# Python Grundlagen

Joern Ploennigs · AI4SC

> Schlangen, warum ausgerechnet müssen es Schlangen sein?
>
> — Indiana Jones

Python ist eine allgemeine Programmiersprache, die in den letzten Jahren sehr stark an Popularität gewonnen hat, insbesondere im Bereich des Maschinellen Lernens, obwohl die Programmiersprache älter ist als andere etablierte Programmiersprachen wie Java, C# und JavaScript. Die neue Popularität ist zum einen darauf zurückzuführen, dass die Sprache sehr auf Lesbarkeit und Einfachheit ausgelegt ist und dadurch sehr leicht zu erlernen ist, insbesondere von Nicht-Informatikern, wie Maschine Learning Anwendern. Ferner gibt es durch ihr Alter heutzutage sehr viele Bibliotheken und für fast jedes Problem eine existierende Lösung, insbesondere im numerischen Bereich.

Python ist eine interpretierte Sprache und sie wird zur Laufzeit in Maschinencode übersetzt, der vom Computer ausführbar ist. Dadurch muss sie nicht erst Compiliert werden, wie C#, was gerade in der Data Science aufwändig ist, da viele Maschine Learning Modelle dynamisch iterativ, durch probieren, entwickelt werden. Zusätzlich ist Python untypisiert, weshalb Variablen nicht explizit und mit unveränderlichen Datentypen deklariert werden müssen, sondern diese dynamisch zugewiesen werden. Dadurch muss man beim Arbeiten mit Daten weniger auf Details wie Datentypen achten. Typische Performance-Probleme, die bei interpretierten Sprachen auftreten, werden durch spezialisierte numerische Bibliotheken überwunden.

## Variablennamen und Variablencodierung

Bevor wir in die Details von Python einsteigen, ein paar allgemeine Bemerkungen zum Codierungsstil. Es gibt immer eine Vielzahl von Möglichkeiten, Variablennamen, Benennungsschemata und Algorithmen zu wählen. Dies ist oft von geringer Bedeutung, kann aber manchmal zu Fehlern führen, die frustrierend schwer zu debuggen sind. Im Folgenden diskutieren wir einige allgemeine

Strategien. Wie immer, fühlen Sie sich frei, gegen eine dieser Regeln zu verstoßen, aber seien Sie in der Lage zu erklären, warum Sie das tun!

**Passende Variablennamen**

Was „angemessen“ ist, hängt von der Aufgabe ab. Anstatt eine Variable nur `x` zu benennen, könnte man `concrete_strength_mpa` nutzen, um klar auszudrücken, dass es sich um die Druckfestigkeit von Beton in Megapascal handelt. Wenn Sie wiederum eine kleine Schleife schreiben, die eine Nachricht dreimal ausgibt, wird der Schleifenzähler oft einfach `i` genannt:

```
for i in range(3):
    print(i, "Hallo")
```

Das einfache `i` macht den Code einfacher zu verstehen als ein komplexerer Name, z.B. `greeting_counter`:

```
for greeting_counter in range(3):
    print(greeting_counter, "hi there!")
```

Allerdings bedeutet dies nicht, immer die einfachsten Variablennamen zu wählen. `greeting_counter` könnte eine gute Wahl sein, wenn Sie mit verschachtelten Schleifen und vielen Zählern arbeiten und genau wissen müssen, was die Schleife zählt.

**Überschreiben von Variablen vermeiden**

Data-Science-Aufgaben beginnen in der Regel mit dem Laden, Bereinigen und Filtern von Daten in den Schritten

```
## Laden
data = pd.read_csv("data.csv")
# Überprüfen Sie, ob das Laden erfolgreich war.
...
## Säubern
data = data.dropna(["var1", "var2"], axis=1)
# Daten weiter bereinigen
...
## Unterteilen
data = data[data.var3.isin(interesting_cases)]
# Daten weiter unterteilen
## Beginnen Sie hier mit der eigentlichen ML Aufgaben
```

Hierbei wird in jeden Schritt die Variable `data` überschrieben. Das ist unproblematisch, solange die Schritte strikt sequentiell ausgeführt werden. In Notebooks kann es jedoch Problemen führen, wenn man Zellen überspringt oder zurückspringt, da dann die Variable schnell falsch belegt sein kann. Auch ist es problematisch, wenn Schritte geändert werden (Zum Beispiel andere Filter beim Säubern), dann die ursprüngliche Variablenbelegung fehlt und alles wiederholt werden muss. Im

obigen Beispiel erhalten Sie einen Fehler, wenn Sie den Reinigungscode erneut ausführen, der besagt, dass die Variablen `var1` und `var2` nicht gefunden wurden.

Sinnvoller ist es temporäre Variablen zu erstellen und Sie bei Bedarf mit `del` zu löschen, wenn man den Speicher braucht.

Wenn du etwa Sensorwerte von einer Baustelle analysierst, kann es sinnvoll sein, Zwischenschritte in neuen Variablen wie `sensor_raw`, `sensor_cleaned`, `sensor_filtered` zu speichern. Das hilft besonders, wenn man im Notebook Zellen mehrfach oder außer der Reihe ausführt.

**Nutzen von Benennungsschema**
Eine weitere häufige Aufgabe ist das Durchlaufen einer Schleife über alle Elemente einer Sammlung. Die Sammlungen haben in der Regel eine bestimmte Bedeutung, und daher neigen Sie dazu, sie entsprechend zu benennen. Aber die einzelnen Elemente, die Sie in der Schleife extrahieren, haben eine recht ähnliche Bedeutung, und Sie sind versucht, sie etwas sehr Ähnliches zu nennen.

Betrachten wir ein verwirrendes Beispiel:

```
friend = ["Max", "Moritz", "Hensel", "Gretel"]
for person in friend:
    ...
    ## Wer ist eine Person, wer ist ein Freund?
    ## Was ist die Sammlung und was ist ein Element?
    ## Sind sie überhaupt miteinander verwandt?
```

Es gibt zwei Probleme mit den gewählten Namen: a) sie sind beide im Singular, daher ist unklar, ob es sich um ein einzelnes Element handelt oder eine Sammlung; und b) sie sind ziemlich unterschiedlich, daher ist nicht klar, ob `person` und `friend` irgendwie zusammenhängen.

Eine Alternative wäre es, konsequent das Pluralende `-s` zu verwenden oder vielleicht das Suffix `_list`:

```
friends = ["Max", "Moritz", "Hensel", "Gretel"]
for friend in friends:
    ...
    ## friends: Plural, also eine Sammlung
    ## friend: Singular, also ein Element in 'friends'
```

Wählen Sie ein kohärentes Benennungsschema. Benutzen Sie dabei folgende Regeln:

- Verwenden Sie Namen, die die Bedeutung und den Inhalt der Variablen klar vermitteln.
- Vermeiden Sie übermäßig generische Namen wie „Daten“ oder „df“.
- Halten Sie die Namenskonventionen in Ihrem Code konsistent.
- Verwenden Sie Kleinbuchstaben mit Unterstrichen (_) zur Trennung von Wörtern.
- Vermeiden Sie Sonderzeichen wie ä, ö, ü, ß
- Vermeiden Sie Abkürzungen oder Akronyme, es sei denn, sie sind allgemein verständlich (z. B. „BIP“).

- Streben Sie nach einem Gleichgewicht zwischen Klarheit und Kürze.

Hier einige Beispiele:

| Good | Bad | Description |
|---|---|---|
| customer_id | cust_id | Klare und prägnante Darstellung der Kunden-ID. |
| order_date | date | „Date“ ist mehrdeutig und könnte jedes Datum bedeuten. |
| is_vip | vip | Beschreibend, zeigt VIP-Kundenstatus an. |
| total_price | price | „Preis“ könnte sich auf den Preis eines einzelnen Artikels beziehen. |
| num_orders | orders | Gibt die Anzahl der Bestellungen für einen Kunden an. |

## Code Blöcke

Ein besonders charakteristisches Element von Python ist das Einrücken von Code um Codeblöcken zu kennzeichnen.

Betrachten Sie das folgende Beispiel: Der *for*-Loop umfasst drei Zeilen Code (*print*, *if* und nested *print*), die durch einen zusätzlichen Einzug von 4 Leerzeichen gekennzeichnet ist. Die *if*-Bedingung öffnet einen neuen, verschachtelten Code-Block, der weiter 4 Zeichen engerückt ist. Die letzte *print*-Anweisung ist nicht eingerückt, da der for-Loop abgeschlossen ist. Sie gibt das folgende Ergebnis aus:

```
for i in range(4):           # Wiederholschleife:   0 Einzug
    print(i)                 # Code-Block von for:  4 Einzug
    if i > 2:                # Wenn-Dann Bedingung: 4 Einzug
        print("fast fertig") # Code-Block von if:   8 Einzug
print("fertig")              # Nach dem for-Loop:   0 Einzug
```

```
0
1
2
3
fast fertig
fertig
```

Ähnliche Einrückungsregeln gelten für alle Codeblöcke, einschließlich Funktionsdefinitionen und Ausnahmebehandlungen: Der Block beginnt mit einem Doppelpunkt am Ende der Deklarationszeile und geht bis die zusätzliche Einrückung abschließt.

Ein sehr einfaches Beispiel für Ausnahmebehandlung ist `try`/`except`. Damit können wir einen erwartbaren Fehler abfangen und eine verständliche Meldung ausgeben. Wenn ein Fehler *nicht* abgefangen wird, zeigt Python einen **Traceback** an. Diesen liest man in der Regel von unten nach oben: Die letzte Zeile nennt meist den Fehlertyp und die konkrete Meldung, die Zeilen darüber zeigen, **wo** der Fehler entstanden ist.

```
text = "abc"

try:
    number = int(text)
except ValueError as e:
    print("Fehler beim Umwandeln:", e)
```

```
Fehler beim Umwandeln: invalid literal for int() with base 10: 'abc'
```

Wenn wir das `except` weglassen würden, sähen wir einen Traceback, der unten mit einer Meldung wie `ValueError: invalid literal for int() with base 10: 'abc'` endet. Für das Debugging ist genau diese letzte Zeile oft der wichtigste Startpunkt.

## Variablen und ihre Zuweisung

Die wichtigsten einfachen Datentypen in Python sind Gleitkommazahlen (floats), Ganzzahlen, Logische und Zeichenketten. Das folgende Beispiel zeigt dein Float. Wir können den Datentyp (Klasse) der Variablen mit der Funktion `type` abfragen:

```
a = 1.0  # Gleitzahl
type(a)
```

```
float
```

Ein Float wird erstellt, indem man explizit `1.0` anstelle von `1` schreibt. Das Letztere wird als Integer interpretiert.

```
b = 2  # Ganzzahl
type(b)
```

```
int
```

Beachten Sie, dass Python UTF-8-Zeichen in Variablennamen unterstützt, wie am Beispiel der logischen Variable `λ` zu sehen ist.

```
λ = False  # Logische Werte mit 'False' (Falsch) und 'True' (Wahr)
type(λ)
```

```
bool
```

Ein Text-String wird mit einfachen ' oder doppelte " Anführungszeichen deklariert

```
s = 'text'  # String
type(s)
```

```
str
```

Wenn nötig, kann man einen Typ explizit in einen anderen umwandeln:

```
str(a)  # Konvertiert to string
```

```
'1.0'
```

```
int(λ)  # Konvertiert to integer
```

```
0
```

Man kann sehen, dass die mit `False` belegte logische Variable `λ` als Ganzzahl zu 0 konvertiert wird. Analog dazu würde `True` zu 1 konvertiert werden. Bei der umgekehrten Konvertierung wird jede Zahl außer 0 zu `True` konvertiert.

## Mathematische, logische und andere Operatoren

Python bietet verschiedene Operatoren für Berechnungen und Logische Auswertungen.

**Arithmetische Operatoren: +, -, *, /, //, %, ****

Arithmetische Operatoren führen die bekannten mathematischen Operationen aus. Sie werden mit numerischen Werten verwendet, um allgemeine mathematische Operationen auszuführen.

| Operator | Beispiel | Wirkung |
|---|---|---|
| Addition | `x + y` | `10 + 3` (Ergebnis: 13) |
| Subtraktion | `x - y` | `10 - 3` (Ergebnis: 7) |
| Multiplikation | `x * y` | `10 * 3` (Ergebnis: 30) |
| Division | `x / y` | `10 / 3` (Ergebnis: 3.33) |
| Ganzzahl-Division | `x // y` | `10 // 3` (Ergebnis: 3) |
| Modulo-Operator | `x % y` | `10 % 3` (Ergebnis: 1) |
| Potenzierung | `x ** y` | `10 ** 3` (Ergebnis: 1000) |

**Zuweisungsperatoren:** =, +=, -=, *=, /=, //=, %=

Zuweisungsoperatoren weisen Variablen Werte zu. Python verwendet für alle arithmetischen Operatoren die abkürzende Zuweisung, es gibt jedoch keine Inkrementierung oder Dekrementierung wie in C-ähnlichen Sprachen.

Bei der Zuweisung `x = y` wird der Wert der Variablen `y` (rechte Seite) der Variablen `x` (linke Seite) zugewiesen. Bei abkürzenden Zuweisungen wird jeweils eine Operation auf dem Wert einer Variablen ausgeführt und dieser neue Wert erneut der Variablen zugewiesen.

| Operator | Beispiel | Wirkung |
|---|---|---|
| Zuweisung | `x = y` | Variable `x` erhält den Wert `y` |
| Abkürzende Addition | `x += 4` | `x = x + 4` # Der Wert der Variablen `x` wird um 4 erhöht |
| Abkürzende Multiplikation | `x *= 4` | `x = x * 4` # Der Wert der Variablen `x` wird mit 4 multipliziert |
| Abkürzende Division | `x /= 4` | `x = x / 4` # Der Wert der Variablen `x` wird durch 4 dividiert |
| Abkürzende Ganzzahl-Division | `x //= 4` | `x = x // 4` # Der Wert der Variablen `x` wird durch 4 ohne Rest dividiert |
| Abkürzende Modulo | `x %= 4` | `x = x % 4` # Der Wert der Variablen `x` wird durch 4 dividiert und nur der Rest zugewiesen |

**Vergleichsoperatoren:** ==, !=,<, >, <=, >=

Vergleichsoperatoren vergleichen zwei Werte und liefern wahr oder falsch als Ergebnis. Zum Beispiel testet `a == b` ob `a` gleich `b` ist, und liefert je nachdem wahr oder falsch zurück, und `a != b` testet, ob `a` ungleich `b` ist.

Vergleichsoperatoren verknüpfen zwei Variablen / Werte, und das Ergebnis wird als wahr oder falsch ausgewertet.

Bei der Zuweisung `x = y` wird der Wert der Variablen `y` (rechte Seite) der Variablen `x` (linke Seite) zugewiesen. Bei abkürzenden Zuweisungen wird jeweils eine Operation auf dem Wert einer Variablen ausgeführt und dieser neue Wert erneut der Variablen zugewiesen.

| Operator | Beispiel | Wirkung |
|---|---|---|
| Gleich | `x == y` | wahr, wenn x gleich y ist |
| Ungleich | `x != y` | wahr, wenn x ungleich y ist |
| Größer | `x > y` | wahr, wenn x größer als y ist |
| Kleiner | `x < y` | wahr, wenn x kleiner als y ist |
| Größer gleich | `x >= y` | wahr, wenn x größer gleich y ist |
| Kleiner gleich | `x <= y` | wahr, wenn x kleiner gleich y ist |

**Logische Operatoren: and, or, not**

Logische Operatoren kombinieren boolesche Ausdrücke, die dann als wahr oder falsch ausgewertet werden. Z.B. wird hier die Variable `is_teenager = (alter > 13) and (alter < 19)` als wahr ausgewertet, wenn der Wert der Variablen alter gleich 14 ist.

`P` und `Q` stehen für logische Ausdrücke, z.B. `P = (x > 10)` und `Q = (x < 20)`.

| Operator | Beispiel | Wirkung |
|---|---|---|
| Logisches Und | `P and Q` | Gibt wahr (1) zurück, wenn die Ausdrücke `P` und `Q` beide wahr sind |
| Logisches Oder | `P or Q` | Gibt wahr (1) zurück, wenn einer der Ausdrücke `P` oder `Q` wahr ist |
| Logische Verneinung | `not P` | Verneint den Ausdruck `P`: aus wahr wird falsch, aus falsch wird wahr |

**Mitgliedsoperatoren: in, not in**

Mitgliedsoperatoren prüfen, ob ein Objekt Mitglied einer Liste ist, z.B. prüft `4 in [1,2,3]`, ob 4 in der Liste enthalten ist.

`x` steht hier für ein beliebiges Objekt, list für eine Auflistung.

| Operator | Beispiel | Wirkung |
|---|---|---|
| enthalten in | `x in list` | wahr, wenn x in der Liste `list` enthalten ist. |
| nicht enthalten | `x not in list` | wahr, wenn `x` nicht in der Liste `list` enthalten ist. |

**Identitätsoperatoren: is, is not**

Identitätsoperatoren vergleichen Objekte. Falls z.B. `x = [1, 2, 3]` gilt, also x eine Liste ist, dann liefert `type(x) is list` wahr, und `type(x) is not list` ist falsch. Sie prüfen dabei, ob zwei Objekte auf denselben Speicherbereich zeigen. `x is y` gibt wahr zurück, wenn beide Variablen dasselbe Objekt sind, d.h. auf denselben Speicherbereich zeigen.

| Operator | Beispiel | Wirkung |
|---|---|---|
| Gleich | `x is y` | wahr, wenn beide Variablen dasselbe Objekt sind. |
| Nicht gleich | `x is not y` | wahr, wenn die Variablen nicht dasselbe Objekt sind. |

**Bitweise Operatoren: &, ^, |, >>, <<**

Bitweise Operatoren führen bitweise Operationen auf ganzzahligen Operanden aus. D.h. die Operanden werden in ihre bitweise Darstellung konvertiert, z.B. `x = 5 = 0101`, `y = 3 = 0011`, und dann wird jedes Bit im Operanden `x` mit dem Bit an der passenden Position im Operanden `y` verknüpft.

| Operator | Beispiel | Wirkung |
|---|---|---|
| Bitweises Und | `x & y` | Ergebnisbit ist 1, wenn beide Bits 1 sind. |
| Bitweises Oder | `x \| y` | Ergebnisbit ist 1, wenn mindestens eines der Bits 1 ist. |
| Bitweises Entweder-Oder (XOR) | `x ^ y` | Ergebnisbit ist 1, wenn genau eines der Bits 1 ist. |
| Bitweises Nicht | `~ x` | Gibt das Einerkomplement von `x` zurück. |
| Shift-right | `x >> n` | Bits in `x` werden um `n` Positionen nach rechts geschoben. |
| Shift-left | `x << n` | Bits in `x` werden um `n` Positionen nach links geschoben. |

**Tips zur Anwendung**

Mathematische Operatoren haben auch eine Kurzversion die vor dem Zuweisungsoperator `=` stehen kann: z.B. `a += 1` entspricht `a = a + 1`, `a *= 2` ist äquivalent zu `a = a*2`.

```
i = 1
i *= 2
i *= 3
i  # 6
```

```
6
```

Logische Operationen funktionieren meistens wie erwartet. Insbesondere `>`, `<`, `>=` und `<=`. Wie in mehreren anderen Sprachen wird Gleichheit mit doppelten Gleichheitszeichen `==` getestet. Ungleichheit kann mit `!=` getestet werden, und die logische Negation ist `not`:

```
a = 1
a > 1  # False
a >= 1  # True
a == 1  # True
a != 1  # False
not a == 1  # False
```

```
False
```

Python unterstützt auch etwas weniger verbreitete, aber äußerst praktische Mehrweg-Vergleichsoperationen, zum Beispiel

```
0 < a < 2  # true
```

```
True
```

## Strings

Zeichenfolgen in Python können auf traditionelle Weise erstellt werden, entweder mit einfachen oder doppelten Anführungszeichen:

```
a = "Dies ist ein String mit doppelten Anführungszeichen."
b = 'Dies ist ein String mit einfachen Anführungszeichen.'
```

Beides sind äquivalente Möglichkeiten, einen String zu definieren. Die erste ist nützlich, um einen String zu erstellen, der ein einzelnes Anführungszeichen enthält, wie:

```
"Was ist richtig: Marios' Pizzeria oder Mario's Pizzeria?"
```

```
"Was ist richtig: Marios' Pizzeria oder Mario's Pizzeria?"
```

und die letztere ist besser, wenn ein doppeltes Anführungszeichen einbezogen werden soll, wie:

```
'Man nennt sie auch "Gänsefüßchen"'
```

```
'Man nennt sie auch "Gänsefüßchen"'
```

Es ist empfehlenswert, doppelte Anführungszeichen für Strings zu verwenden, da dies die meisten Texteditoren und IDEs unterstützen. Einfache Anführungszeichen können jedoch verwendet werden, wenn sie in doppelten Anführungszeichen enthalten sein müssen.

Zeichenfolgen können mit dem + Operator verkettet werden. Dies lässt keine Leerzeichen zwischen den Zeichenfolgen, dieses muss explizit hinzugefügt werden, wenn erforderlich:

```
a + b
a + " " + b
```

```
'Dies ist ein String mit doppelten Anführungszeichen. Dies ist ein String mit
einfachen Anführungszeichen.'
```

Man kann Zahlen und Zeichenketten auf ähnliche Weise verketten, nur müssen Zahlen explizit in Zeichenketten umgewandelt werden, indem die `str` Funktion verwendet wird:

```
a = 1
"x" + str(a)
```

```
'x1'
```

Die Python-Standardbibliothek enthält viele nützliche Funktionen im Zusammenhang mit Zeichenketten. Viele davon sind tatsächlich *Methoden* und sollten als `s.method()` aufgerufen werden, wobei `s` die Zeichenkette und `method` der Name der Methode ist. Zum Beispiel konvertiert `upper` eine Zeichenkette in Großbuchstaben und `split` teilt sie in Teile auf:

```
country = "kleine wörter, große wirkung"
country.upper()
```

```
'KLEINE WÖRTER, GROSSE WIRKUNG'
```

```
sentence = "Wann wird es mal wieder richtg Sommer"
sentence.split()
```

```
['Wann', 'wird', 'es', 'mal', 'wieder', 'richtg', 'Sommer']
```

Manchmal möchten Sie einen langen String definieren. Solche mehrzeiligen Strings können mit dreifachen Anführungszeichen definiert werden:

```
message = """
Hat der alte Hexenmeister
Sich doch einmal wegbegeben!
Und nun sollen seine Geister
Auch nach meinem Willen leben.
"""
print(message)
```

```

Hat der alte Hexenmeister
Sich doch einmal wegbegeben!
Und nun sollen seine Geister
Auch nach meinem Willen leben.
```

## Funktionen

Funktionen in Python verhalten sich sehr ähnlich wie in anderen traditionellen Programmiersprachen. Funktionen können mit dem Schlüsselwort `def` definiert werden, gefolgt vom Funktionsnamen und der Liste der Argumente in Klammern. Dies wird durch einen Doppelpunkt und einen eingerückten Funktionskörper fortgesetzt.

```
def add(x, y):
    z = x + y
    return z
```

```
add(4,5)
```

```
9
```

Wird in einer Funktion kein Wert explizit mit `return` zurückgeben, so geben sie implizit den speziellen leeren Wert `None` zurück. Dies kann zu Fehlern im folgenden Code führen, wenn man eine Rückgabe erwartet.

Python-Funktionen unterstützen auch Standardwerte bei Parametern zu deklarieren, zum Beispiel:

```
def multiply(x, y=2):
    return x*y ## Ergebnisrückgabe nicht vergessen
```

```
multiply(4, 3) # funktioniert
```

```
12
```

```
multiply(4)    # funktioniert auch
```

```
8
```

## Zusammengesetzte Datentypen

Die Basisversion von Python enthält drei sehr nützliche Sammlungsdatenstrukturen: *Listen*, *Wörterbücher* und *Sets*. Diese sind in vielerlei Hinsicht ähnlich zu Java-Sammlungen oder C++ Containern, jedoch viel einfacher zu verwenden. Sie sind auch sehr weit verbreitet und daher ein wesentlicher Bestandteil des Basiswissens von Python.

**Liste** (`list`) sind Sequenzen von Objekten, mit den folgenden Eigenschaften:

- *geordnet* da die Objekte in einer bestimmten Reihenfolge gespeichert sind und man die Fragen „ist *a* vor *b* in der Liste?“ beantworten kann.
- *positional* da die Elemente eine wohldefinierte numerische Position haben (Index) und Aufgaben wie „setze ‚w‘ an Position 2“ wohl definiert sind.
- *veränderbar* da Elemente in einer Position überschrieben werden können und neue Elemente an einer beliebigen Position hinzugefügt werden können.

**Tupel** (`tupel`) sind Sequenzen von Objekten die Listen sehr ähnlich sind, allerdings *nicht veränderbar* sind. Das ist in machen Aufgaben sinnvoll (z.B. Vektoren).

**Wörterbuch** (`dict`), auch *Maps* (dt. Abbildung) genannt, sind Sammlungen von Schlüssel-Wert-Paaren. Sie sind:

- *über Schlüssel adressierbar* da die Werte einem Schlüssel zugewiesen sind und Aufgaben wie „weise ‚w‘ dem Schlüssel ‚k‘ zu“ wohl definiert sind.
- *geordnet nach Einfügereihenfolge* da Python-Dictionaries in moderner Python-Version die Reihenfolge der eingefügten Schlüssel beim Iterieren beibehalten. Diese Reihenfolge ist nützlich, ersetzt aber keine Sortierung nach Größe oder Alphabet.
- *veränderbar* da Werte, die einem Schlüssel zugewiesen sind, überschrieben werden können und neue Schlüssel-Wert-Paaren erstellt werden können.

**Mengen** (`set`) sind Sammlungen von eindeutigen Elementen, d.h. sie können nur eine einzelne Kopie jedes Elements enthalten, was nützlich ist, wenn man eindeutige Werte finden muss. Die Elemente eines Sets werden in keiner bestimmten Reihenfolge gespeichert, höchstwahrscheinlich in der Reihenfolge, die der Computer für praktisch hält.

- *ungeordnet* da die Werte in „beliebiger“ Reihenfolge vom Computer abgelegt werden (es werdem Hashes benutzt, die es erlauben, sehr schnell zu prüfen, ob ein Element enthalten ist).

- *nicht positional* da Aufgaben wie „gib das 2. Element des Sets“ zurück, nicht definiert sind und zu Fehlern führen.
- *veränderbar* da neue Werte dem Set hinzugefügt werden können, wenn sie noch nicht enthalten sind.

Es ist sehr sinnvoll sich mit diesen Datentypen vertraut zu machen, da sie auch beim Verarbeiten von Daten immer wieder gebraucht werden. So nutzt man `list` z.B. wenn man Datensätze sammelt, `set` wenn man sehr schnell prüfen möchte ob man z.B. Wörter schon kennt, oder `dict` wenn man z.B. Worthäufigkeit zählen will.

**Listen**

Listen sind geordnete Sammlungen, die prinzipielle alle Datentypen in Python enthalten können. Deshalb sind sie einer der beliebtesten Sammlungstype, da sie intuitiv, einfach zu handhaben, schnell, veränderbar und vielseitig einsetzbar sind.

Listen können mit eckigen Klammern erstellt werden:

```
e = []  # Lehre Liste
l = [1.0, 2, "a"]  # Liste mit 3 Elemente von unterschiedlicher Datentypen
l
```

```
[1.0, 2, 'a']
```

```
m = [1, 2, 3, 4, 5]  # Liste mit 5 Elemente des gleichen Datentyps
m
```

```
[1, 2, 3, 4, 5]
```

Listen können auch aus anderen Sammlungen und iterierbaren Objekten erstellt werden, indem man die `list`-Funktion verwendet:

```
n = list(range(5))
print(n)
```

```
[0, 1, 2, 3, 4]
```

**Indizierung und Slicing**

Listenelemente können mithilfe von Klammern zugegriffen werden. Listen in Python und andere Sammlungen verwenden eine 0-basierte Indizierung: Das erste Element hat den Index 0 (nicht 1 wie in unserer Umgangssprache).

```
l[0]  # Das erste Element der Variable `l`
```

```
1.0
```

```
m[1]  # Das zweite Element der Variable `m`
```

```
2
```

```
m[2] = -7  # Wir überschreiben das dritten Element der Variable `m`
print(m)
```

```
[1, 2, -7, 4, 5]
```

Python erlaubt auch negative Indizes, um schnell auf Elemente am Ende der Liste zuzugreifen

```
m[-1]  # Das letzte Element
```

```
5
```

Man kann auf mehr als ein Listen-Element zugreifen, das nennt man *Slicing*. Slicing wird mit der Konstruktion `[erstes:letztes]` durchgeführt, wobei `erstes` den ersten eingeschlossenen Index bedeutet und `letztes` den ersten nicht eingeschlossenen Index bedeutet. Also extrahiert beispielsweise `x[1:3]` die Elemente `x[1]` und `x[2]`; aber nicht `x[3]` da es nicht eingeschlossen ist (es ist auch zu beachten, dass `x[1]` das 2. Element ist):

```
m[1:4]  # 2., 3., und 4. Element
```

```
[2, -7, 4]
```

Man kann `first` und `last` im Slice weglassen. Wenn `first` weggelassen wird, nimmt Python das erste mögliche Element, und wenn `last` weggelassen wird, nimmt es das letzte mögliche Element. Also bedeutet `x[3:]` vom 4. bis zum letzten Element, und `x[:3]` bedeutet vom ersten bis zum 3. Element (das 4., mit Index 3, wird nicht eingeschlossen):

```
m[2:] # vom 3. bis zum Letzten
```

```
[-7, 4, 5]
```

```
m[:4] # vom 1. bis zum 4.
```

```
[1, 2, -7, 4]
```

Daher bedeutet `x[:]` dasselbe wie `x`, da alle Elemente vom ersten bis zum letzten enthalten sind.

Das Slicing funktioniert auch mit negativen Indizes, die in diesem Fall vom Ende aus gezählt werden:

```
m[-3:]  # Die letzten 3
m[:-2]  # Ohne die letzten 2 Elemente
```

```
[1, 2, -7]
```

Das Slicing akzeptiert auch ein optionales drittes Argument, die *Schrittweite*, nach dem zweiten Doppelpunkt:

```
m[0:5:2]  # nehme vom 1. bis ausschließlich dem 5. Element jedes 2. Element
```

```
[1, -7, 5]
```

Dies kann man nutzen, um eine Liste umzukehren indem man eine negative Schrittweite angibt. Dadurch können wir die Liste sehr einfach rückwärts bearbeiten, z.B. wenn man nach dem letzten Vorkommen eines Wertes sucht.

```
m[::-1]
```

```
[5, 4, -7, 2, 1]
```

Beachten Sie, dass wir die *ersten* und *letzten* Argumente ausgelassen haben und daher Python die ersten und letzten möglichen Werte ausgewählt hat, wobei berücksichtigt wurde, dass wir rückwärts gehen. Es hat also vom letzten Element begonnen und mit dem ersten Element geendet.

**Kombinieren von Listen**

Man kann einzelne Elemente zur Liste mit der `append`-Methode hinzufügen und zwei Listen mit dem +-Operator verketten. Hier ist ein Beispiel:

```
a = [1, 2, 3]
a.append(4)
a
```

```
[1, 2, 3, 4]
```

```
a + [5, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

Aber beachten Sie die Einschränkung: `append` fügt ein einzelnes Element hinzu. Wenn Sie etwas wie `a.append([5,6])` machen, fügt es immer noch ein einzelnes Element hinzu, in diesem Fall eine Liste, die 5 und 6 enthält. Daher wird das letzte Element der Liste eine weitere Liste sein:

```
a.append([5,6])
a
```

```
[1, 2, 3, 4, [5, 6]]
```

Wollen Sie mehrere Elemente hinzufügen nutzen Sie die Methode `extend`

```
a.extend([5,6])
a
```

```
[1, 2, 3, 4, [5, 6], 5, 6]
```

**Elemente löschen**

Mit dem `del`-Befehl können wir Elemente aus der Liste löschen. So können wir den fehlerhaften Eintrag `[5,6]` entfernen:

```
del a[-2]  # Wir löschen das vorletzte Element
print(a)
```

```
[1, 2, 3, 4, [5, 6], 6]
```

**Erstellen von Listen in einer Schleife**

Oft müssen wir einen Wert für jedes Element in einer Sammlung berechnen und alle Ergebnisse in einer einzigen Liste speichern. Zum Beispiel möchte man vielleicht sehen, wie viele Beobachtungen es in einer Reihe von Datendateien gibt oder wie viele Zutaten es in verschiedenen Rezepten gibt. Eine beliebte Lösung in solchen Fällen ist folgende: Zuerst eine leere Liste erstellen und dann über die Sammlung iterieren und den berechneten Wert an die Liste anhängen. Hier ist zum Beispiel Code, der eine Liste von Quadraten von Zahlen erstellt:

```
squares = []
for i in range(10):
    squares.append(i**2)
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Dies ist ein praktischer und häufig verwendeter Algorithmus. Allerdings ist er nicht besonders effizient und wird sehr langsam, wenn die Sammlung groß ist. Das Problem ist, dass die Listen

mit fester endlicher Länge erstellt werden und wenn Sie neue Elemente zur Liste hinzufügen, gehen Ihnen die vorab zugewiesenen Speicherplätze aus. Der Computer muss neuen Speicherplatz zuweisen und die früheren Daten an die neue Stelle kopieren. Aber für kleine Sammlungen funktioniert dieser Ansatz gut.

**Listen-Abstraktion**

Listen-Abstraktion ist eine elegante Möglichkeit, Listen in Python zu erstellen. Es ermöglicht, Listen auf einfache und kompakte Weise zu erstellen, indem eine Schleife und eine Bedingung in einer einzigen Zeile Code kombiniert werden. Hier ist ein einfaches Beispiel:

Der Syntax ist der folgende:

```
[ausdruck for variable in iterator]
```

Der `ausdruck` ist ein Python-Ausdruck, der einen Wert berechnet, in der Regel unter Verwendung der `variable` im Prozess. Die `variable` wird wiederum mittels Durchlaufen (Iteration) des `iterator` extrahiert. Zum Beispiel können wir eine Liste von Quadraten wie oben erstellen, indem wir iterieren.

Dieser Code erstellt eine neue Liste `squared_numbers`, in der jedes Element der ursprünglichen Liste `numbers` quadriert wird. Das Ergebnis wird dann ausgegeben. List comprehension ist eine leistungsstarke Technik, um den Code lesbarer und effizienter zu gestalten.

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = [x**2 for x in numbers]
print(squared_numbers)
```

```
[1, 4, 9, 16, 25]
```

Hier $i$ läuft von 0 bis 9, und für jedes $i$ wird $i^2$ der Liste hinzugefügt.

Natürlich können wir auch andere Datentypen verwenden, nicht nur Zahlen für Listenverständnis. Zum Beispiel erstellen wir hier eine Liste von nummerierten Fragen:

```
["Frage " + str(i) for i in range(1,5)]
```

```
['Frage 1', 'Frage 2', 'Frage 3', 'Frage 4']
```

Wir müssen auch nicht die Schleifenvariable im Ausdruck verwenden. So erstellt der folgen Aufruf ein Liste von Nullen.

```
[0 for i in range(5)]
```

```
[0, 0, 0, 0, 0]
```

### List Methoden

List-Objekte haben viele nützliche Methoden. Zum Beispiel ordnet `sort` die Liste in einer natürlichen Reihenfolge:

```
l = [1, 5, 3, 2]
l.sort()
l
```

```
[1, 2, 3, 5]
```

Beachten Sie, dass `sort`, wie die meisten anderen Methoden, *lokal* arbeitet, d.h. sie ändern die aktuelle Liste und geben keine neue Liste zurück. Dies ist eine häufige Quelle für Verwirrung und Fehler, zum Beispiel wenn man vergisst, dass `sort` in-place arbeitet und schreibt

```
l = [1, 5, 3, 2]
l1 = l.sort()
l1
```

Man erhält ein leeres Ergebnis. Dies liegt daran, dass die `sort` Methode die Liste direkt verändert und den speziellen leeren Wert `None` zurückgibt.

Es gibt jedoch eine Funktion `sorted`, die eine sortierte Liste zurückgibt, während die Originalliste unverändert bleibt:

```
l = [1, 5, 3, 2]
sorted(l)
l
```

```
[1, 5, 3, 2]
```

Das Vorhandensein ähnlicher Funktionen, von denen einige direkt im Speicher arbeiten und andere ein modifiziertes Objekt zurückgeben, ist eine häufige Ursache für Verwirrung bei Anfängern.

### Tupel

Python enthält auch eine listenähnliche Sammlung namens *Tupel*, die nicht *veränderbar* ist, d.h. man kann das bereits erstellte Tupel nicht ändern. Die Syntax ähnelt der von Listen, nur anstelle von eckigen Klammern verwendet sie Klammern. Zum Beispiel:

```
b = (0, 1, 2, 3, 4)
b
```

```
(0, 1, 2, 3, 4)
```

Die Indizierung funktioniert wie bei Listen.

```
b[1]
```

```
1
```

Man kann Tupel auch slicen.

```
b[:4]
```

```
(0, 1, 2, 3)
```

Man kann ein Tupel aber nicht ändern. Der folgende Code führ zu einem Fehler

```
b[3] = -1
```

```
TypeError: 'tuple' object does not support item assignment
?[31m---------------------------------------------------------------------------?
[39m
?[31mTypeError?[39m                                 Traceback (most recent
call last)
?[36mCell?[39m?[36m ?[39m?[32mIn[53]?[39m?[32m, line 1?[39m
?[32m----> ?[39m?[32m1?[39m b[?[32m3?[39m] = -?[32m1?[39m

?[31mTypeError?[39m: 'tuple' object does not support item assignment
```

Die spezielle Syntax für leere und ein-elementares Tupel ist:

```
empty = ()
one = (1,)
```

Beachten Sie das Komma nach `1` im Ein-Element-Tupel. Dies sagt Python, dass dies ein Tupel ist und nicht nur die Zahl eins in Klammern.

Tupel werden weit verbreitet verwendet, wenn nicht veränderbare Elemente erforderlich sind. Dazu gehören *dict*-Schlüssel, *set*-Elemente und andere Fälle, in denen das Objekt hashbar sein muss.[^Im Prinzip kann man auch den Hash-Code von veränderbaren Variablen berechnen. Das würde jedoch erfordern, dass der Computer über Änderungen an den Daten informiert ist und bei Bedarf den Hash-Code neu berechnet. Das ist machbar, aber ineffizient. Python löst dieses Dilemma auf eine Weise, dass es nur Hash-Codes für unveränderliche Variablen berechnet.]

Tupel sind auch beliebt, wenn eine Funktion mehrere Werte zurückgeben muss, oder für die Mehrfachzuweisung von Variablen und für die interaktive Ausgabe von mehreren Elementen. Bei

der Mehrfachzuweisung schreiben wir einfach ein Tupel von Variablennamen auf der linken Seite des Zuweisungszeichens und ein Tupel von Werten auf der rechten Seite. Zum Beispiel:

```
name, age, address = "Gao", 44, "Mountain Alley 22"
print(name)
print(age)
print(address)
```

```
Gao
44
Mountain Alley 22
```

Beim Drucken auf einer interaktiven Konsole oder in einer Notebook-Zelle ziehen wir es vielleicht vor, die `print`-Funktion nicht zu verwenden. Aber wir können immer noch mehrere Werte als Tupel ausgeben:

```
data = [1,2,3,4]
# print both min and max on a single line:
min(data), max(data)
```

```
(1, 4)
```

**Wörterbücher**

Wörterbücher, auch Dictionaries oder Map genannt, sind Datenstrukturen, die Werte unter einem Schlüssel speichern, z.B. die Bedeutung eines Wortes als Wert unter dem Wort als Schlüssel. Wörterbücher werden häufig verwendet, um Schlüssel-Wert-Paare zu verwalten (z.B. Nutzer-Passwort) oder komplexe Datenstrukturen zu erstellen.

Der Syntax zum Erstellen lautet: `{Schlüssel1: Wert1, Schlüssel2: Wert2, ...}`. Zum Beispiel können wir ein Wörterbuch der Quadrate von Zahlen erstellen:

```
squares = {0:1, 1:1, 2:4, 3:9, 4:16}
print(squares)
```

```
{0: 1, 1: 1, 2: 4, 3: 9, 4: 16}
```

Das Extrahieren von Werten basierend auf Schlüsseln ähnelt sehr dem Indizieren von Listen:

```
squares[2]
squares[4]
```

```
16
```

Wir können auch neue Schlüssel-Wert-Paare hinzufügen und die bestehenden überschreiben, indem wir eine ähnliche Syntax verwenden:

```
squares[5] = 25
squares[3] = 8  # lol :-)
squares
```

```
{0: 1, 1: 1, 2: 4, 3: 8, 4: 16, 5: 25}
```

Aber weder Schlüssel noch Werte müssen Zahlen sein. Diese können auch andere Datentypen sein, einschließlich komplexer. Hier ist ein Beispiel für die Verknüpfung von Städten mit geografischen Koordinaten:

```
cities = {"Shanghai": [31.228611, 121.474722],
          "Dhaka": [23.763889, 90.388889],
          "Bangkok": [13.7525, 100.494167]
          }
print(cities)
```

```
{'Shanghai': [31.228611, 121.474722], 'Dhaka': [23.763889, 90.388889],
'Bangkok': [13.7525, 100.494167]}
```

Schließlich hier ein Beispiel für eine komplexere Datenstruktur, die mit einer Liste erstellt wurde:

```
address = {"house":2,
           "street": "Justus-von-Liebig Weg",
           "city": "Rostock",
           "province":"Mecklenburg-Vorpommern",
           "zip":18059,
           "country":"Deutschland"}
```

**Schlüssel und Werte**

Man kann alle Schlüssel eines Wörterbuchs mit der Methode `keys` finden. Dies ist eine durchlaufbare Sammlung, die man in eine Liste oder eine andere Sammlung umwandeln oder einfach darüber iterieren kann. Das folgende Beispiel gibt einfach alle Schlüssel und Werte auf eine schöne Weise aus:

```
for key in address.keys():
    print(key, ": ", address[key])
```

```
house :  2
street :  Justus-von-Liebig Weg
city :  Rostock
```

```
province :  Mecklenburg-Vorpommern
zip :  18059
country :  Deutschland
```

### Sets

Die letzte Struktur, über die wir hier sprechen, ist die Menge. Sie modelliert die Menge im mathematischen Sinne, d.h. es handelt sich um eine ungeordnete Sammlung, die nur eine Kopie jedes Elements enthält. *Ungeordnet* bedeutet, dass beim Durchlaufen der Elemente eine unvorhersehbare Reihenfolge entsteht und es unterstützt auch keinen positionellen Zugriff.

Mengen werden oft dort verwendet, wo wir sicherstellen müssen, dass wir nur eine Kopie jedes Elements haben. Hier ist ein Beispiel zum Zählen eindeutiger Elemente in einer Liste:

```
x = [1, 2, 3, 2, 1, 2, 2, 5, 3]
s = set(x)
print("wir haben", len(s), "einzigartige Elemente:", s)
```

```
wir haben 4 einzigartige Elemente: {1, 2, 3, 5}
```

Sets unterstützen auch mathematische Mengenoperationen wie Vereinigung und Schnittmenge, man kann auch über die Elemente der Menge iterieren (sie ist *iterierbar*).

Wenn wir auf die Elemente der Menge positionell zugreifen müssen, können wir sie zurück in eine Liste umwandeln. Da die Menge nicht geordnet ist, möchten wir möglicherweise die resultierende Liste sortieren, um eine konsistente Reihenfolge zu haben.

```
l = list(s)
l.sort()
print(l)
```

```
[1, 2, 3, 5]
```

## Sprachkonstrukte

### if-elif-else

Die `if`-Anweisung funktioniert auf eine sehr vorhersehbare Weise, ähnlich wie in vielen anderen gängigen Sprachen:

```
x=1
if x > 0:
    print("positive")
elif x < 0:
    print("negative")
```

```
else:
    print("zero")
```

```
positive
```

`if` erfordert einen logischen Ausdruck. Wenn dieser wahr ist, wird der folgende eingerückte Block ausgeführt. Wenn er nicht wahr ist, wird die eventuelle `elif`-Bedingung überprüft (es können viele `elif`-Blöcke vorhanden sein), und schließlich wird der `else`-Block ausgeführt, sofern ein `else`-Block vorhanden ist.

Manchmal ist es nützlich, einen Block zu haben, der nichts tut. In diesem Fall kann man das `pass`-Statement verwenden.

```
if x < 0:
    pass
else:
    print("nicht negativ")
```

```
nicht negativ
```

Beachten Sie, dass es in der Regel besser ist, die logische Bedingung umzukehren und den else-Block wegzulassen.

**For-Schleifen**

For-Schleifen sind eine der beliebtesten Methoden, um über Sammlungen zu iterieren. Die einzige Anforderung ist, dass die Sammlung iterierbar ist, sie muss nicht geordnet sein (und noch mehr, es muss keine Sammlung sein, wie z.B. `range` keine Sammlung ist. Die Syntax ist leicht zu merken: `for _variable_ in _collection_:`. Dem Doppelpunkt folgt ein eingerückter Block, der Körper der For-Schleife.

Ein triviales Beispiel:

```
for i in range(3):
    print(i)
```

```
0
1
2
```

Denken Sie daran, dass die Sammlung keine Elemente desselben Typs enthalten muss. Wir können auch schreiben

```
for i in [1, 'a', True]:
    print(i)
```

```
1
a
True
```

Und schließlich ein Beispiel für funktionale Programmierung: Wir durchlaufen eine Liste von Funktionen und geben den Funktionswert bei 1 aus:

```
import math
for func in [math.sin, math.cos, math.sqrt]:
    print(func(1))
```

```
0.8414709848078965
0.5403023058681398
1.0
```

`import` ist der Python-Weg, um Bibliotheken (Module) zu laden, siehe den Abschnitt Module unten.

For-Schleifen sind ein praktisches Werkzeug für verschiedene Aufgaben. Oft müssen wir etwas basierend auf einer Anzahl von Elementen berechnen. In diesem Fall möchten wir das Ergebnis initialisieren (auch als *Akkumulator* bezeichnet) und es in einer For-Schleife aktualisieren, in der wir über all diese Elemente iterieren. Zum Beispiel können wir For-Schleifen verwenden, um Fakultäten zu berechnen (Produkt aller ganzen Zahlen bis zu einer bestimmten Zahl):

```
p = 1  # initialize p (accumulator)
## compute 10!
for i in range(1, 11):
    p = p*i
p
```

```
3628800
```

Und hier ist ein weiteres Beispiel: Kombinieren Sie alle Namen in einer Liste, damit wir einen einzelnen, durch Kommas getrennten String haben:

```
## flowers
names = ["viola glabella", "monothropa hypopithys", "lomatium utriculatum"]
s = ""  # initialize accumulator
for name in names:
    if s != "":
```

```
        s += ", "
    s += name
s
```

```
'viola glabella, monothropa hypopithys, lomatium utriculatum'
```

Hier muss der Code etwas anders funktionieren, je nachdem, ob wir mit dem ersten oder einem nachfolgenden Namen arbeiten, da nur die nachfolgenden Namen von `", "` vorangestellt werden.

Diese Beispiele oben sind trivial, und es gibt einfachere Möglichkeiten, die Ergebnisse mit Standard-Python-Bibliotheken zu erzielen. Aber dieser Ansatz ist allgemeiner und kann in vielen Fällen angewendet werden, in denen solche Bibliotheksfunktionen nicht existieren.

## Bibliotheken (Module)

Die Basisversion von Python lädt automatisch eine minimalistische Reihe von Funktionen. Beispielsweise werden gängige mathematische Operatoren wie die Quadratwurzel oder Sinus nicht geladen, und es werden keine Betriebssystemfunktionen wie Verzeichnislisten geladen. Diese Funktionalität muss explizit durch das Importieren der entsprechenden *Module* (Bibliotheken) geladen werden.

Es gibt verschiedene Möglichkeiten, Module zu importieren. Erstens kann man das gesamte Modul laden und die Syntax `Modulname.Funktion` verwenden, um auf die Funktion zuzugreifen. Zum Beispiel:

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

```
1.4142135623730951
```

Dies importiert das Modul *math*, das eine Vielzahl von mathematischen Operationen enthält. Danach können wir diese Funktionen mit dem Präfix `math.` verwenden. Der Vorteil dieses Ansatzes ist, dass man im Code sofort sehen kann, wo bestimmte Funktionen herkommen. Allerdings erfordert es mehr Tipparbeit und längere Namen.

Alternativ können wir nur die benötigten Funktionen importieren und diese ohne Präfix verwenden:

```
from math import sqrt, sin, pi
sqrt(2)
sin(pi/2)
```

```
1.0
```

Dies führt zu kürzerem und saubererem Code, aber manchmal kann es schwierig sein zu erraten, wo die entsprechenden Funktionen definiert sind. Die Funktion `sin` kann entweder im `math`-Modul, im `numpy`-Modul oder vielleicht einfach an anderer Stelle in derselben Code-Datei definiert sein.

Es ist auch möglich, das Modul beim Import umzubenennen, ein sehr beliebter Ansatz, wenn man mit Bibliotheken mit längeren Namen arbeitet:

```
import math as m
m.sqrt(2)
m.sin(m.pi/2)
```

```
1.0
```

## Bibliographie