3. Eine informelle Einführung in Python

Zum Ausprobieren der folgenden Beispiele muss alles eingetippt werden, was auf die Eingabeaufforderung (`>>>` oder `...`) folgt. Zeilen die nicht mit einer solchen Eingabeaufforderung beginnen, sind Ausgaben des Interpreters. Steht die sekundäre Eingabeaufforderung `...` allein in einer Zeile, dann muss eine Leerzeile eingegeben werden. Dadurch werden mehrzeilige Befehle abgeschlossen.

Viele Beispiele in diesem Tutorial enthalten Kommentare - auch solche, die im interaktiven Modus eingegeben werden. Kommentare beginnen in Python mit einer Raute `#` und gelten bis zum Ende der physikalischen Zeile. Ein Kommentar kann am Anfang einer Zeile beginnen oder im weiteren Verlauf einer Zeile, allerdings nicht innerhalb eines Zeichenkettenliterals. Eine Raute innerhalb eines Zeichenkettenliterals ist einfach nur eine Raute. Da Kommentare dazu dienen, Code zu erklären und von Python nicht interpretiert werden, können sie beim Abtippen weggelassen werden.

Ein paar Beispiele:

```
# Das ist der erste Kommentar
spam = 1                    # und dies ist der zweite Kommentar
                               # ... und jetzt ein dritter!
string = "# Dies ist kein Kommentar."
```

3.1. Benutzung von Python als Taschenrechner

Wir wollen ein paar einfache Python-Befehle ausprobieren: Starte den Interpreter und warte auf die primäre Eingabeaufforderung, `>>>`.

3.1.1. Zahlen

Der Interpreter kann wie ein Taschenrechner eingesetzt werden: Man kann einen Ausdruck eingeben und der Interpreter berechnet das Ergebnis. Die Syntax für solche Ausdrücke ist einfach: Die Operatoren `+`, `-`, `*` und `/` wirken genauso wie in den meisten anderen Sprachen (beispielsweise Pascal oder C); Klammern können zur Gruppierung benutzt werden. Zum Beispiel:

```
>>> 2 + 2
4
>>> # Dies ist ein Kommentar
... 2 + 2
4
>>> 2 + 2  # und dies ist ein Kommentar in derselben Zeile wie Code
4
>>> (50 - 5 * 6) / 4
5.0
>>> 8 / 5 # Brüche gehen nicht verloren, wenn man Ganzzahlen teilt
1.6
```

Anmerkung: Möglicherweise liefert die letzte Berechnung bei dir nicht genau das gleiche Ergebnis, weil sich Ergebnisse von Fließkommaberechnungen von Computer zu Computer

unterscheiden können. Im weiteren Verlauf wird noch darauf eingegangen, wie man die Darstellung bei der Ausgabe von Fließkommazahlen festlegen kann. Siehe Fließkomma-Arithmetik: Probleme und Einschränkungen für eine ausführliche Diskussion von einigen Feinheiten von Fließkommazahlen und deren Repräsentation.

Um eine Ganzzahldivision auszuführen, die ein ganzzahliges Ergebnis liefert und den Bruchteil des Ergebnisses vernachlässigt, gibt es den Operator `//`:

```
>>> # Ganzzahldivision gibt ein abgerundetes Ergebnis zurück:
... 7 // 3
2
>>> 7 // -3
-3
```

Das Gleichheitszeichen (`'='`) wird benutzt um einer Variablen einen Wert zuzuweisen. Danach wird kein Ergebnis vor der nächsten interaktiven Eingabeaufforderung angezeigt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Ein Wert kann mehreren Variablen gleichzeitig zugewiesen werden:

```
>>> x = y = z = 0  # Null für x, y und z
>>> x
0
>>> y
0
>>> z
0
```

Variablen müssen “definiert” sein, bevor sie benutzt werden können, sonst tritt ein Fehler auf. Diese Definition geschieht durch eine Zuweisung:

```
>>> # Versuche eine undefinierte Variable abzurufen
... n
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Python bietet volle Unterstützung für Fließkommazahlen. Werden Operanden verschiedener Zahlentypen durch einen Operator verknüpft, dann werden ganzzahlige Operanden in Fließkommazahlen umgewandelt:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Auch komplexe Zahlen werden unterstützt. Der Imaginärteil wird mit dem Suffix `j` oder `J` angegeben. Komplexe Zahlen mit einem Realanteil, der von Null verschieden ist, werden als `(real+imagj)` geschrieben oder können mit der Funktion `complex(real, imag)` erzeugt werden.

```
>>> 1j * 1J
```

```
(-1+0j)
>>> 1j * complex(0, 1)
(-1+0j)
>>> 3 + 1j * 3
(3+3j)
>>> (3 + 1j) * 3
(9+3j)
>>> (1 + 2j) / (1 + 1j)
(1.5+0.5j)
```

Komplexe Zahlen werden immer durch zwei Fließkommazahlen repräsentiert, dem Realteil und dem Imaginärteil. Um diese Anteile einer komplexen Zahl z auszuwählen, stehen `z.real` und `z.imag` zur Verfügung.

```
>>> a = 1.5 + 0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Die Konvertierungsfunktionen in Fließkommazahlen und Ganzzahlen (**`float()`** , **`int()`**) stehen für komplexe Zahlen nicht zur Verfügung. Man kann `abs(z)` verwenden, um den Betrag einer komplexen Zahl (als Fließkommazahl) zu berechnen, oder `z.real` , um den Realteil zu erhalten:

```
>>> a = 3.0 + 4.0j
>>> float(a)
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

Im interaktiven Modus wird der zuletzt ausgegebene Ausdruck der Variablen `_` zugewiesen. Die ist besonders hilfreich, wenn man den Python-Interpreter als Taschenrechner einsetzt

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
>>>
```

Die Variable `_` sollte man so behandeln, als wäre sie schreibgeschützt und ihr nicht explizit einen Wert zuweisen. Dadurch würde eine unabhängige lokale Variable mit demselben Namen erzeugt, die die eingebaute Variable `_` mit ihrem speziellen Verhalten verdeckt.

3.1.2. Zeichenketten (Strings)

Außer mit Zahlen kann Python auch mit Zeichenketten umgehen, die auf unterschiedliche Weise darstellbar sind. Sie können in einfache oder doppelte Anführungszeichen eingeschlossen werden:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Ja," hat er gesagt.'
'"Ja," hat er gesagt.'
>>> "\"Ja,\" hat er gesagt."
'"Ja," hat er gesagt.'
>>> '"Isses nich\'," sagte sie.'
'"Isses nich\'," sagte sie.
```

Der Interpreter gibt das Ergebnis von Zeichenketten-Operationen auf die gleiche Weise aus, wie sie eingegeben werden: Innerhalb von Anführungszeichen und mit durch Backslashes maskierten Anführungszeichen oder anderen seltsamen Zeichen, um den exakten Wert wiederzugeben. Die Zeichenkette wird von doppelten Anführungszeichen eingeschlossen, wenn sie ein einfaches Anführungszeichen, aber keine doppelten enthält, sonst wird sie von einfachen Anführungszeichen eingeschlossen. Die Funktion `print()` produziert eine lesbarere Ausgabe.

Es gibt mehrere Möglichkeiten, mehrzeilige Zeichenkettenliterale zu erzeugen, zum Beispiel durch Fortsetzungszeilen, die mit einem Backslash am Ende der physikalischen Zeile anzeigen, dass die nächste Zeile die logische Fortsetzung der aktuellen ist:

```
hello = "Dies ist eine ziemlich lange Zeichenkette,\n\
die mehrere Zeilen Text enthält und wie man sie auch in C schreiben würde.\n\
    Achtung: Leerzeichen am Anfang haben eine Bedeutung\
für die Darstellung."

print(hello)
```

Zu beachten ist, dass Zeilenumbrüche immer noch in den Zeichenkette mit Hilfe von `\n` eingebettet werden müssen. Der auf den Backslash folgende Zeilenumbruch gehört allerdings nicht mit zur Zeichenkette. Die vom Beispiel erzeugte Ausgabe sieht so aus :

```
Dies ist eine ziemlich lange Zeichenkette,
die mehrere Zeilen Text enthält und wie man sie auch in C schreiben würde.
    Achtung: Leerzeichen am Anfang haben eine Bedeutung für die Darstellung.
```

Zeichenketten können auch mit einem Paar von dreifachen Anführungszeichen umgeben werden: `"""` oder `'''`. Zeilenenden müssen nicht hierbei escaped werden, sondern werden in die Zeichenkette übernommen. Deshalb wird im folgende Beispiel das erste Zeilenende escaped, um die unerwünschte führende Leerzeile zu vermeiden:

```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
```

```
     -H hostname               Hostname to connect to
""")
```

Das erzeugt folgende Ausgabe:

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Wenn wir den Zeichenkettenliteral zu einem “raw”-String machen, wird `\n` nicht in einen Zeilenumbruch umgewandelt; auch der Backslash am Ende und das Zeilenumbruch-Zeichen im Quellcode sind Teil der Zeichenkette. Das Beispiel:

```
hello = r"Dies ist eine ziemlich lange Zeichenkette,\n\
die mehrere Zeilen Text enthält und wie man sie auch in C schreiben würde."

print(hello)
```

führt zu folgender Ausgabe:

```
Dies ist eine ziemlich lange Zeichenkette,\n\
die mehrere Zeilen Text enthält und wie man sie auch in C schreiben würde.
```

Zeichenketten können mit dem `+`-Operator verkettet und mit `*` wiederholt werden:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Zwei Zeichenkettenliterale nebeneinander werden automatisch miteinander verknüpft. Die erste Zeile im obigen Beispiel hätte also auch `word = 'Help' 'A'` lauten können. Das funktioniert allerdings nur mit zwei Literalen, nicht mit beliebigen String-Ausdrücken:

```
>>> 'str' 'ing'             #Das ist ok
'string'
>>> 'str'.strip() + 'ing'   #Das ist ok
'string'
>>> 'str'.strip() 'ing'     #Das ist ungültig
 File "<stdin>", line 1, in ?
   'str'.strip() 'ing'
                 ^
SyntaxError: invalid syntax
```

Zeichenketten können indiziert werden, wobei das erste Zeichen einer Zeichenkette wie in C den Index 0 hat (“nullbasierte Zählung”). Es gibt keinen speziellen Zeichentyp (wie `char` in C) — ein Zeichen ist einfach eine Zeichenkette der Länge eins. Wie in der Programmiersprache Icon können Teile einer Zeichenkette mittels Slice-Notation (Ausschnittschreibweise) festgelegt werden. Angegeben werden zwei Indizes getrennt durch einen Doppelpunkt (`:`).

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Slice-Indizes haben nützliche Standardwerte: Wird der erste Index ausgelassen, beginnt der Ausschnitt mit dem ersten Zeichen der Zeichenkette (Index 0), wird der zweite Index ausgelassen, reicht der Ausschnitt bis zum Ende der Zeichenkette

```
>>> word[:2]    # Die ersten beiden Zeichen
'He'
>>> word[2:]    # Alles außer den ersten beiden Zeichen
'lpA'
```

Im Unterschied zu einem C-String kann ein Python-String nicht verändert werden — Zeichenketten sind unveränderbar (immutable). Der Versuch, einer indizierten Position einer Zeichenkette etwas zuzuweisen, führt zu einer Fehlermeldung

```
>>> word[0] = 'x'
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: 'str' object does not support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: 'str' object does not support slice assignment
```

Stattdessen erzeugt man einfach eine neue Zeichenkette mit dem kombinierten Inhalt

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

Noch ein Beispiel: `s[:i] + s[i:]` entspricht `s`.

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Werden bei der Slice-Notation Indizes angegeben, die die tatsächliche Länge einer Zeichenkette überschreiten, führt dies nicht zu einer Fehlermeldung: Ein zu großer zweiter Index wird durch die Länge der Zeichenkette ersetzt und Ausschnitte, die keine Zeichen enthalten, liefern eine leere Zeichenkette zurück.

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Indizes können auch negative Zahlen sein — dann wird von rechts nach links gezählt. Zum Beispiel:

```
>>> word[-1]     # Das letzte Zeichen
'A'
>>> word[-2]     # Das vorletzte Zeichen
'p'
>>> word[-2:]    # Die letzten zwei Zeichen
'pA'
>>> word[:-2]    # Alles außer den letzten beiden Zeichen
'Hel'
```

Achtung: -0 ist dasselbe wie 0. Das heißt, die Zählung beginnt ganz normal von links!

```
>>> word[-0]     # (da -0 gleich 0)
'H'
```

Das automatische Kürzen bei Verwendung von Indizes, die außerhalb der tatsächlichen Länge der Zeichenkette liegen, funktioniert allerdings nur bei der Slice-Notation, nicht beim Zugriff auf ein einzelnes Zeichen mittels Indexschreibweise:

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # Fehler
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Man kann sich die Indizes beim Slicing so vorstellen, dass sie zwischen den Zeichen liegen — wobei die linke Ecke des ersten Zeichens den Index 0 hat und die rechte Ecke des letzten Zeichens einer n Zeichen langen Zeichenkette den Index n. Ein Beispiel

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

Die erste Zahlenreihe gibt die Position der Indizes 0...5 im String an, die zweite Reihe die entsprechenden negativen Indizes. Der Ausschnitt von i bis j besteht aus allen Zeichen zwischen den Positionen, die durch i beziehungsweise j gekennzeichnet werden.

Bei Verwendung von nicht-negativen Indizes entspricht die Länge des dadurch festgelegten Ausschnitts der Differenz der beiden Indizes, sofern beide innerhalb der tatsächlichen Grenzen der Zeichenkette liegen. Die Länge von `word[1:3]` ist zum Beispiel 2.

Die eingebaute Funktion **`len()`** gibt die Länge eines Strings zurück:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Siehe auch

Sequence Types

Zeichenketten gehören zu den Sequenztypen und verfügen über alle Operationen, die von diesen Typen unterstützt werden.

String Methods

Strings verfügen über eine große Zahl an Methoden für grundlegende Transformationen und Suche.

String Formatting

Informationen über Stringformatierung mit **`str.format()`** sind hier zu finden.

Old String Formatting Operations

Die alten Formatierungsoperationen, die aufgerufen werden, wenn Strings und Unicodestrings die linken Operanden des `%`-Operators sind, werden hier ausführlich beschrieben.

3.1.3. Über Unicode

Beginnend mit Python 3.0 unterstützen alle Strings Unicode.

Unicode hat den Vorteil, dass es eine Ordnungszahl für jedes Zeichen in jedem Schriftstück bereitstellt, das in modernen und antiken Texten benutzt wird. Davor waren nur 256 Ordnungszahlen für Schriftzeichen möglich. Texte waren typischerweise an eine Codepage gebunden, die die Ordnungszahlen den Schriftzeichen zugeordnet hat. Das führte zu großer Verwirrung, vor allem im Hinblick auf Internationalisierung von Software (üblicherweise `i18n` — `'i'` + 18 Zeichen + `'n'`) . Unicode löst diese Probleme, indem es eine Codepage für alle Schriftzeichen definiert.

Will man spezielle Zeichen in eine Zeichenketten einbinden, erreicht man das durch die Verwendung von Pythons Unicode-Escape-Schreibweise. Das folgende Beispiel zeigt wie:

```
>>> 'Hello\u0020World !'
'Hello World !'
```

Die Escapesequenz `\u0020` gibt an, dass das Unicodezeichen mit der Ordnungszahl 0x0020 (das Leerzeichen) an der gegebenen Position eingefügt werden soll.

Andere Zeichen werden interpretiert, indem ihre jeweiligen Ordnungszahlen direkt als Unicode-Ordnungszahlen benutzt werden. Hat man Zeichenkettenliterale in der normalen Latin-1-Kodierung, die in vielen westlichen Ländern benutzt wird, dann entsprechen die ersten 256 Zeichen von Unicode denselben Zeichen der Latin-1-Kodierung.

Neben diesen Standardkodierungen stellt Python eine ganze Reihe anderer Möglichkeiten bereit, Unicodestrings zu erstellen, sofern man die verwendete Kodierung kennt.

Zur Konvertierung von Zeichenketten in Bytefolgen stellen Stringobjekte die Methode **`encode()`** bereit, die den Namen der Kodierung als Argument entgegennimmt, und zwar möglichst in Kleinbuchstaben.

```
>>> "Äpfel".encode('utf-8')
b'\xc3\x84pfel'
```

3.1.4. Listen

Python kennt viele zusammengesetzte Datentypen (compound data types), die zur Gruppierung unterschiedlicher Werte verwendet werden können. Die flexibelste davon ist die Liste (list): Eine Liste von Werten (Elemente), die durch Kommas getrennt und von

eckigen Klammern eingeschlossen werden. Listenelemente müssen nicht alle denselben Typ haben.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Ebenso wie die Indizierung bei Zeichenketten ist auch die Listenindizierung nullbasiert — das erste Element hat also den Index 0. Auch das von Zeichenketten bekannte Slicing sowie die Verkettung und Vervielfachung `+` bzw. `*` sind mit Listen möglich

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

Alle Slicing-Operationen geben eine neue Liste zurück, die die angeforderten Elemente enthält. Das bedeutet, dass die folgende Operation eine flache Kopie (shallow copy) der Liste `a` zurückgibt:

```
>>> a[:]
['spam', 'eggs', 100, 1234]
```

Im Unterschied zu Zeichenketten sind Listen allerdings veränderbar (mutable), so dass es möglich ist, innerhalb einer Liste Veränderungen vorzunehmen

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Selbst Zuweisungen an Slices sind möglich. Dadurch kann man die Länge einer Liste verändern oder sie sogar ganz leeren

```
>>> # Ein paar Elemente ersetzen:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Ein paar entfernen:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Ein paar einfügen:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # (Eine Kopie von) sich selbst am Anfang einfügen:
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Die Liste leeren: Alle Elemente durch eine leere Liste  ersetzen
>>> a[:] = []
```

```
>>> a
[]
```

Die eingebaute Funktion `len()` lässt sich auch auf Listen anwenden:

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

Es ist auch möglich Listen zu verschachteln (nest), das heißt, Listen zu erzeugen, die andere Listen enthalten. Ein Beispiel:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
```

Man kann auch etwas ans Ende einer Liste hängen:

```
>>> p[1].append('xtra')
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Beachte, dass im letzten Beispiel `p[1]` und `q` wirklich auf dasselbe Objekt zeigen! Wir kommen später zur Objektsemantik zurück.

3.2. Erste Schritte zur Programmierung

Natürlich kann man Python für kompliziertere Aufgaben verwenden, als nur zwei und zwei zu addieren. Beispielsweise lassen sich die ersten Glieder der Fibonacci-Folge folgendermaßen erzeugen:

```
>>> # Fibonacci-Folge:
... # Die Summe der letzten beiden Elemente ergibt das nächste
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Dieses Beispiel stellt ein paar neue Eigenschaften vor.

- Die erste Zeile enthält eine Mehrfachzuweisung (multiple assignment): Die Variablen `a` und `b` bekommen gleichzeitig die neuen Werte 0 und 1. In der letzten Zeile wird sie erneut eingesetzt, um zu zeigen, dass zuerst alle Ausdrücke auf der

rechten Seite ausgewertet werden, bevor irgendeine Zuweisung vorgenommen wird! Die Ausdrücke auf der rechten Seite werden von links nach rechts ausgewertet.

- Die `while` Schleife wird solange ausgeführt, wie die Bedingung (hier: `b < 10`) wahr ist. In Python wie in C ist jede von Null verschiedene Zahl wahr (True), Null ist unwahr (False). Die Bedingung kann auch ein String- oder Listenwert sein, eigentlich sogar jede Sequenz. Alles mit einer von Null verschiedenen Länge ist wahr, leere Sequenzen sind unwahr. Die Bedingung im Beispiel ist ein einfacher Vergleich. Die normalen Vergleichsoperatoren werden wie in C geschrieben: `<` (kleiner als), `>` (größer als), `==` (gleich), `<=` (kleiner oder gleich), `>=` (größer oder gleich) und `!=` (ungleich).
- Der Schleifenrumpf ist eingerückt (indented): Durch Einrückung wird in Python eine Gruppierung vorgenommen. In der interaktiven Eingabeaufforderung muss man Tabs oder Leerzeichen für jede eingerückte Zeile eingeben. In der Praxis wird man kompliziertere Codestücke mit einem Texteditor vorbereiten und alle vernünftigen Editoren haben eine Möglichkeit, um automatisch einzurücken. Wenn eine zusammengesetzte Anweisung (compound statement) interaktiv eingegeben wird, muss eine Leerzeile darauf folgen, um anzuzeigen, dass sie komplett ist, da der Parser nicht erahnen kann, wenn man die letzte Zeile eingegeben hat. Beachte, dass jede Zeile in einem zusammengehörigen Block gleich eingerückt sein muss.
- Die Funktion `print()` gibt den Wert des Ausdrucks aus, der ihr übergeben wurde. Die Ausgabe unterscheidet sich bei Mehrfachausdrücken, Fließkommazahlen und Zeichenketten von der Ausgabe, die man erhält, wenn man die Ausdrücke einfach so eingibt (wie wir es vorher in den Taschenrechnerbeispielen gemacht haben). Zeichenketten werden ohne Anführungszeichen ausgegeben, und bei Angabe mehrere Argumente wird zwischen je zwei Argumenten ein Leerzeichen eingefügt. So lassen sich einfache Formatierungen vornehmen, wie das Beispiel zeigt

```
>>> i = 256 * 256
>>> print('Der Wert von i ist', i)
Der Wert von i ist 65536
```

Durch Verwendung des Schlüsselwortarguments end kann der Zeilenumbruch nach der Ausgabe verhindert oder die Ausgabe mit einem anderen String beendet werden.

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print(b, end=' ')
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```