassen definieren und nutzen

Eine Klasse definiert man **Python** wie folgt:

```
1  class klassenname:
2    def __init__(self, weitere_parameter):
3      self.membervariable_1 = ...
4      self.membervariable_2 = ...
5      ...
6
7    def memberfunktion_1(self, weitere_parameter):
8      ...
9
10   def memberfunktion_2(self, weitere_parameter):
11     ...
12
13   ...
14
15   def __spezielle_funktion_1__(self, weitere_parameter):
16     ...
17
18   ...
```

Der Klassenname kann so wie alle Variablenamen fast frei gewählt werden. Variablen die zur Klasse gehören heißen "Membervariablen". Mit dem Präfix `self.` greift man auf sie **innerhalb der Klasse** zu (aber nicht von außen). Funktionen die zur Klasse gehören heißen "Memberfunktionen". Ihr erstes Argument muss bei ihrer **Definition innerhalb der Klasse** `self` sein, beim Zugriff wird der erste Prameter `self` nicht angegeben.

Es gibt zwei Sorten von Memberfunktionen: sogenannte "spezielle Funktionen" die mit `__` beginnen und enden und "normale Funktionen", die nicht mit `__` beginnen und enden. Mehr zu spezielle Funktionen besprechen wir in Abschnitt 7.3

Die prominenteste spezielle Funktion ist der sogenannte "Konstruktor" oder "Initialisierungsfunktion" mit dem Namen `__init__`. Sie wird beim Erstellen eines Objekts automatisch aufgerufen. Außerdem kann man beim Erstellen eines Objekts gewisse Parameter übergeben. Bei unserem Matrixbeispiel würde es Sinn machen, die Zeilen- und Spaltengröße zu übergeben. In der Initialisierungsfunktion sollen alle Membervariablen definiert und (in Abhängigkeit von den übergebenen Parametern) in einen sinnvollen Ausganszustand gebracht werden. Man darf in der Initialisierungsfunktion auch andere Memberfunktionen aufrufen. Hat der Konstruktor der Klasse `klassenname` die Parameter `p_1, ..., p_k`, so erstellt man ein Objekt vom Typ `klassenname` zu den Werten `par_1, ..., par_k` wie folgt.

```
1 # Erstellt ein Objekt vom Typ klassenname bzgl. par_1, ..., par_k.
2 # Die Variable var referenziert auf dieses Objekt.
3 var = klassenname(par_1, ..., par_k)
```

Man sagt "das Objekt `obj` ist eine Instanz der Klasse `klassenname`" wenn der Typ des Objekts `obj` die Klasse `klassenname` ist.

Zeigt eine Variable `var` auf ein Instanz von `klassenname`, so kann man auf die Membervariablen und Memberfunktionen der Instanz mithilfe des Präfixes `var.` zugreifen. Zeigt beispielsweise die Variable `var` auf ein Objekt dass eine Memberfunktion mit Namen `python_ist_cool`, dann greifen wir auf diese Memberfunktion mit `var.python_ist_cool()` zu.

7.2. Eine sehr naive Matrixklasse

Als einführendes Beispiel definieren wir eine sehr naive Matrixklasse. Dabei vernachlässigen wir alle Plausibilitätsprüfungen (zum Beispiel prüfen wir nicht, ob der Typ von `zeilen` wirklich `int` ist). Bei einer gut geschriebenen Klasse dürfen diese Plausibilitätsprüfungen natürlich nicht fehlen.

```
 1 class Matrix:
 2   """Eine sehr naive Matrixklasse"""
 3
 4   def __init__(self, zeilen=0, spalten=0):
 5     self.zeilen = zeilen
 6     self.spalten = spalten
 7     self.elemente = self.zeilen*self.spalten*[0.0]
 8
 9   def getitem(self,i,j):
10     """Gibt den Koeffizienten der Matrix in der i-ten Zeile und
```

```
       der j-ten Spalte zurueck."""
    return self.elemente[i*self.zeilen + j]

  def setitem(self, i,j ,z):
    """Setzt den Koeffizienten der Matrix in der i-ten Zeile und
       der j-ten Spalte auf den Wert z."""
    self.elemente[i*self.zeilen + j] = z

  def string(self):
    """Gibt einen String zurueck, der die Matrix beschreibt."""
    s = "Zeilen:␣{}␣Spalten:␣{}\n".format(self.zeilen, self.spalten)
    for i in range(self.zeilen):
      for j in range(self.spalten):
        s = s + "{};␣".format(self.getitem(i,j))
      s = s + "\n"
    return s
```

Mit dieser sehr naiven Definition, erstellt man eine Matrix mit drei Zeilen und vier Spalten wie folgt:

```
m = Matrix(zeilen=3, spalten=4)
```

Auf die Memberfunktionen greift man mit dem Präfix `m.` zu:

```
m.setitem(1,1,3.0) # Setzt den Eintrag in Zeile 1 und Spalte 1 auf 3.0.
print(m.string())  # Druckt die Darstellung der Matrix aus.
```

7.3. Spezielle Funktionen

In diesem Abschnitt erklären wir zunächst was “Duck-Typing” ist, listen die wichtigsten speziellen Memberfunktionen auf und verbessern unser Matrizenbeispiel aus Abschnitt 7.1.

7.3.1. Duck-Typing

Als “Duck-Typing” bezeichnet man das Konzept, dass ein Objekt durch sein Verhalten und nicht durch seinen Bauplan beschrieben wird. Der Name Duck-Typing ist an ein Gedicht von James Rileys angelehnt:

> When I see a bird that walks like a duck and swims like a duck and
> quacks like a duck, I call that bird a duck.

Auch **Python** unterstützt Duck-Typing. Zum Beispiel können viele Klassen als `bool` oder `str` interpretiert werden. Das funktioniert genau dann, wenn die Klasse die spezielle Memberfunktionen `__str__` implementiert und diese einen String zurückgibt. Dann kann man mit der Funktion `str` das Objekt als String interpretieren.

Wenn wir uns unsere sehr naive Matrixklasse aus Abschnitt 7.2 nochmal anschauen, so kann **Python** diese momentan noch nicht als String interpretieren (denn sie implementiert spezielle Memberfunktionen `__str__` noch nicht). Allerdings kann sie als String interpretiert werden, wenn wir den Namen der Memberfunktion `string` durch `__str__`

ersetzen. Soland wir das erledigt haben, kann **Python** den folgenden Code interpretieren:

```
m = Matrix(zeilen=3, spalten=4)
print(m)  # Druckt die Darstellung der Matrix aus.
```

Die Funktion `str` wird vom **Python**-Standard definiert und könnte ungefähr so aussehen:

```
def my_str( t ):
  """Nachbau der eingebauten Funktion 'str'."""
  s = t.__str__()      # Hole den String
  if type(s) is str: # Schaue ob s den richtigen Typ hat
    return s           # Gibt den String s zurueck
  else:                # Fehlerbehandlung
    typename = type(s).__name__
    raise TypeError("__str__ returned non-string (type "+typename+")")
```

7.3.2. Spezielle Memberfunktionen

In diesem Abschnitt besprechen wir kurz die wichtigsten speziellen Memberfunktionen. Dabei machen wir optionale Funktionsparameter mit `[...]` kenntlich.

Der Konstruktor wird beim Erstellen eines Objekts aufgerufen und soll das Objekt in Ausgangszustand bringen. Dem Konstruktor kann man weitere Parameter übergeben.

```
__init__(self [,...])  # Konstruktor (wird beim Erstellen ausgefuehrt)
```

Ist der Konstruktor definiert, kann man ein Objekt der Klasse `K` in Abhängigkeit von den Parametern `p_1, ..., p_r` so erstellen:

```
var = K(p_1, ..., p_r)
```

Auf den Destuktor gehen wir hier nicht ein, da wir das sogenannte "Reference Counting" und den sogenannten "Garbadge Collector" in diesem Skript nicht ausreichend behandelt haben.

Ein Objekt kann als String, Ganzzahl, Gleitkommazahl bzw. Boolean interpretiert werden, sofern die entsprechende spezielle Funktion definiert ist.

```
__str__(self)   # Konvertiert Objekt zu String
__int__(self)   # Konvertiert Objekt zu Integer
__float__(self) # Konvertiert Objekt zu Float
__bool__(self)  # Konvertiert Objekt zu Boolean
```

Ist beispielsweise `__bool__` implementiert, kann man ein Objekt `obj` mit `bool(obj)` als Boolean interpretieren.

Die nachfolgenden speziellen Funktionen ermöglichen es, zwei Objekte miteinander zu vergleichen.

```
__lt__(self, other) # Implementiert: obj <  other
__le__(self, other) # Implementiert: obj <= other
__eq__(self, other) # Implementiert: obj == other
```

```
4 __ne__(self, other) # Implementiert: obj != other
5 __gt__(self, other) # Implementiert: obj >  other
6 __ge__(self, other) # Implementiert: obj >= other
```

Hier ist es ganz wichtig darauf zu achten, dass diese Funktion konsistent implementiert werden. Zum Beispiel will man, dass wenn `obj == other` gilt dann auch `obj <= other` erfüllt ist. Außerdem muss man ganz doll aufpassen, wenn man zwei Objekte vergleichen möchte, die einen verschiedenen Typ haben. Wenn man das nicht möchte, kann man gegebenfalls eine Ausnahme auslösen:

```
1 class K:
2   ... # Konstruktor und weitere Funktionen
3   def __eq__(self, other):           # Definition der Fkt __eq__
4     if type(self) == type(other):   # Wenn die Typen gleich sind
5       ...                           # Pruefe Objekte auf Gleichheit
6     else:                           # Sonst:
7       raise TypeError("Typen sind verschieden") # Loese Ausnahme aus
8   ... # Weitere Funktionen
```

An dieser Stelle wollen wir darauf aufmerksam machen, dass das Vergleichen der Typen von zwei Objekten, so wie im obigen Beispiel, bei abgeleiteten Klassen zu ungewollten Effekten führen kann. Da wir abgeleitete Klassen in diesem Skript nicht besprechen, gehen wir nicht näher auf diese Bemerkung ein.

Um ein Objekt als Schlüssel für ein Verzeichniss machen zu können, muss es "hashbar" sein. Genauer muss man die spezielle Funktion `__hash__` implementieren. Das besprechen wir ausführlicher im Abschnitt 7.5.

```
1 __hash__(self)  # Objekt hashen
```

Um ein Objekt `obj` wie eine Funktion aufzurufen, also um dem Ausdruck `obj(...)` interpretierbar zu machen, muss die spezielle Funktion `__call__` implementiert werden. Ein Objekt, welches sich in diesem Sinne wie eine Funktion verhält, nennt man *Funktionsobjekt*.

```
1 __call__(self,[, args...]) # Objekt wird zum Funktionsobjekt
```

Um ein Objekt `obj` wie ein Verzeichniss (siehe Abschnitt 4.7) zu behandeln, implementiert man die folgenden speziellen Funktionen:

```
1 __len__(self)                  # Laenge des Objekts
2 __getitem__(self, key)         # Implementiert: obj[key]
3 __setitem__(self, key, value)  # Implementiert: obj[key] = value
4 __delitem__(self, key)         # Implementiert: del obj[key]
5 __reversed__(self)             # Implementiert: reversed(obj)
6 __contains__(self, item)       # Implementiert: is item in obj
```

An dieser Stelle machen wir darauf aufmerksam, welche Werte der Schlüssel `key` haben kann. Ein Schlüssel muss immer ein hashbares Objekt sein. Im einfachsten Beispiel sind das unveränderbare Standardtypen wie `int` oder `str`. Es können aber auch benutzerdefinierte Klassen sein, welche die spezielle Funktion `__hash__` implementieren (siehe auch Abschnitt 7.5). In diesen beiden Beispielen wird `obj[key] = z` so interpretiert:

```
obj.__setitem__(key,z)     # Aequivalent zu obj[key] = z
```

Man kann aber auch mehrere Schlüssel zum Zugriff benutzen. Diese werden dann zu einem einzigen Tupel gebündelt. Schauen uns zum Beispiel an, wie `obj[key1,key2] = z` interpretiert wird:

```
obj.__setitem__( (key_1,key_2) ,z) # Aequivalent zu obj[key1,key2] = z
```

In unserem Matrixbeispiel aus Abschnitt 7.2, würden wir sehr gern mit `M[i,j]` den Koeffizient in der i-ten Zeile und der j-ten Spalte setzen. Die spezielle Memberfunktionen `__setitem__` kann so implementiert werden:

```
class Matrix:
  """Eine native Matrixklasse"""
  ...            # Konstruktor und weitere Funktionen
  def __setitem__(self, ij ,z):
    i,j = ij     # Wir interpretieren ij als Tupel ij = (i,j)
    self.elemente[i*self.zeilen + j] = z # Setze Koeffizient
  ...            # Weitere Funktionen
```

In **Python** ist es (genauso wie in **C++**) möglich aus zwei Objekten mithilfe der binären Operatoren `+`, `-`, `*` usw. ein neues Objekt zu erstellen. Dazu implementiert man die folgenden speziellen Memberfunktionen. Wir machen hier darauf aufmerksam, dass die speziellen Memberfunktionen die beiden Objekte `obj` und `other` in so gut wie allen Fällen nicht veränderen, da man aus den beiden Objekten `obj` und `other` mithilfe des Operators `op` ein neues Objekt `obj op other` erstellt.

```
__add__(self, other)       # Implementiert: obj  + other
__sub__(self, other)       # Implementiert: obj  - other
__mul__(self, other)       # Implementiert: obj  * other
__truediv__(self, other)   # Implementiert: obj  / other
__floordiv__(self, other)  # Implementiert: obj // other
__mod__(self, other)       # Implementiert: obj  % other
__pow__(self, other)       # Implementiert: obj ** other
__lshift__(self, other)    # Implementiert: obj << other
__rshift__(self, other)    # Implementiert: obj >> other
__and__(self, other)       # Implementiert: obj  & other
__xor__(self, other)       # Implementiert: obj  ^ other
__or__(self, other)        # Implementiert: obj  | other
```

Es ist wichtig zu bemerken, dass keine die obigen binären Operationen symmetrisch ist:

```
obj1.__add__(obj2)         # obj1 + obj2
obj2.__add__(obj1)         # obj2 + obj1
```

Es kommt vor, dass man zwei Objekte von verschiedenen Typen miteinander verbinden möchte. Hier gibt es zwei Möglichkeiten. Entweder man implementiert die binären Operationen für beide Klassen oder man implementiert sie nur für eine. Letzteres macht in vielen Fällen mehr Sinn, zum Beispiel wenn eine der beiden Klassen schon von jemand anderem implementiert ist. Wie Letzteres funktioniert erklären wir jetzt.

Nehmen wir an, wir wollen zwei Objekte `obj1` und `obj2` mit einer binären Operation `op` verbinden, i.e. wir wollen den Ausdruck `obj1 op obj2` interpretierbar machen. Wir verlangen dass (a) die beiden Objekte verschiedene Typen haben und (b) das linke Objekt `obj1` die spezielle Memberfunktion `__op__` nicht implementiert hat. Dann können wir `obj1 op obj2` durch die spezielle Memberfunktion `__rop__` vom rechten Objekt `obj2` interpretierbar machen.

```
1 obj2.__rop__(obj1)          # obj1 op obj2 falls (a) und (b) gelten
```

Die folgende Liste erklärt die binären Operationen, die wir im Sinne des vergangenen Abschnitts interpretierbar machen können. Auch hier machen wir darauf aufmerksam, dass diese binären Operationen nach Möglichkeit ein neues Objekt erstellen sollen ohne dabei die Operanden zu verändern.

```
 1 __radd__(self, other)       # Implementiert: other  + obj
 2 __rsub__(self, other)       # Implementiert: other  - obj
 3 __rmul__(self, other)       # Implementiert: other  * obj
 4 __rtruediv__(self, other)   # Implementiert: other  / obj
 5 __rfloordiv__(self, other)  # Implementiert: other // obj
 6 __rmod__(self, other)       # Implementiert: other  % obj
 7 __rpow__(self, other)       # Implementiert: other ** obj
 8 __rlshift__(self, other)    # Implementiert: other << obj
 9 __rrshift__(self, other)    # Implementiert: other >> obj
10 __rand__(self, other)       # Implementiert: other  & obj
11 __rxor__(self, other)       # Implementiert: other  ^ obj
12 __ror__(self, other)        # Implementiert: other  | obj
```

Wir wissen ja bereits, dass `c = a + b` ein neues Objekt aus `a` und `b` erstellt und dass sich die Variable `c` im Anschluss darauf bezieht. Insbesondere bleiben die Objekte `a` und `b` unverändert. Dasslbe gilt für `a = a + b`. Insbesondere ist `a = a + b` etwas anderes als `a += b`. Das haben wir zum Beispiel bei Listen im Abschnitt 4.3.4 gesehen. Mit den folgenden speziellen Memberfunktionen realisiert man die Operationen der Form `a += b`, `a -= b`, usw..

```
 1 __iadd__(self, other)       # Implementiert: obj  += other
 2 __isub__(self, other)       # Implementiert: obj  -= other
 3 __imul__(self, other)       # Implementiert: obj  *= other
 4 __itruediv__(self, other)   # Implementiert: obj  /= other
 5 __ifloordiv__(self, other)  # Implementiert: obj //= other
 6 __imod__(self, other)       # Implementiert: obj  %= other
 7 __ipow__(self, other)       # Implementiert: obj **= other
 8 __ilshift__(self, other)    # Implementiert: obj <<= other
 9 __irshift__(self, other)    # Implementiert: obj >>= other
10 __iand__(self, other)       # Implementiert: obj  &= other
11 __ixor__(self, other)       # Implementiert: obj  ^= other
12 __ior__(self, other)        # Implementiert: obj  |= other
```

Wie man den Absolutbetrag und die uniären Operationen `-ob`, `+ob` und `~ob` interpretierbar macht, erklärt die nachfolgende Liste.

```
1 __abs__(self, other)       # Absolutbetrag des Objekts
2 __neg__(self, other)       # Implementiert: -obj
```

```
3 __pos__(self, other)       # Implementiert: +obj
4 __invert__(self, other)    # Implementiert: ~obj
```

7.4. Eine naive Matrixklasse

Wir verbessern hier unsere sehr naive Matrixklasse aus Abschnitt 7.2 indem wir die sehr naiven Memberfunktionen durch spezielle Memberfunktionen ersetzen. Dadurch ist der Zugriff auf die Koeffizienten einer Matrix und die Ausgabe einer Matrix viel komfortabler.

```
 1 class Matrix:
 2   """Eine naive Matrixklasse"""
 3
 4   def __init__(self, zeilen=0, spalten=0):
 5     self.zeilen = zeilen
 6     self.spalten = spalten
 7     self.elemente = self.zeilen*self.spalten*[0.0]
 8
 9   def __getitem__(self,ij):
10     """Implementiert den Zugriffsoperator []. Mit m[i,j] wird der
11        Koeffizienten der Matrix m in der i-ten Zeile und der j-ten Spalte
12        zurueckgegeben."""
13     i,j = ij
14     return self.elemente[i*self.zeilen + j]
15
16   def __setitem__(self, ij ,z):
17     """Implementiert den Zugriffsoperator []. Mit m[i,j] = z wird der
18        Koeffizienten der Matrix m in der i-ten Zeile und der j-ten Spalte
19        auf den Wert z gesetzt."""
20     i,j = ij
21     self.elemente[i*self.zeilen + j] = z
22
23   def __str__(self):
24    """Implementiert die Konvertierung zu str. Mit str(m) wird ein String
25       zurueckgegeben, der die Matrix m beschreibt."""
26     s = "Zeilen: {} Spalten: {}\n".format(self.zeilen, self.spalten)
27     for i in range(self.zeilen):
28       for j in range(self.spalten):
29         s = s + "{}; ".format(self[i,j])
30       s = s + "\n"
31     return s
```

Mit dieser verbesserten Implementierung der Klasse `Matrix` können wir viel komfortabler mit Matrizen arbeiten.

```
1 m = Matrix(zeilen=3,spalten=4) # Erstellt eine 3x4 Matrix
2 m[1,1] = 3.0   # Setzt den Eintrag in Zeile 1 und Spalte 1 auf 3.0.
3 print(m)       # Druckt die Darstellung der Matrix aus.
```

7.5. Hashbare Objekte

Wir erklären hier, was hashbare Objekte sind und wie man selbstdefinierte Klassen hashbar macht.

Hashfunktionen und Hashtables wurden im Fortgeschrittenen Programmierkurs (siehe [ABH17]) bereits erklärt, deshalb halten wir uns hier kurz. Für einen festgelegten Typ `T` nennen wir Menge aller Objekte diesen Typs Set_T. Die Menge der **Python**-Ganzzahlen nennen wir hier Int. Man sagt dass

$$hash\colon Set_T \to Int$$

eine "zulässige Hashfunktion" ist, wenn für je zwei Objekte `x` und `y` aus Set_T für die `x == y` als `True` auswertet auch `hash(x) == hash(y)` als `True` auswertet. Desweiteren muss bei mutable Objekten gewähleistet sein, dass beim Ändern des Werts eines jeden Objekts `x` die Ausdücke `x == y` und `hash(x) == hash(y)` unabhängig von der Änderung sind. Man sagt, dass ein Objekt oder ein Typ `T` "hashbar" ist, wenn eine zulässige Hashfunktion für Set_T definiert ist. Der **Python**-Standard hält für die dort definierten immutable Typen (also Objekte vom Typ `int`, `float`, `complex`, `tuple`, `str`, `bool` und `none`, aber nicht `list` und `dict`) zulässige Hashfunktionen bereit. Demnach sind diese Typen hashbar.

Um eine selbstgeschriebene Klasse `K` hashbar zu machen, müssen wir die beiden spezielle Funktion `__eq__` und `__hash__` implementieren. Dabei muss (wir oben gefordert) für je zwei Instanzen `x` und `y` von `K` für die `x == y` als `True` auswertet, der Wert von `x.__hash__()` und `y.__hash__()` übereinstimmen. Außerdem verlangen wir, dass, falls das Objekt (durch Memberfunktionen) verändert werden kann, so soll `__eq__` und `__hash__` unabhängig von dieser Änderung sein.

Hier ein ganz simples Beispiel. Wir beschreiben eine Klasse mit dem Namen `Geldboerse`, inder man 1-Euro Münzen und 1-Cent Münzen speichern kann. Die Anzahl der Münzen soll nach dem Erstellen nicht mehr geändert werden. Zwei Geldbörsen sind genau dann gleich, wenn sie dieselbe Anzahl von 1-Euro und 1-Cent Münzen enthalten. Als Hashfunktion nehmen wir die Anzahl der 1-Euro Münzen modulo 100. Wir bemerken, dass das eine zulässige Hashfunktion ist. Der Code sieht dann so aus:

```
1  class Geldboerse:
2    """Eine sehr naive Klasse die eine Geldboerse darstellt."""
3    def __init__(self, euro, cent):
4      self._euro = euro
5      self._cent = cent
6
7    def __str__(self): # Macht Geldboerse als String interpretierbar
8      return 'Euro: {}, Cent: {}'.format(self._euro, self._cent)
9
10   def __eq__(self, other): # Implementiert x == y
11     if self._euro == other._euro and self._cent == other._cent:
12       return True
13     else:
14       return False
15
16   def __hash__(self): # Eine sehr naive Hashfunktion
17     return self._euro % 100
```

Wenn wir nun doch zulassen, dass man die Anzahl der Münzen geändert werden kann, so wäre unsere oben angegebene Hashfunktion nicht mehr zulässig. Es ist also gar nicht

so leicht, eine zulässige Hashfunktion für beliebige Klassen zu schreiben und in der Tat, man will meistens nur solche Objekte hashbar machen, die sich nach dem Erstellen nur unwesentlich oder garnicht mehr verändern.

8. Ausnahmen

Gut geschriebene Bibliotheken und Programme stürzen nicht ab, wenn sie falsch bedient werden oder wenn eine andere außergewöhnliche Situation eintritt. Beispielsweise soll unsere `joelixblas` Bibliothek nicht abstürzen, wenn der Benutzer eine Funktion mit unzulässigen Werten aufruft oder wenn der Benutzer mehr Speicher verlangt, als das System uns bieten kann.

In **Python** trennt man den "gewöhnlichen Programmfluss" und die "außergewöhnlichen Programmfluss" wie folgt: Wenn der normale, sequenzielle Programmfluss durch ein außergewöhnliches Vorkommnis unterbrochen werden muss, nennt man das eine *Ausnahme* oder *Exception*. Sobald eine Ausnahme *ausgelöst* wird, muss sie *behandelt* werden.

Typische Ausnahmen sind Fehler wie Syntaxfehler, Zugriffsfehler oder unzureichend viel Speicher. Es gibt in **Python** auch Ausnahmen, die nicht durch Fehler ausgelöst werden. Außerdem kann man eigene Ausnahmen definieren.

Wird eine Ausnahme nicht behandelt, bricht der Programmfluss der gerade laufenden Funktion ab und die Ausnahme wird an die aufrufende Funktion weitergegeben. Die Ausnahme wird dann entweder dort behandelt, oder das Weiterreichen wird fortgesetzt bis die Ausnahme entweder irgendwann behandelt wird oder das Programm mit der besagten Ausnahme abbricht.

Die folgende Ausnahme hat man bestimmt schon einige Male gesehen, wenn man Code direkt im **Python**-Interpreter schreibt:

```
1 a,b = 2,5
2 if a == b
3   print( "A ist ja wirklich B")
4
5 # Der Code liefert:
6 #   File "...", line ...
7 #     if a == b
8 #             ^
9 # SyntaxError: invalid syntax
```

Hier wird die zweite Zeile vom **Python**-Interpreter gelesen. Dieser stellt einen Syntaxfehler fest und löst eine Ausnahme aus, die den Programmfluss an dieser Stelle unterbricht. Schlussendlich wird das Programm beendet.

8.1. Ausnahmen behandeln

Bevor wir erklären, wie eine Ausnahme behandelt wird, gehen wir kurz darauf ein, was eine Ausnahme ist: Eine Ausnahme ist ein Objekt und somit hat sie einen festen Typ. Der **Python**-Standard definiert eine Hand voll Ausnahmetypen, wie zum Beispiel den Ausnahmetyp `ZeroDivisionError` (der bei einer Division durch Null ausgelöst wird) oder den Ausnahmetyp `SyntaxError` (den der **Python**-Interpreter bei einem Syntaxfehler auslöst). Der Wert einer Ausnahme `ausnahme` vom Ausnahmetyp `Ausnahmetyp` ist (für uns und nur hier) ein String und jede Ausnahme kann als String interpretiert werden. Eine Liste der wichtigsten, bereits definierten Ausnahmetypen sind in Abschnitt 8.3 zu finden.

Ein Programmabschnitt in dem Ausnahmen ausgelöst werden können, die wir (im Fall das Fälle) behandeln wollen, wird der Programmabschnitt in ein `try-except` Konstrukt eingefasst. Nach `try:` folgt unser (eingerückter) Programmabschnitt. Dann werden die möglichen Ausnahmen behandelt. Möchte oder muss man eine Ausnahme vom Ausnahmetyp `Ausnahmetyp` behandeln, geschieht das mit `except Ausnahmetyp:` gefolgt von der auszuführenden Ausnahmebehandlung. Hier kann man auch mehrere Ausnahmentypen zusammenfassen mit `except (Ausnahmetyp_1, Ausnahmetyp_2, ...):`. Alle anderen Ausnahmentypen sammelt man mit `except:`. Das sieht dann zum Beispiel so aus:

```
1  # Programmfluss
2
3  try:
4    # Abschnitt der Ausnahmen ausloesen kann, die wir behandeln wollen
5  except Ausnahmetyp_1:
6    # Ausnahmetyp 1 behandeln
7  except (Ausnahmetyp_2, Ausnahmetyp_3):
8    # Ausnahmetyp 2 behandeln
9  # Hier noch mehr Ausnahmentypen die man behandeln moechte
10 except:
11   # Alle anderen Ausnahmen behandeln
12
13 # Hier geht der normale Programmfluss weiter
```

Um auf eine Ausnahme von einem Ausnahmetyp zugreifen zu können nutzt man `except Ausnahmetyp as ausnahme`. Hier ein Beispiel, das die Ausnahme “Division durch Null” behandelt.

```
1 try:
2   a = 1/0
3 except ZeroDivisionError as ausnahme:
4   print('Ausnahme:', ausnahme)
```

Man kann auch mehrere Ausnahmetypen mit `as` benennen.

```
1 try:
2   a = 1/0
3 except (ZeroDivisionError, ValueError) as ausnahme:
4   print('Ausnahme␣vom␣Typ:', type(ausnahme), 'mit␣Wert:', ausnahme )
```

Es gibt Situationen, da möchte man einen Programmabschnitt ausführen der Ausnahmen auslösen kann und diese dann folgendermaßen behandeln. Wird eine (behandelnare) Ausnahme ausgelöst, so soll sie behandelt werden. Wird jedoch keine Ausnahme ausgelöst, dann (und nur dann) soll ein weiterer Programmabschnitt ausgeführt werden. Das funktioniert mit der `try-except-else` Konstruktion:

```
1 # Programmfluss
2
3 try:
4   # Abschnitt A
5 except ...: # Zu behandelnden Ausnahmetyp festlegen
6   # Ausnahmen behandeln
7 ... # Weitere Ausnahmebehandlungen
```

```
else: # Wird ausgefuehrt genau dann wenn Absch. A keine Ausnahme ausloest
  # Abschnitt B

# Hier geht der normale Programmfluss weiter
```

Der obige Code verhält sich genau wie der nachfolgende Code:

```
try:
  ausnahme_aufgetreten = False
  # Abschnitt A
except ...: # Zu behandelnden Ausnahmetyp festlegen
  ausnahme_aufgetreten = True
  # Ausnahmen behandeln
... # Weitere Ausnahmebehandlungen, die ausnahme_aufgetreten=True setzen
if ausnahme_aufgetreten == False:
  # Abschnitt B
```

An dieser Stelle kann man bereits verstehen, warum Ausnahmen ein gutes Konzept sind. Durch die Aufteilung in einen von `try` eingeleiteten Block schreibt man den auszuführenden Programmcode und teilt die Ausnahmebehandlung in die von `except` eingeleiteten Blöcken ein. Das führt zu wesentlich übersichtlicherem Code.

Hier noch ein Beispiel:

```
import math
def ganzzahlige_wurzel( x ):
  """Diese Funktion zieht die ganzzahlige Wurzel."""
  y = 0 # Wir definieren eine Variable y, die wir am Ende zurueckgeben,
        # unabhaengig davon, ob eine Ausnahme behandelt werden muss oder
        # nicht
  try:  # Versuche die ganzzahlige Wurzel zu siehen
    y = int(math.sqrt(x))
  except TypeError as ausnahme:  # Ausnahme: x hat den falschen Typ
    print('Falscher Typ:', ausnahme)
  except ValueError as ausnahme: # Ausnahme: x ist negativ.
    print('Falscher Wert:', ausnahme)
  except:                        # Ausnahme: andere Ausnahme
    print('Anderer, komischer Fehler...')
  return y

ganzzahlige_wurzel('Suppe') # Druckt: 'Falscher Typ: a float is required'
ganzzahlige_wurzel(-3)      # Druckt: 'Falscher Wert: math domain error'
ganzzahlige_wurzel(6)       # Druckt nix.
```

Man beachte, dass die Zeile `y=0` nicht vergessen werden darf, denn sonst kann es passieren, dass `y` in Zeile 15 nicht definiert ist. Falls `math.sqrt(x)` eine Ausnahme auslöst, wird der Programmfluss in Zeile 8 unterbrochen und die Ausnahme behandelt. Dass heißt, beim Auslösen einer Ausnahme wird `y` in dieser Zeile weder definiert noch auf ein Objekt gesetzt und kann insbesondere in Zeile 15 nicht zurückgegeben werden.

8.2. Ausnahmen auslösen und weitergeben

Wir wollen nun verstehen, wie man Ausnahmen auslöst und wie Ausnahmen weitergegeben werden. Beides geschieht mit `raise`.

Man löst eine Ausnahme vom Typ `Ausnahmetyp` mit dem beschreibenden String `ausnahmestring` durch folgendes Statement aus:

```
raise Ausnahmetyp(ausnahmestring)
```

Wenn man sein Programm sehr trotzig abbrechen möchte kann man das also so tun:

```
raise RuntimeError("Mir ist jetzt alles egal!")
```

Im folgenden Beispiel definieren wir eine Funktion, die nur mit Strings und Ganzzahlen umgehen möchte:

```
def ich_mag_nur_strings_und_ganzzahlen( x ):
  """Diese Funktion mag nur Strings und Ganzzahlen."""
  if not (type(x) is int or type(x) is str):
    raise ValueError("Ich mag nur Strings und Ganzzahlen")
  print("Ich mag dich: '{}'".format(x))
```

Nun klären wir die Frage:

Wem wird eine Ausnahme zum Behandeln eigentlich übergeben?

Zuerst führen wir den sogenannten *Call Tree* eines Programms ein. Der Call Tree ist bei sequenziellen Programmen das ohne ausgelöse Ausnahmen auskommt immer ein gewurzelter Baum. Die Wurzel v ist die main-Funktion (oder genauer das main-Modul). Wird eine Funktion `f` aufgerufen, definiert das eine Kante mit einem neuen Knoten in unserem Baum, den wir hier der Einfachheit halber $v(f)$ nennen. Die in `f` aufgerufenen Funktionen, sagen wir `g`, `h` oder vielleicht sogar `f`, definieren dann neue Kanten zu neuen Knoten, sagen wir $v(g, f)$, $v(h, f)$, $v(f, f)$. Wird eine Funktion `k` mehrere Male hintereinander aufgerufen, erstellen wir für jeden Aufruf eine neue Kante mit einem neuen Endknoten.

Per Konstruktion entspricht jede Kante einem Funktionsaufruf. Braucht man k Kanten um von der Wurzel v zu einem anderen Knoten w zu kommen, bedeutet dass wir k ineinander verschachtelte Funktionsaufrufe benötigt haben.

Also entsteht zur Programmlaufzeit ein gewurzelter Baum. Zu einem festen gewählen Zeitpunkt, während das Programm läuft, gibt es immer einen Knoten, der zuletzt erstellt wurde. Diesen Knoten nennen wir "aktiv". Der aktive Knoten entspricht dem Programmabschnitt, in dem wir uns (zur festgelegten Laufzeit) befinden.

Nun können wir leicht verstehen, wem eine Ausnahme zum Behandeln übergeben wird. Dazu nehmen wir uns den Call Tree das laufende Programm zur Hilfe. Wird eine Ausnahme in einem Programmabschnitt ausgelöst, so entspricht dieser Programmabschnitt dem aktiven Knoten. Die Ausnahme kann dann vom aktiven Knoten behandelt werden, falls der Programmabschnitt im einem `try-except` Konstrukt liegt. Außerdem muss die Ausnahme in ihrem `except` Abschnitt behandelt werden. Ist mindestens eins von beiden nicht der Fall, wird die Ausnahme an den Knoten über dem aktiven Knoten

zum Behandeln weitergeben. Dies wird so lange fortgeführt, bis die Ausnahme behandelt wurde oder bis sie schlussendlich auch in der Wurzel nicht behandelt wurde. Im letzteren Fall erklärt der **Python**-Interpreter wo die Ausnahme aufgetreten ist und beendet das Programm.

Man kann sogar Ausnahmen behandeln und nach der Behanlung die Ausnahme mit `raise` an den darüberliegenden Knoten weitergeben. Das besprechen wir an dieser Stelle aber nicht ausführlicher.

Schauen wir uns das ganze mal anhand eines Beispiels an:

```
1  def drucke_float_aus(zahl): # Druckt float aus oder loest Ausnahme aus
2    if type(zahl) is not float:
3      raise TypeError('Ich will float und sonst nichts.')
4    else:
5      print('{}, ich liebe dich'.format(zahl))
6
7  def eins_durch_null(): # Teilt durch Null
8    return (1.0/0.0)
9
10 def behandle_ausnahmen_nicht(): # Behandelt Ausnahmen nicht
11   eins_durch_null()              # Ausnahme wird nicht behandelt
12
13 def behandle_ausnahmen():
14   try:
15     drucke_float_aus('Hallo!')
16   except:
17     print('Ausnahme an Stelle 1 wurde ausgeloest')
18
19   try:
20     drucke_float_aus(eins_durch_null())
21   except:
22     print('Ausnahme an Stelle 2 wurde ausgeloest')
23
24   try:
25     behandle_ausnahmen_nicht()
26   except:
27     print('Ausnahme an Stelle 3 wurde ausgeloest')
```

Beim Aufruf der Funktion `behandle_ausnahmen()` versuchen wir zunächst, die Funktion `drucke_float_aus('Hallo!')` auszuführen. Diese löst eine Ausnahme aus, die wir in `behandle_ausnahmen()` abfangen. In der Ausnahmebehandlung drucken wir:

`'Ausnahme an Stelle 1 wurde ausgeloest'`

Nun versuchen wir `drucke_float_aus(eins_durch_null())` auszuführen. Dabei wird zuerst die innere Funktion, also `eins_durch_null()` ausgeführt. Diese löst eine Ausnahme aus. Insbesondere gibt die Funktion `eins_durch_null()` nichts zurück denn die normale Ausführung wird unterbrochen und wir machen sofort mit der Fehlerbehandlung weiter. In der Ausnahmebehandlung drucken wir:

`'Ausnahme an Stelle 2 wurde ausgeloest'`

Nun versuchen wir `behandle_ausnahmen_nicht()` auszuführen. Diese Funktion ruft `eins_durch_null()` aus. In unserem Call Tree haben wir momentan also einen Weg von `behandle_ausnahmen` über `behandle_ausnahmen_nicht` zu `eins_durch_null`. Die Funktion `eins_durch_null` löst eine Ausnahme aus. Diese wird an `behandle_ausnahmen_nicht` weitergegeben und dort nicht behandelt. Also wird sie weitergegeben an `behandle_ausnahmen`. Dort wird sie behandelt. In der Ausnahmebehandlung drucken wir:

`'Ausnahme an Stelle 3 wurde ausgeloest'`

8.3. Bereits definierte und eigens definierte Ausnahmen

Wir listen nachfolgend einige Ausnahmentypen auf. Diese Liste ist nicht vollständig und wir verweisen den interessierten Leser auf [Pytc, Library: Exceptions]. Außerdem sind einige der Ausnahmen voneinander abgeleitet. Da wir in diesem Kurs "abgeleitete Klassen" nicht behandelt haben, gehen wir hier weiter nicht darauf ein und verweisen nocheinmal auf [Pytc, Library: Exceptions].

```
1  Exception          # Allgemeine Ausnahme.
2
3  FloatingPointError # Gleitkommafehler
4  OverflowError      # Overflowfehler
5  ZeroDivisionError  # Du hast durch Null geteilt
6  ImportError        # Fehler beim Importieren
7  IndexError         # Falscher Index beim Sequenzzugriff
8  KeyError           # Falscher Key beim Verzeichniszugriff
9  MemoryError        # Wir haben nicht genug Speicher
10 FileExistsError    # Datei existiert (beim erstellen einer neuen Datei)
11 FileNotFoundError  # Datei nicht gefunden (beim oeffnen einer Datei)
12 IsADirectoryError  # Dateioperation auf Ordner angewendet
13 NotADirectoryError # Ordneroperation auf Datei angewendet
14 PermissionError    # Unzureichende Zugriffsrechte (bei Dateien / Ordnern)
15 RuntimeError       # Laufzeitfehler (wird vom Programmierer ausgeloest)
16 NotImplementedError # Funktion ist nicht implementiert
17 SyntaxError        # Syntaxfehler
18 IndentationError   # Syntaxfehler: Falsch eingerueckt
19 SystemError        # Komischer Systemfehler
20 TypeError          # Falscher Typ
21 ValueError         # Falscher Wert
22
23
24 Warning            # Allgemeine Warnung
25
26 DeprecationWarning # Warnung: Veraltete Funktion / Klasse wird verwendet
27 ImportWarning      # Warnung beim Importieren
```

Da wir in diesem Kurs "abgeleitete Klassen" nicht behandelt haben, erklären wir hier nur, wie man eigene Ausnahmen erstellt, aber nicht wie das im Detail funktioniert. Eine eigenen Ausnahmetyp erstellt man so:

```
1 class meine_ausnahme(Exception): pass
```

Dann kann man im `try-except` Konstrukt seinen eigenen Ausnahmetyp verwenden. Hier ein Beispiel:

```
import math

class NichtQuadratisch(Exception) : pass
class KeineReelenLoesungen(Exception) : pass

def loese_quad_gl(a,b,c):
  if a == 0:
    ausnbeschr = 'Nicht quadratisch a={}, b={}, c={}'.format(a,b,c)
    ausnahme = NichtQuadratisch(ausnbeschr)
    raise ausnahme
  if b**2 - 4.0*a*c < 0:
    ausnbeschr = 'Keine reelen Loesungen a={}, b={}, c={}'.format(a,b,c)
    ausnahme = KeineReelenLoesungen(ausnbeschr)
    raise ausnahme
  x1 = ( -b + math.sqrt( b**2 - 4.0*a*c) ) / (2.0*a)
  x2 = ( -b - math.sqrt( b**2 - 4.0*a*c) ) / (2.0*a)
  return x1, x2

try:
  loese_quad_gl(1,0,-1) # (1,0), (-1,0)
  loese_quad_gl(1,0,1)  # Ausnahme: Keine reelen Loesungen a=1, b=0, c=1
  loese_quad_gl(0,0,1)  # Ausnahme: Nicht quadratisch a=0, b=0, c=0
except (NichtQuadratisch, KeineReelenLoesungen) as ausnahme:
  print('Meine Ausnahme', ausnahme)
```

Als letztes erklären wir in diesem Abschnitt wie man in einem `try-except` Konstrukt die "restlichen Ausnahmentypen", also solche die mit `except:` behandelt werden, auch mit `as` benennen kann. Warum das funktioniet erklären wir an dieser Stelle nicht, da man "abgeleitete Klassen" für die Erklärung braucht, die haben wir in diesem Kurs aber nicht behandeln können. Um also "die restlichen Ausnahmen" mit `as` zu benennen verwendet man die Zeile `except Exception as ausnahme:`, auch wenn `Exception` nicht alle sondern nur alle sinnvollen Ausnahmen zusammenfasst. Zum Beispiel ist die Ausnahme `SystemExit` nicht vom Typ `Exception`. Die Ausnahmebehandlung soll so aussehen:

```
try:
  # Programmabschnitt der Ausnahmen ausloesen kann
except Ausnahmetyp_1 as ausnahme:
  # Ausnahmetyp 1 behandeln
...
except Exception as ausnahme:
  # Alle anderen sinnvollen Ausnahmen behandeln
except:
  raise # Lass das mal lieber den Python Interpreter handhaben.
# Hier geht der normale Programmfluss weiter
```

9. Module

In **Python** organisiert man seinen Code (also Klassen, Funktionen, Konstanten, usw.) in sogenannten Modulen. Ein Modul ist entweder eine einzelne Datei oder ein Ordner (ggf. mit weiteren Unterordnern) gefüllt mit **Python** Code.

Bekammt der Interpreter eine Skriptdatei `hauptprogramm.py`, so wird diese interpretiert. Dabei wird Zeile für Zeile gelesen, geprüft und ausführt. Es werden also Klassen und Funktionen definiert, sowie Objekte erstellt und Funktionen aufgerufen. Das interpretierte Skript, sagen wir `hauptprogramm.py`, wird bereits als Modul verstanden. Es ist das sogenannte main-Modul.

9.1. Module einbinden

Python ist so mächtig und empfehlenswert, weil es eine riesige Auswahl an guten Modulen gibt, die nur noch auf ihre Nutzung warten. Module werden mit dem Befehl `import` eingebunden.

```
import modulname
```

Beim Einbinden eines Modul `modul` wird `modul` Zeile für Zeile gelesen, geprüft und ausgeführt (also genauso wie beim main-Modul). Wird ein Modul eingebunden, bekommt es Attribute, zum Beispiel wird der Name des Moduls in der Variable `__name__` gespeichert. Um das besser zu verstehen, schauen wir uns folgendes Beispiel an. Beginnen wir mit der Datei `hilfsmodul.py`, die so aussehen könnte.

```
if __name__ == '__main__': # Pruefe, ob das Modul das main-Modul ist
  print('Ich bin das main-Modul.')
else:
  print('Ich bin als Modul mit Namen "' + __name__ + '" geladen worden.')
```

Python3 interpretiert `hilfsmodul.py` dann so:

```
python3 hilfsmodul.py
  # Ich bin das main-Modul.
```

Nun erstellen wir (im selben Ordner) noch ein **Python** Skript, mit Namen `hauptprogramm.py`, das so aussieht.

```
import hilfsmodul

if __name__ == '__main__': # Pruefe, ob das Modul das main-Modul ist
  print('Ich bin hier der Boss.')
```

Schauen wir uns jetzt an, wie **Python** dieses Skript interpretiert. Als erstes wird das Modul `hilfsmodul` geladen, welches bei uns durch die Datei `hilfsmodul.py` repräsentiert wird. Beim Laden des Moduls `hilfsmodul` wird dieses Zeile für Zeile gelesen, geprüft und ausgeführt. Da es nicht das main-Modul ist, trägt es einen anderen Namen und dieser wird dann auch ausgedruckt. Nachdem das Modul `hilfsmodul` fertig eingebunden wurde, geht es weiter mit dem Skript `hauptprogramm.py`. Dieses ist das main-Modul,

also wird noch 'Ich␣bin␣hier␣der␣Boss.' ausgedruckt. Der gesamte Output sieht also so aus:

```
python3 hauptprogramm.py
  # Ich bin als Modul mit Namen "hilfsmodul" geladen worden.
  # Ich bin hier der Boss.
```

Haben wir ein Modul `hilfsmodul` eingebunden, können wir auf die darin definierten Klassen, Funktionen und Variablen mit dem Präfix `hilfsmodul.` zugreifen. Wurde in `hilfsmodul` beispielsweise eine Funktion `funktion_eins` definiert, greifen wir auf diese im main-Modul mit `hilfsmodul.funktion_eins` zu. Hier ein vollständiges Beispiel.

```
import math             # Importiere das Modul math

s = math.sin(3.141/5.0) # Berechne den Sinus von 3.141/5.0
print(s)                # Drucke den Sinus von 3.141/5.0 aus.
```

Es kommt manchmal vor, dass man auf den Inhalts eines Moduls nicht über seinen gegebenen Namen sondern einen frei gewählten Namen zugreifen möchte. Das kommt zum Beispiel vor, wenn es zwei Versionen einer Bibliothek benutzen möchte (eine ist optimiert, die andere produziert viele Debuginformationen). In diesem Fall will man nur ganz zu Anfang des Programs festlegen welche Bibliothek eingebunden werden soll ohne weitere Zeilen im Code zu ändern. In **Python** funktioniert das so:

```
import modulname as neuer_modulname
```

Nun kann man auf die Klassen, Funktionen und so weiter von `modulname` mit dem Präfix `neuer_modulname.` zugreifen. Schauen wir uns das ganze mal in einem Beispiel an. In **Python** gibt es das Modul `math` und das Modul `cmath`. Das erste bietet die mathematischen Größen und Funktionen an, die im **C**-Standard definiert sind; das zweite bietet analoge mathematische Funktionen an, die auch für komplexe Zahlen definiert sind.

```
import random # Binde random ein
nimm_cmath = random.choice( (True, False) ) # Waehlt True oder False

if nimm_cmath:  # Pruefe, ob das wir cmath importieren wollen
  import cmath as m # Binde cmath ein und nenne es m.
else:
  import math as m  # Binde  math ein und nenne es m.

print( m.sqrt(-1) ) # Druckt 1j wenn cmath eingebunden wurde
                    # und bricht sonst mit einer Ausnahme ab.
```

Manchmal will oder muss man eine Hand voll Klassen, Objekte oder Funktionen aus einem Modul `hilfsmodul` in sein Programm einbinden und gleichzeitig auf das Präfix `hilfsmodul.` verzichten. Oft passiert das beim erstellen eines eigenen Moduls (was wir im nächsten Abschnitt behandeln). Um also eine endliche Menge von Objekten `obj_1, ..., obj_k` aus einem Modul `hilfsmodul` direkt einzubinden, nutzen wir den folgenden Code.

```
from hilfsmodul import obj_1, ..., obj_k
```

Ähnlich wie eingebundene Module, kann man auch eingebundene Objekte mit `as` umbennen:

```
from hilfsmodul import obj_1 as neuer_name_1, ..., obj_k as neuer_name_k
```

Schauen wir uns das an einem simplen Beispiel an:

```
from math import sin as Sinus, cos as Cosinus, pi as PI

x = Sinus(PI/3.0)
y = Cosinus(PI/3.0)
print( x, y ) # 0.8660254037844386 0.7071067811865476
```

9.2. Eigene Module in Python bereitstellen

Um in **Python** Bibliotheken zur Verfügung zu stellen, packt man (ähnlich wie in **C/C++**) seinen gesamten Code in ein Modul. Das kann zwar ein einziges **Python**-Skript sein, aber so gut wie immer macht es viel mehr Sinn, seinen Code in mehreren Dateien (die auf mehrere Ordnern verteilt sind) zu organisieren. Es ist gängige Praxis seine logischen Einheiten in einem gewurzelten Baum zu organisieren. Jedes Blatt entspricht einer Datei und alle anderen Knoten sind Ordner, welche die darunter liegenden Knoten (also Ordner und Dateien) enthalten. Außerdem ist es oft klug einen gewissen Anteil des Codes dem Benutzer nicht direkt zugänglich machen. So kann man als Anbieter der Bibliothek kleine Details ändern, ohne dass sich der Benutzer der Bibliothek an neue Funktionen gewöhnen muss.

Wir besprechen anhand eines Beispiel die einfachsten Möglichkeiten ein eigenes Modul zu erstellen. Es sei hier noch gesagt, dass Module auch Untermodule bereitstellen können.

9.2.1. Skripte als Module einbinden

Im aller einfachsten Fall ist ein Modul nichts anderes als eine **Python**-Skript. Das haben wir im vergangenen Abschnitt bereits gesehen. Nehmen wir an, wir haben bereits eine **Python**-Skript `prog.py` in einem Order `wurzel` erstellt und mit sinnvollem Code gefüllt. Wenn wir nun ein weiteres **Python**-Skript `hauptprog.py` im selben Ordner (also `wurzel`) anlegen, können wir `prog.py` mithilfe von `import` laden. Beim Importieren wird die Dateiendung weggelassen:

```
# Datei: hauptprog.py
import prog # Importiert prog.py
...
```

Beim importieren des **Python**-Skripts `prog.py` wird dieses Zeile für Zeile gelesen, geprüft und ausgeführt. Danach kann man auf die Funktionen, Klasses und Variablen mithilfe des Präfixes `prog.` zugreifen. Also greifen wir auf die Variable `x` die in `prog.py` definiert wurde mit `prog.x` zu.

9.2.2. Ordner als Module einbinden

Im nächst einfachen Fall ist ein Modul nichts anderes als ein Order gefüllt mit **Python**-Skripten. Gehen wir davon aus, dass wir einen Ordner `skript_bib` haben, in dem die **Python**-Skripte `skript_eins.py`, `skript_zwei.py` und `skript_drei.py` liegen haben. Damit der **Python**-Interpreter den Ordner `skript_bib` als Modul interpretieren kann, muss der Ordner ein **Python**-Skript mit dem Namen `__init__.py` enthalten. Dieses Skript wird beim Einbinden des Moduls Zeile für Zeile gelesen, geprüft und ausgeführt. Alle anderen Skripte werden nicht automatisch interpretiert, denn das ist überlicherweise die Aufgabe von `__init__.py`. In `__init__.py` bindet man dann entweder alle Klassen, Funktionen und Variablen der anderen Skripte ein. Das geschieht mithilfe von

```
1 from skript_eins.py import *
```

oder man bindet nur die benötigen Klassen, Funktionen und Variablen der anderen Skripte ein. Das geschieht mithilfe von

```
1 from skript_zwei.py import klasse_a, klasse_b, funktion_z
```

Sobald das Modul `skript_bib` importiert ist, kann man auf alle Klassen, Funktionen und Variablen, die dem Skript `__init__.py` bekannt sind, mithilfe des Präfixes `skript_bib.` zugreifen.

Wir besprechen das nun anhand eines Beispiels. Angenommen wir wollen ein Modul `JoelixBlas` anfertigen, dass sich wie die Bibliothek `joelixblas` verhält (siehe [ABH17]). Das heißt, wir wollen ein Modul, das Vektoren und (dünnbesetzte Matrizen) anbietet und elementare Operationen (wie zum Beispiel Matrix-Vektor-Multiplikation) bereitstellt. Zunächst erstellen wir ein Verzeichnis mit dem Namen unseres Moduls, also `JoelixBlas`. Die logischen Einheiten `JoelixVektor` und `JoelixMatrix` sind zu gewissen Teilen voneinander unabhängig und wir entscheiden uns dazu, sie in zwei separate Dateien zu lagern. Unsere Verzeichnisstruktur sollte zu diesem Zeitpunkt so aussehen:

```
1 JoelixBlas
2   +--joelix_vektor.py
3   +--joelix_matrix.py
```

Nun implementieren wir die Klassen `JoelixVektor` und `JoelixMatrix`. Um `JoelixVektor` zu implementieren, brauchen wir die Klasse `JoelixMatrix` nicht zu kennen. Die Datei `joelix_vektor.py` sieht so aus:

```
1 class JoelixVektor:
2   """Vektorklasse der JoelixBlas."""
3   ...
4   def __add__(self, other): # Implementiert Vektor-Vektor-Addition
5     """Implementiert Vektor-Vektor-Addition."""
6     if type(other) is type(self):
7       # Vektor-Vektor-Addition
8   ...
```

Die Matrix-Matrix-Multiplikation und die Matrix-Vektor-Multiplikation soll durch eine spezielle Memberfunktion von `JoelixMatrix` realisiert werden. Da wir die Klasse

`JoelixVektor` für die Matrix-Vektor-Multiplikation benötigen, binden wir `joelix_vektor` in `joelix_matrix` ein: Die Datei `joelix_matrix.py` sieht so aus:

```
# JoelixMatrix.py
from joelix_vektor import JoelixVektor

class JoelixMatrix:
  """Duennbesetzte Matrix der JoelixBlas."""
  ...
  def __mul__(self, other): # Definiere Matrix-X-Multiplikation
    """Implementiert die Matrix-X-Multiplikation."""
    if type(other) is type(self):
      # Matrix-Matrix-Multiplikation
    elif type(other) is JoelixVektor:
      # Matrix-Vektor-Multiplikation
  ...
```

Damit wir den Ordner `JoelixBlas` als Modul einbinden können, braucht der Ordner `JoelixBlas` noch das **Python**-Skript `__init__.py`. In diesem importieren wir die Klasse `JoelixVektor` aus `joelix_vektor.py` und die Klasse `JoelixMatrix` aus der Datei `joelix_matrix.py`.

```
# Datei: __init__.py
from joelix_vektor import JoelixVektor
# Jetzt kennt __init__.py die Klasse JoelixVektor
from joelix_matrix import JoelixMatrix
# Jetzt kennt __init__.py die Klasse JoelixMatrix
```

Unsere Verzeichnisstruktur sollte also so aussehen:

```
JoelixBlas
  +--__init__.py
  +--joelix_vektor.py
  +--joelix_matrix.py
```

Jetzt können wir das Modul `JoelixBlax` in einem **Python**-Programm verwenden. Zum Beispiel erstellen wir ein **Python**-Skript mit dem Namen `test.py` das imselben Order liegt, wie der Ordner `JoelixBlas`. Unsere Verzeichnisstruktur sollte also so aussehen:

```
mein_programm
  +--test.py
  +--JoelixBlas
        +--__init__.py
        +--joelix_vektor.py
        +--joelix_matrix.py
```

Das **Python**-Skript `test.py` sieht so aus:

```
import JoelixBlas

def main():
  Vek = JoelixBlas.JoelixVektor(dim=3)
  Vek[0] = 5.0
```

```
6     Vek[1] = 4.6
7     Vek[2] = 1.8
8     print( Vek )
9
10 if __name__ == '__main__':
11     main()
```

9.3. Empfohlene Module

In diesem Abschnitt beschreiben wir eine Hand voll Module, die wir für besonders wichtig halten. Um dieses Skript nicht unnötig lang werden zu lassen, besprechen wir die Module nur ganz oberflächlich und verweisen die interessierte Leserin auf die offizielle Dokumentation der hier besprochenen Module. Die Module in den Abschnitten 9.3.1, 9.3.2, 9.3.3, 9.3.4, 9.3.5, 9.3.6, 9.3.7, 9.3.8, 9.3.9, 9.3.10, 9.3.11, 9.3.12, und 9.3.13 sind (in dieser Reihenfolge) durch den **Python**-Standard beschrieben. Im Abschnitt 9.3.14 empfehlen wir noch einige Module, die für Mathematiker interessant sein dürften.

9.3.1. re

Ein "regulärer Ausdruck" ist ein String (mit vielen Sonderzeichen) mitdem eine Menge von gültigen Strings beschrieben wird. Zum Beispiel ist die Menge der gültigen String zum reguläre Ausdruck `*.py`, die Menge der Strings die mit `.py` enden und einen beliebigen (eventuell leeren) Wortanfang besitzen. Eigentlich verdienen reguläre Ausdrücke ein eigenes Kapitel, das würde dieses Skript aber inhaltlich sprengen. Wir machen hier nur darauf aufmerksam, dass das Modul `re` viele Methoden zum Arbeiten mit regulären Ausdrücken bereitstellt. Wenn man mit regulären Ausdrücken vertraut ist, ließt man in [Pytc, Library, Text Processing Services, Regular expression operations] wie das Modul `re` zu benutzen ist. Sonst verweisen wir den Leser auf [Lot10, Kapitel 33].

9.3.2. copy

Wir haben in Abschnitt 4.3.2 gesehen, dass Kopien von Sequenzen immer oberflächeliche Kopien sind. Das ist zwar schnell, kann aber zu ungewollten Effekten führen wenn man nicht vorsichtig ist. Um echte Kopien zu erstellen, benutzt man das Modul `copy`. Die darin implementierte Funktion `copy.deepcopy` erstellt sogenannte tiefe Kopien.

9.3.3. math

Das Modul `math` stellt die mathematischen Funktionen und Konstanten zur Verfügung, die der **C**-Standard beschreibt. Zum Beispiel zieht man die Wurzel mit `math.sqrt`.

```
1 import math
2 math.sqrt(4.0)
```

9.3.4. cmath

Das Modul `cmath` beschreibt dieselben mathematischen Funktionen und Konstanten, allerdings ist der Definitions- und Wertebereich `complex` statt `float`. Zum Beispiel zieht man die komplexe Wurzel mit `cmath.sqrt`.

```
1 import cmath
2 cmath.sqrt(-1.0)
```

9.3.5. random

Das Modul `random` implementiert einfache Pseudozufallszahlengeneratoren für verschiedene Verteilungen. Um aus einer Sequenz ein Element (bezüglich der Gleichverteilung) zufällig auszuwählen, nutzt man die Funktion `random.choice`. Beispielsweise kann man einen Buchstaben aus dem String `"Hallo␣Welt"` wie folgt zufällig auswählen:

```
1 import random
2 random.choice( "Hallo␣Welt" )
```

9.3.6. itertools

Um aus Sequenzen (oder allgemeiner "iterierbaren Objekten") neue iterierbare Objekte zu erstellen, benutzt man das Modul `itertools`. Dieses Modul ist nicht nur speicher- und laufzeiteffizient sondern auch sehr mächtig, wenn man es richtig einzusetzen weiß. An dieser Stelle sei gesagt, dass die Funktionen von `itertools` bei gegebenem Input keine Sequenzen erstellen (die dann anschließend Element für Element ausgelesen werden können), sondern sie erstellen sogenannte "iterierbares Objekte", die den aktuell zu verarbeiteten Eintrag zur Laufzeit generieren. Dass spart bei großem Input wertvollen Speicher und ist meistens schneller. Wir stellen hier nur zwei sehr einfache Funktionen vor und legen der interessierten Leserin [Pytc, Library, Funktional Programming Tools, itertools] an Herz.

Um aus zwei Sequenzen `A` und `B` eine Sequenz `((x,y) for x in A for y in B)` zu erstellen, reicht der angegebene Ausdruck, allerdings ist die Funktion `itertools.product` schneller und speichereffizienter. Hier ein Beispiel:

```
1 import itertools
2 A = 'Hallo␣Welt'
3 B = ( 1.0, 4.5, 'sss' )
4 C = itertools.product(A,B) # Aequivalent zu ((x,y) for x in A for y in B)
```

Außerdem kann man `product` eine beliebige Anzahl von Sequenzen übergeben.

Um (eine Sequenz) alle(r) Permutationen einer gegebenen Sequenz `s` zu erhalten, nimmt man die Funktion `itertools.permutations`. Beispielsweise erhält man alle Permutationen des String `'Valhalla'` wie folgt:

```
1 import itertools
2 p = itertools.permutations('Valhalla')
3 for x in p:
```

```
4   print(x)   # Druckt:
               # ('V', 'a', 'l', 'h', 'a', 'l', 'l', 'a')
               # ('V', 'a', 'l', 'h', 'a', 'l', 'a', 'l')
               # ('V', 'a', 'l', 'h', 'a', 'l', 'l', 'a')
               # ...
```

Will man (eine Sequenz) alle(r) Permutationen von `s` der Länge `r`, haben, übergibt man `permutations` noch den Parameter `r`:

```
import itertools
p = itertools.permutations('Valhalla', 2)
for x in p:
  print(x)   # Druckt:
             # ('V', 'a')
             # ('V', 'l')
             # ('V', 'h')
             # ...
```

In beiden Fällen ist es wichtig zu wissen, dass man die gesamte Symmetrische Gruppe auf (einem variierenten Teil der Länge r) der Sequenz operieren lässt. Insbesondere besteht `itertools.permutations(s)` aus $l!$ vielen Elementen, wenn `s` eine Sequenz der Länge l ist, selbst wenn manche Werte in `s` mehrfach vorkommen.

9.3.7. pickle

Um Objekte in **Python** auf der Festplatte zu speichern und von der Festplatte zu laden, kann man das Modul `pickle` benutzen. Das Speichern und Laden ist plattformunabhängig, solange die gleiche **Python**-Version verwendet wird, das heißt, man kann mit `pickle` gespeicherte Daten zwischen den Betriebssystemen GNU/Linux, Mac und Windows hin und hertauschen, solange man dieselbe **Python**-Version verwendet.

Um mehrere Objekte in einer Datei zu speichern, muss zuerst das Modul `pickle` importiert werden. Nun muss man eine Datei `datei` zum Schreiben im "Bytemodus" geöffnet werden. Dies geschieht durch den Ausdruck `datei = open('pfad', 'wb')`. Im Anschluss kann man mit `pickle.dump(obj, datei)` ein Objekt nach dem anderen in die zuvor geöffnete Datei `datei` schreiben. Im folgenden Beispiel speichern wir die Zahlen `0, ..., 9`:

```
import pickle                   # Importiert pickle
datei = open( 'test', 'wb' )    # Oeffnet Datei 'test' im Bytemodus
for i in range(10):             # Fuer 0 <= i < 10:
  pickle.dump(i, datei)         # Schreibe i in datei
datei.close()                   # Datei schliessen
```

Um mehrere Objekte aus einer Datei zu lesen, muss ebenfalls zuerst das Modul `pickle` importiert werden. Nun muss man eine Datei `datei` zum Lesen im "Bytemodus" geöffnet werden. Dies geschieht durch den Ausdruck `datei = open('pfad', 'rb')`. Im Anschluss kann man mit `pickle.load(datei)` ein Objekt nach dem anderen aus der zuvor geöffneten Datei `datei` lesen. Das Lesen der zuvor gespeicherten Objekte funktioniert nach dem "FIFO"-Prinzip, d.h. Daten die zuerst geschrieben wurde, werden zuerst gelesen. Der Funktionsaufruf `pickle.load(datei)` gibt das gespeicherte Objekt zurück

oder löst eine Ausnahme vom Typ `EOFError` aus sofern die Datei keine weiteren Objekte enthält. Im folgen Beispiel lesen wir alle Daten aus einer Datei aus.

```
1  import pickle                    # Importiert pickle
2  datei = open( 'test', 'rb' )    # Oeffnet Datei 'test' im Bytemodus
3  dateiende_erreicht = False      # Soll True sein wenn Datei am Ende ist
4  objekte = []                    # Liste der gelesenen Objekte
5  while dateiende_erreicht == False: # Solange die Datei nicht am Ende ist
6     try:                         # Versuche
7       obj = pickle.load(datei) # Ein Objekt zu lesen
8       objekte.append(obj)      # Das gelesene Objekt zur Liste hinzufuegen
9     except EOFError:             # Fange EOFError ab (= Dateiende erreicht)
10      dateiende_erreicht = True # Setze dateiende_erreicht auf True
11 datei.close()                   # Datei schliessen
12 print(objekte)                  # Drucke Liste der Objekte aus
```

9.3.8. os

Das Modul `os` ermöglicht es uns betriebssystemespeziefische Aufgaben plattformunabhängig zu lösen. Beispielsweise müssen Unterordner im Dateipfad unter Windows mit `'\'` und unter GNU/Linux und Mac mit `'/'` kennzeichnen. Als **Python**-Programmererin hat man an solchen lästigen Fallunterscheidungen kein Interesse. Mit `os.mkdir('pfad')` erstellt man neue Ordner. Um zum Beispiel zu testen, ob ein Dateipfad einen Ordner oder eine normale Datei benennt, kann man folgende Funktionen verwenden.

```
1 import os
2 os.path.isfile('pfad') # True gdw 'pfad' eine normale Datei ist
3 os.path.isdir('pfad')  # True gdw 'pfad' ein Ordner ist
```

9.3.9. time

Wie der Name des Moduls `time` vermuten lässt, benutzt man es um die Zeit zu messen oder eine Pause einzulegen. Um einen Prozess für `k` Sekunden schlafen zu legen, nutzt man die Funktion `time.sleep(k)`. Um die Echtzeit zu messen, benutzt man die Funktion `time.perf_counter`. Die CPU-Zeit eines Prozesses misst man mit `time.process_time`. Hier ein Beispiel:

```
1 import time
2 echtzeit_start = time.perf_counter()
3 cpuzeit_start  = time.process_time()
4 # Hier eine aufwaendige Berechnung einsetzen, welche die CPU zu
5 # 100% auslastet und ca. 1000 Sekunden dauert
6 time.sleep(500) # Warte nochmal 500 Sekunden (CPU-Kosten ~ 0%)
7 echtzeit_dauer = time.perf_counter() - echtzeit_start # ~ 1500
8 cpuzeit_dauer  = time.process_time() - cpuzeit_start  # ~ 1000
```

9.3.10. argparse

Mit dem Modul argparse kann man ganz einfach benutzerfreundliche Kommandozeileninterfaces herstellen. Kurz gesagt, legt man mit argparse fest, welche Kommandozeilenparameter es gibt, welche Parameter optional sind und welche notwendig sind. Sind die Kommandozeilenparameter gültig, wird ein Verzeichniss der Parameter erstellt, die den Programm dann zur Verfügung stehen.

Zuerst erstellen wir einen "Argumentparser" mit

```
import argparse
parser = argparse.ArgumentParser(description='Beschreibung d. Programms')
```

Als Parameter description übergibt man einen String, der das Programm genau beschreibt. Als nächstes erklären wir, welche Parameter dem Programm per Kommandozeile übergeben werden können. Außerdem können wir Standardwerte festlegen oder bestimmen, welche Parameter optional sind und welche nicht. Dazu benutzen wir die Memberfunktion add_argument. Neben dem Namen der Kommandozeilenoption, erklären wir noch einige optionale Parameter der Funktion.

```
name     # Name des Parameters z.B. spam
action   # Wir besprechen hier:
         # (1) 'store': der nach spam folgende Parameter wird gespeichert
         # (2) 'store_true': speichert True (wenn spam ein Parameter ist)
type     # Legt den Typ fest, der gespeichert wird
default  # Legt einen Standardwert fest, auch wenn spam kein Parameter ist
required # Wenn required=True ist, muss spam Parameter sein
help     # String zur Erklaerung des Parameter fuer den Benutzer
```

Um Übergebene Kommandozeilenparameter auf Gültigkeit zu prüfen und im Anschluss Objekt mit den entsprechenden Parametern zu erstellen, rufen wir die Memberfunktion parse_args auf. Diese gibt (auch wenn wir in diesem Skript nicht genug auf Namespaces eingegangen sind) einen Namespace, der die übergebenen Parameter beschreibt. Aus diesem machen wir ein Verzeichniss mit der eingebauten Funktion vars. Die Schlüssel des Verzeichnisses sind die Namen der Parameter:

```
argumente = vars(parser.parse_args()) # Erstellt Verzeichniss d. Argumente
print(argumente)                      # Druckt das Verzeichniss aus
```

Das ganze versteht man am besten anhand eines Beispiels: Hier wollen wir ein simples Programm schreiben, das zwei Zahlen addiert oder subtrahiert. Dabei sollen dem Programm die Zahlen x und y und der Operand op in beliebiger Reihenfolge übergeben werden. Wenn man einen Parameter vergisst soll die Hilfe ausgegeben werden, die beschreibt wie das Programm funktioniert. Die Berechnungsfunktion sieht so aus:

```
def berechne(x,y,op):
  if op == 'plus':
    print('{} + {} = {}'.format(x,y,x+y))
  else:
    print('{} - {} = {}'.format(x,y,x-y))
```

Jetzt erstellen wir den Argumentenparser:

```
import argparse
parser = argparse.ArgumentParser(description='Berechnet x op y')
parser.add_argument(
  '-x', action='store', type=float, required=True, help='Die Zahl x')
parser.add_argument(
  '-y', action='store', type=float, required=True, help='Die Zahl y')
parser.add_argument(
  '-op', action='store', type=str, required=True, help='plus oder minus')
```

Jetzt setzen wir alles im main-modul zusammen. Das heißt, wir lassen der Argumentparser die übergebenen Parameter interpretieren und rufen dann die Funktion `berechne` auf. Der vollständige Code ist wie folgt.

```
import argparse
parser = argparse.ArgumentParser(description='Berechnet x op y')
parser.add_argument(
  '-x', action='store', type=float, required=True, help='Die Zahl x')
parser.add_argument(
  '-y', action='store', type=float, required=True, help='Die Zahl y')
parser.add_argument(
  '-op', action='store', type=str, required=True, help='plus oder minus')

def berechne(x,y,op):
  if op == 'plus':
    print('{} + {} = {}'.format(x,y,x+y))
  else:
    print('{} - {} = {}'.format(x,y,x-y))

def main():
  args = vars(parser.parse_args()) # Erstellt Verzeichniss d. Argumente
  print(args)
  berechne(**args)

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

9.3.11. multiprocessing

Auch in **Python** können wir nebenläufige Programme schreiben, indem wir (mehr oder weniger) voneinander unabhängige Programmabschnitte parallel ausführen. Im aller einfachsten Fall, wo wir eine Funktionen `f` mit Parametern `p_1, ..., p_k` parallel aufrufen möchten, gehen wir wie folgt vor: Wir importieren `multiprocessing` und erstellen einen sogenannten "Arbeiterpool". Ein Arbeiterpool ist eine Menge von Arbeiterprozessen, die ihre Aufgaben parallel bearbeiten können. Man kann die Größe des Pools mit dem Parameter `processes` selbst angeben oder das Betriebssystem die maximale Anzahl von Arbeitern selbst bestimmen lassen. Unser Programmcode sieht bis jetzt zum Beispiel so aus:

```
import multiprocessing
if __name__ == '__main__':
```

```
3   arbeiterpool = multiprocessing.Pool(processes=2) # Pool der Groesse 2
```

Haben wir eine Funktion `f` definiert und eine Liste `data` von (Listen von) Argumenten, die wir `f` übergeben, so können wir diese Funktionen wie folgt parallel ausführen.

```
1 import multiprocessing
2 import time
3 def f( x,t ):
4   time.sleep(t)
5   print(x)
6 if __name__ == '__main__':
7   arbeiterpool = multiprocessing.Pool(processes=2) # Pool der Groesse 2
8   data = [('Hallo', 2), ('Welt',1), ('Python',3), ('ist',1), ('cool',2)]
9   arbeiterpool.starmap( f, data )
```

Es ist ganz wichtig, dass wir hier abfragen, ob das Modul das main-Modul ist, denn beim Parallelisieren mit `multiprocessing` wird das main-Modul ggf. in sich selbst neu eingebunden. Wenn man mit Funktionen arbeitet, die nur ein Argument bekommt, nimmt man oft `arbeitspool.map` statt `arbeitspool.starmap`.

9.3.12. subprocess

Manchmal möchte oder muss man ein anderes Programm aus **Python** heraus aufrufen. Dazu verwendet man das Modul `subprocess`. Man kann mit dem aufgerufenen Programm kommunizieren (zum Beispiel Eingaben oder Signale Senden und Ausgaben empfangen), wie das genau funktioniert erklären wir hier jedoch nicht und verweisen auf [Pytc, Library, Concurrent Execution, Subprocess management]. Ein Programm `programm` mit einer Kommandozeilen Parametern `'par_1', ..., 'par_k'` startet man wie folgt:

```
1 import subprocess
2 p = subprocess.run([programm, par_1, ..., par_k], stdout=subprocess.PIPE)
```

Im Anschluss liegt der Rückgabewert des Programms `programm` in `p.returncode` und der Standardoutput von `programm` liegt in `p.stdout`.

Wir machen hier darauf aufmerksam, dass `subprocess.run` erst ab der **Python**-Version 3.5 existiert. Für führere **Python**-Versionen, kann man `subprocess.call` verwenden; diese Funktion gibt aber nur den Rückgabewert des Programms zurück, den Standardoutput muss man sich anders besorgen.

9.3.13. sys

Das Modul `sys` hält einige Variablen und Funktionen bereit, die start vom **Python**-Interpreter abhängen. Zum Beispiel ist die **Python**-Version durch das Tupel `sys.version_info` beschrieben und der Standardoutput wird durch `sys.stdout` abgebildet.

9.3.14. Weitere Python Module

Um numerische Berechnungen durchzuführen, gibt es das **Python**-Projekt `scipy`. Es ist sehr gut und baut auf `numpy` auf.

Sowohl numerische Berechnungen als auch algebraische Modellierung kann man mit dem **Python**-Projekt `SageMath` durchführen. Dieses greift auf eine riesige Bibliothek von Projekten zu, die in verschiedenen Programmiersprachen geschrieben sind. Es ist sehr gut.

Um ein bisschen zu malen oder mit Bildern zu arbeiten empfehlen wir noch `pygame`, `cairo` und `PIL`. Auch diese Module sind sehr gut.

10. Python + C = ♡

Wir nutzen **C/C++** für Programme, die (so gut wie) ausschließlich zeitkritische Probleme lösen sollen (z.B. einen Poissonlöser [ABH17]) oder die direkten Hardwarezugriff brauchen (z.B. Treiber). Für alles andere nutzen wir **Python**. Es gibt mehrere Möglichkeiten die Vorteile beider Sprachen zu kombinieren. Hier stellen wir zwei Möglichkeiten vor. Bei beiden bindet man in **C** geschriebenen Bibliotheken in ein **Python** Projekt ein. Die sogenannten "Extension Modules" sind nicht so schnell zu coden wie die sogenannten "CTypes", haben aber eine höhere Performance als letztere. Die "CTypes" sind nicht so performant wie "Extension Modules", sind dafür aber schneller zu coden.

10.1. Extension Modules

Bevor wir erklären, was "Extension Modules" sind und wie sie in **Python** eingebunden werden können, machen wir darauf aufmerksam, dass Extension Modules nur mit dem **Python**-Interpreter CPython funkionieren: "The C extension interface is specific to CPython, and extension modules do not work on other Python implementations.", see [Pytc, Pytb, Extending Python with C or C++; §1]. Das liegt daran, wie Extension Modules technisch umgesetzt sind. Der **Python**-Interpreter CPython ist ein in **C** geschriebenes Programm. So wie alle in **C** geschriebenen Programme, kann auch CPython in **C** geschriebene Bibliotheken zur Laufzeit nachladen und benutzen. Ein Extension Module ist (rein technisch gesehen) eine **C** Bibliothek, die CPython zur Laufzeit nachladen und benutzen kann.

Nun besprechen wir, wann man Extension Modules verwenden sollte und wie man sie verwendet.

10.1.1. Die Ausgangssituation

Gehen wir davon aus, dass wir in einem **Python** Projekt eine Funktion `addition_py` haben, die besonders schnell ausgeführt werden muss oder Hardwarezugriff braucht. Der Einfachheit halber wollen wir davon ausgehen, dass `addition_py` eine feste Anzahl von Parametern eines festen Typs benötigt und einen definierten Ausgabetyp hat.

10.1.2. Schnelle C-Funtkion implementieren

Als erstes implementieren wir eine in **C** geschriebenene Funktion `addition`, die sich wie die Funktion `addition_py` verhält. Außerdem binden wir die `<Python.h>` ein, die vom **Python**-Standard bereit gestellt wird[5]. Es ist wichtig, dass `<Python.h>` vor allen anderen Bibliotheken einbegunden wird, denn `<Python.h>` stellt gewissen Makros bereit, welche die anderen eingebundenen Bibliotheken unter Umständen modifiziert. Unser Beispiel sieht also momentan so aus:

```
1 // Datei: addition.c
```

[5] Es kann sein, dass man die Entwicklerbibliotheken nachträglich installieren muss. Unter Debianderivaten geschieht dass mit `sudo apt-get install python3-dev`.

```
2 // Die Python.h muss vor allen anderen Bibliotheken eingebunden werden.
3 #include <Python.h>
4
5 int addition( int op1, int op2 )
6 {
7   return op1 + op2;
8 }
```

10.1.3. Jede schnelle C-Funktion braucht eine Hilfsfunktion

Nun erstellen wir (immer noch in **C**) für jede schnelle **C**-Funktion genau eine Hilfsfunktionen. Mithilfe der Hilfsfunktionen kann der (ebenfalls in **C** geschriebenene) **Python**-Interpreter CPython unsere Funktion `addition` innerhalb unseres **Python** Projekts aufrufen. Wir erklären nun, wie die Hilfsfunktion aufgebaut ist.

(1) Die Signatur einer jeden Hilfsfunktionen ist:

```
1 static PyObject* hilfsfunktionsname( PyObject* self, PyObject* args )
```

(2) Nun definiert man alle **C**-Variablen, die wir für den Aufruf unserer **C**-Funktion brauchen. Das beinhaltet eine Variable für den Rückgabewert unserer **C**-Funktion.

```
1 static PyObject* add_hilfsfkt( PyObject* self, PyObject* args )
2 {
3   // Erstelle fuer Rueckgabe und jeden Parameter eine Variable.
4   int op1, op2, reuckgabe;
5   // Hier gehts dann gleich weiter
6   // ...
7 }
```

(3) Dann interpretieren wir die in **Python** übergebenen Parameter als **C**-Variablen. Dazu benutzen wir die **C**-Funktion

```
1 int PyArg_ParseTuple( PyObject* args, const char* format, ... )
```

Was das erste Argument ist, müssen wir hier nicht nicht verstehen. Das zweite Argument ist ein (konstanter) String, der erklärt, wie die übergebenen **Python**-Parameter in **C**-Variablen konvertiert werden sollen. Dieses Konzept sollte von der **C**-Funktion `printf` bekannt sein. Wir behandeln hier nur den einfachsten Fall, i.e. der **Python**-Funktion werden nur die folgenden Typen übergeben: `int`, `float` oder `str`. Wie man andere **Python**-Objekte zu **C**-Variablen konvertiert und wie man mit variablen Parametern umgeht behandeln wir hier nicht und verweisen auf [Pytc, Extending Python with C or C++; §1.7]. Hat die **Python**-Funktion genau k Parameter `p_1,...,p_k` und soll der i-te Parameter als `int`, `float` bzw. `str` interpretiert werden, so erstellen wir einen String der Länge k, dessen i-tes Zeichen `'i'`, `'f'` bzw. `'z'` ist. Im Anschluss übergeben wir der **C**-Funktion `PyArg_ParseTuple` genau k Referenzen auf die Objekte, die wir im vorherigen Schritt erstellt haben. In unserem Beispiel sieht die Hilfsfunktion bis jetzt also so aus:

```
1 static PyObject* add_hilfsfkt( PyObject* self, PyObject* args )
2 {
```

```
3     // Erstelle fuer Rueckgabe und jeden Parameter eine Variable.
4     int op1, op2, reuckgabe;
5     // Wandel Python-Variablen in C-Variablen um.
6     PyArg_ParseTuple( args, "ii", &op1, &op2 )
7     // Hier gehts dann gleich weiter
8     // ...
9 }
```

Die Funktion `PyArg_ParseTuple` gibt genau dann `0` zurück, wenn die Konvertierung fehlgeschlagen hat. In diesem Fall wollen wir die Hilfsfunktion ebenfalls abbrechen und geben `NULL` zurück.

(4) Hat die Konvertietung der **Python**-Parameter in **C**-Variablen funktioniert, können wir unsere schnelle **C**-Funktion mit den gegebenen Parametern aufrufen und den Rückgabewert speichern.

(5) Nun konvertieren wir den Rückgabewert unserer schnellen **C**-Funktion in ein **Python**-Objekt und geben dieses als Rückgabewert der Hilfsfunktion zurück. Die Konvertierung geschieht mit der **C**-Funtkion `Py_BuildValue( char* , ... )`. Ähnlich wie oben gehen wir hier davon aus, dass der Rückgabewert unserer schnellen **C**-Funktion der Typ `int`, `float` oder `char*` ist. Für andere Rückgabetypen verweisen wir auf [Pytc, Extending Python with C or C++; §1.9]. Der zu übergebe String ist `"i"`, `"f"` bzw. `"s"` je nachdem ob unser Rückgabetyp unserer schnellen **C**-Funktion `int`, `float` oder `char*` ist.

Damit sieht die fertige Hilfsfunktion für unser Beispiel so aus:

```
1  static PyObject* add_hilfsfkt( PyObject* self, PyObject* args )
2  {
3      // Erstelle fuer Rueckgabe und jeden Parameter eine Variable.
4      int op1, op2, reuckgabe;
5      // Wandel Python-Variablen in C-Variablen um.
6      if( !PyArg_ParseTuple( args, "ii", &op1, &op2 ) ) return NULL;
7      // Rufe die C-Funktion auf.
8      reuckgabe = addition( op1, op2 );
9      // Wandle C-Variable in Python-Variable um.
10     return Py_BuildValue( "i", reuckgabe );
11 }
```

10.1.4. Die Hilfsfunktion werden in einem Modul zusammengefasst

Jetzt beschreiben wir ein Modul `hilfsmodul` welches am Ende in **Python** eingebunden werden kann und unsere Funktionen bereit stellt. Ein Modul bestehend aus einer sogenannten “Method Table” und einer “Modulinitialisierung”. Die Method Table ist ein Array von `struct`s bestehend aus den vier Feldern: Der Python-Name der Funktion; der Funktionspointer auf die in **C** implementierte Funktion; der `enum` der die Parameterübergabe beschreibt (siehe unten); die Python-Beschreibung der Funktion. Der Array wird mit `{NULL, NULL, 0, NULL}` beendet.

Der besagte `enum` beschreibt die Möglichkeiten wie die Parameter übergeben werden. Wir konzentrieren uns hier einzig und allein auf den Wert `METH_VARARGS` welcher der

einfachen Übergabe von Variablen entspricht. Die anderen Möglichkeiten Variablen zu übergeben sind in [Pytc, Extending Python with C or C++; §1.4] beschrieben.

Die Method Table für unser Beispiel sieht so aus:

```
static PyMethodDef HilfsFunktionen[] =
{
  {"addition_py", add_hilfsfkt, METH_VARARGS, "Addiere zwei Ganzzahlen"},
  {NULL, NULL, 0, NULL}
};
```

Nun können wir unsere Modulinitialisierung anfertigen. Die sieht für ein einfaches Projekt immer so aus (wobei man `"hilfsmodul"` durch einen beliebigen Namen für sein Modul ersetzten kann und `HilfsFunktionen` der Name der Method Table ist).

```
PyMODINIT_FUNC inithilfsmodul(void)
{
  (void) Py_InitModule("hilfsmodul", HilfsFunktionen);
}
```

Die gesamte **C**-Datei sieht demnach so aus:

```
// Datei: addition.c
// Die Python.h muss vor allen anderen Bibliotheken eingebunden werden.
#include <Python.h>

// Unsere schnelle C-Funktion addition.
int addition( int op1, int op2 )
{
   return op1 + op2;
}

// Die Hilfsfunktion fuer unsere schnelle C-Funktion addition
static PyObject* add_hilfsfkt( PyObject* self, PyObject* args )
{
  // Erstelle fuer Rueckgabe und jeden Parameter eine Variable.
  int op1, op2, reuckgabe;
  // Wandel Python-Variablen in C-Variablen um.
  if( !PyArg_ParseTuple( args, "ii", &op1, &op2 ) ) return NULL;
  // Rufe die C-Funktion auf.
  reuckgabe = addition( op1, op2 );
  // Wandle C-Variable in Python-Variable um.
  return Py_BuildValue( "i", reuckgabe );
}

// Die Method Table.
static PyMethodDef HilfsFunktionen[] =
{
  {"addition_py", add_hilfsfkt, METH_VARARGS, "Addiere zwei Ganzzahlen"},
  {NULL, NULL, 0, NULL}
};

// Die Modulinitialisierung.
PyMODINIT_FUNC inithilfsmodul(void)
{
```

```
34   (void) Py_InitModule("hilfsmodul", HilfsFunktionen);
35 }
```

10.1.5. Das Modul kompileren und benutzen

Jetzt ist es an der Zeit, das Modul zu bauen. Dazu benutzen wir ein einfaches **Python** Skript. Es besteht aus wenigen Zeilen Code. Wir binden `distutils.core` ein, definieren unsere Erweiterung (bestehend aus einem Namen und einer Liste von zu kompilierenden **C**-Dateien) und rufen die Funktion auf, die unsere Erweiterung baut. Die Namen der Variablen sind selbsterklärend.

```
1  # Datei: modul_bauen.py
2
3  from distutils.core import setup, Extension
4  modulname = 'hilfsmodul'
5  c_quellen = ['addition.c']
6  ordername = 'hilfsmodul'
7  ver       = '0.1'
8  beschr    = 'Unser␣Hilfsmodul'
9  ext       = Extension(name=modulname, sources=c_quellen)
10 setup(name=ordername, version=ver, description=beschr, ext_modules=[ext])
```

Nun rufen wir das Skript zweimal auf. Im ersten Schritt wird unser Modul gebaut und im zweiten Schritt wird es installiert, so dass es vom **Python**-Interpreter CPython genutzen werden kann.

```
1 python modul_bauen.py build
2 python modul_bauen.py install --user
```

Von jetzt an kann man unsere Funktion durch das Modul `hilfsmodul` aufrufen.

```
1 import hilfsmodul
2 print ('4␣+␣5␣=', hilfsmodul.addition_py(4,5))
```

Wird die **C**-Datei verändert erneuert man das installierte Modul wie folgt.

```
1 python modul_bauen.py build
2 python modul_bauen.py install --user --force
```

10.2. CTypes

Um ein **Python** Projekt (fast) ohne Aufwand um eine bereits in **C** implementierte Funktion zu erweitern, kann man “CTypes” verwenden. Das funktioniert sogar, wenn die Funktion nur als “shared library” existiert und der Quellcode nicht vorliegt.

Nehmen wir an, wir brauchen eine **Python**-Funktion `subtraktion`, welche die Differenz zweier Gleitkommazahlen berechnet. Nehmen wir außerdem an, dass die Funktion bereits in kompilierter Form vorliegt und folgende Signatur hat.

```
1 double subtraktion(double, double);
```

Die zugehörige shared library heißt `libminus.so`.

Um `libminus.so` nutzen zu können, binden wir zunächst das Paket `ctypes` ein.

```
1 import ctypes
```

Dann laden wir die Bibliothek `libminus.so`.

```
1 bib = ctypes.cdll.LoadLibrary("./libminus.so")
```

Auf Funktionen `f` und globalen Variablen `x` von `libminus.so` können wir nun durch `bib.f` und `bib.x` zugreifen. Wir erstellen zunächst eine Variable `subtraktion_py` die auf die **C**-Funktion `subtraktion` zeigt.

```
1 subtraktion_py = bib.subtraktion
```

Nun legen wir fest, wie der Rückgabetyp der **C**-Funktion in **Python** interpretieren werden soll und wie die **Python**-Argumente in **C** interpretiert werden sollen. Ist ein **C**-Parameter oder der **C**-Rückgabewert vom Typ `typ` und ist dieser Typ durch den **C**-Standard definiert, so können wir die Konvertierung in diesen Typ mit `ctypes.c_typ` festlegen. Die Konvertierung in den **C**-Typ `float` wird also durch `ctypes.c_float` festgelegt. Den Rückgabetyp einer Funktion `f` setzt man mit `f.restype = ctypes.c_...`. Die Argumenttypen einer Funktion `f` wird durch eine Listen von `ctypes.c_...` angegeben, also `f.argtypes = [ctypes.c_..., ...]`. Den Rückgabetyp und die Argumente unserer Funktion `subtraktion_py` setzen wir also so:

```
1 subtraktion_py.restype  = ctypes.c_double
2 subtraktion_py.argtypes = [ctypes.c_double, ctypes.c_double]
```

Jetzt können wir die **C**-Funktion aus **Python** heraus ausrufen.

```
1 print( subtraktion_py(3.5, 6.2) )
```

Weiterführende Konzepte wie zum Beispiel variable Argumente findet man in [Pytc, Library, Generic Operating System Services, CTypes].

A. Installation

In diesem Abschnitt besprechen wir ganz kurz wie man **Python3** und einen von uns bevorzugen Editor installiert.

A.1. Python3 installieren

Nutzt man eine auf Debian basierende GNU/Linux Distribution (so wie Debian, Ubuntu oder Mint), installiert man **Python3** zum Beispiel so:

```
1 sudo apt-get install python3
```

Nutzt man Mac OS X gibt es zwei Möglichkeiten. Ab Mac OSX 10.8 ist **Python2** bereits installiert und das reicht für unsere Zwecke eigentlich aus. Um **Python3** zu nutzen, lädt man **Python3** von der offiziellen Webseite [Pyta] herunter und installiert es.

Nutzt man Windows, lädt man **Python3** von der offiziellen Webseite [Pyta] herunter und installiert es. Wichtig ist hierbei, dass man sich vom Installationsprogramm die PATH-Variable automatisch setzen lässt.

A.2. PyCharm Community Edition installieren

Als IDE (Integrated Developer Environment) verwenden wir PyCharm Community Edition 2016.3. Sowohl bei GNU/Linux als auch bei Mac OS X und auch bei Windows lädt man die "Community Edition" von PyCharm Webseite [Jet16] herunter (und nicht die "Professional Edition"). Diese wird dann entpackt beziehungsweise installiert.

Literatur

[ABH17] Clelia Albrecht, Felix Boes, and Johannes Holke. *C-Programmierkurs für Fortgeschrittene*. selfpublished, 2017.

[ISO99] Programming languages – C. Standard, International Organization for Standardization and International Electrotechnical Commission, December 1999.

[Jet16] JetBrains. PyCharm 2016.3.2 Community Edition. `https://www.jetbrains.com/pycharm/`, December 2016.

[Lot10] Steven F. Lott. *Building Skills in Python – Release 2.6.5*. selfpublished, April 2010.

[Mun] Randall Munroe. Python. `https://xkcd.com/353/`.

[Pyta] Python Software Foundation. The Python Language. `https://www.python.org/downloads/`.

[Pytb] Python Software Foundation. The Python Language Reference Version 2.7.13. `https://docs.python.org/2.7/reference/`.

[Pytc] Python Software Foundation. The Python Language Reference Version 3.6.0. `https://docs.python.org/3.6/reference/`.