2 Schnelleinstieg

2.1 Hallo Welt (Hello World)

Wir beginnen das Kennenlernen von Python mit einem simplen Programm, nämlich wie traditionell in den allermeisten Büchern zum Programmierenlernen mit »Hello World«, der Ausgabe eines Grußes auf der Konsole. Das haben wir schon in etwas komplizierterer Form in der Einleitung gesehen. Wir wollen es aber weiter vereinfachen und damit unsere Entdeckungsreise starten.

Python bietet einen interaktiven Kommandozeileninterpreter, auch Read-Eval-Print-Loop (REPL) genannt. Dieser interaktive Kommandozeileninterpreter erlaubt es uns, kleinere Python-Codeschnipsel auszuprobieren. Das erleichtert das schrittweise Erlernen.

Wir starten den interaktiven Kommandozeileninterpreter mit dem Kommando `python` (Windows) bzw. `python3` (macOS) wie folgt:

```
$ python3
Python 3.11.2 (v3.11.2:878ead1ac1, Feb  7 2023, 10:02:41) [Clang
      13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more
      information.
>>>
```

Neben dem Hinweis auf die Version zeigt der Prompt >>> an, dass wir nun Python-Befehle eingeben können. Probieren wir es gleich einmal aus:

```
>>> print("Hello World from Python!")
Hello World from Python!
```

Herzlichen Glückwunsch zur ersten Programmausgabe mit Python. Die eingebaute Funktionalität `print()` ermöglicht Ausgaben auf der Konsole. Derartige Aktionen werden durch sogenannte Funktionen bereitgestellt, in diesem Fall durch die Funktion `print()`, der man in runden Klammern einen Text eingerahmt von Anführungszeichen übergibt.

Selbst wenn es in diesem Schnelleinstieg-Kapitel ein klein wenig anspruchsvoller wird, keine Sorge, Sie müssen noch nicht alle Details verstehen – wir werden diese im Verlauf dieses Buchs noch ausführlich besprechen. Jetzt geht es zunächst um die ersten Schritte und ein initiales Gespür. Tippen Sie die Beispiele einfach ab und wenn Sie sich sicher fühlen, dann variieren Sie diese auch gerne ein wenig.

Wir haben bereits einen netten Gruß auf der Konsole ausgeben können. Das war schon ein recht guter Anfang. Allerdings wäre es etwas eintönig, nur Texte ohne Variation auszugeben. Um den Gruß anzupassen oder Berechnungen ausführen zu können, lernen wir nun Variablen kennen.

Übrigens hätte man die Ausgabe sogar noch etwas kürzer erhalten können – allerdings ohne die wichtige Funktion `print()` kennenzulernen:

```
>>> "Hello World from Python!"
'Hello World from Python!'
```

2.2 Variablen und Datentypen

Stellen wir uns vor, wir wollten einige Eigenschaften einer Person mit Python verwalten, etwa Name, Alter, Gewicht, Wohnort, Familienstand usw. Dabei helfen uns Variablen. Diese dienen zum Speichern und Verwalten von Werten. Dabei gibt es unterschiedliche Typen (auch Datentypen genannt), etwa für Texte, Zahlen und Wahrheitswerte. Wir schauen uns dies zunächst einmal einführend an und erweitern dann unser Wissen Schritt für Schritt, beispielsweise um ergänzende Infos oder Aktionen mit den Variablen. Zum Einstieg betrachten und nutzen wir folgende Typen:

- `str` – Textuelle Informationen, wie z. B. "Hallo" oder 'Peter'. Stringwerte werden von doppelten oder einfachen Anführungszeichen eingeschlossen.
- `int` – Ganzzahlen, wie 123 oder –4711.
- `float` – Gleitkommazahlen mit Vor- und Nachkommastellen, wie 72,71 oder –1,357.
  Achtung: Weil Python die amerikanische Notation nutzt, werden als Dezimaltrenner Punkte statt Kommata verwendet. Somit muss es im Python-Programm `72.71` oder `-1.357` heißen.
- `bool` – Wahrheitswerte als wahr oder falsch, in Python: `True` oder `False`.

2.2.1 Definition von Variablen

In Python erzeugen wir eine Variable, indem wir zunächst den Namen und danach eine Wertzuweisung nach folgendem Muster (auch Syntax genannt) nutzen:

```
variablenname = wert
```

Schauen wir uns ein paar Beispiele an:

```
>>> age = 18
>>> the_answer = 41
>>> simple_pi = 3.1415
>>> name = "Michael"
>>> one_million = 1_000_000
>>> the_answer = 42
>>> is_valid = True
```

Für das Beispiel mit einer Million sehen wir die Angabe von Unterstrichen, die sich gut zur Separierung, hier von Tausendersegmenten, eignen. Zwar habe ich es für `the_answer` gerade nur angedeutet, aber einer Variablen kann im Programmverlauf mehrmals ein anderer Wert zugewiesen werden. Denken Sie etwa an einen Punktestand in einem Spiel, einen Temperaturverlauf pro Tag und eine Variable `temperature` oder aber die Modellierung der Tageszeit von Morgens, Mittags, Nachmittags,

Abends und Nachts. Für derart in sich abgeschlossene Wertebereiche existieren Aufzählungen, die wir in Abschnitt 6.5 kennenlernen werden.

Den aktuellen Wert einer Variablen können wir durch Eingabe des Namens in der Konsole abfragen bzw. auslesen.

```
>>> age
18
>>> name
'Michael'
>>> one_million
1000000
```

Alternativ können wir die bereits kurz vorgestellte Funktion `print()` nutzen, die einzelne Werte ausgibt:

```
>>> print(age)
18
>>> print(name)
Michael
>>> print(one_million)
1000000
```

Wir erkennen in beiden Fällen, dass die Simulation der Tausendertrennzeichen nur bei der Definition zum Tragen kommt, Python die Werte jedoch ohne diese Zeichen speichert.

2.2.2 Variablen und Typen

Tatsächlich sind die Typen von Variablen in Python nicht direkt sichtbar, aber sie lassen sich mit der eingebauten Funktion `type()` folgendermaßen abfragen:

```
>>> type(age)
<class 'int'>
>>> type(simple_pi)
<class 'float'>
>>> type(name)
<class 'str'>
>>> type(is_valid)
<class 'bool'>
```

Neben den kurz erwähnten Typnamen sehen wir weitere Informationen wie `class`, die hier nicht relevant sind und erst später in Kapitel 4 besprochen werden.

Dynamische Typisierung und Typwechsel

Python ist eine sogenannte dynamisch typisierte Sprache. Das bedeutet, dass sich zum einen der Typ ausgehend vom Wert ergibt und sich der Typ zum anderen sogar im Programmverlauf ändern kann, etwa von `float`

```
>>> age = 50.25
>>> type(age)
<class 'float'>
```

auf `str` durch eine entsprechende Zuweisung eines textuellen Werts:

```
>>> age = "Mittelalt"
>>> type(age)
<class 'str'>
```

Allerdings ist eine derartige Wiederverwendung kein guter Stil und kann zu Missverständnissen und Fehlverwendungen führen. Das gilt insbesondere, wenn im zeitlichen Verlauf statt einer Zahl ein Text gespeichert wird und somit der Typ wechselt.

Typumwandlung (Cast)

Gelegentlich sind Ganzzahlen statt Gleitkommazahlen als Ergebnis gewünscht. Liefert eine Berechnung jedoch Gleitkommazahlen, so können wir mit dem sogenannten ***Casting*** arbeiten: Durch die Angabe `typ(wert)`, wobei `typ` für `str`, `int` usw. steht, lässt sich ein Wert in den angegebenen Typ umwandeln. Nachfolgend nutzen wir `int`, um eine Gleitkommazahl in eine Ganzzahl zu verwandeln, wodurch die Nachkommastellen wie gewünscht abgeschnitten werden. Somit wird aus `3.1415` der Wert `3`:

```
>>> int(3.1415)
3
```

Bei diesen Casts ändert sich insbesondere der Typ. In unserem Beispiel war das Abschneiden der Nachkommastellen ein schöner und gewünschter Nebeneffekt.

2.2.3 Ausgaben mit `print()`

Schauen wir uns nun noch ein paar Details im Zusammenhang mit der Funktion `print()` an, die einzelne Werte ausgibt – mehrere textuelle Werte kann man mit + verknüpfen und mit `upper()` in Großbuchstaben wandeln:

```
>>> print(age)
18
>>> print("Hello " + name.upper())
Hello MICHAEL
>>>
```

Auf ein Detail möchte ich nochmals hinweisen: Bei `print()` handelt es sich um eine in Python integrierte sogenannte Funktion, man spricht auch von Built-in-Funktion. Diese dienen dazu, gewisse Aktionen standardisiert bereitstellen zu können. Funktionen schauen wir uns dann in Abschnitt 2.5 genauer an.

Besonderheit: Zahlen und Texte gemischt Bislang scheint die Ausgabe mit `print()` recht eingängig. Was passiert aber, wenn wir Zahlen und Texte gemischt ausgeben wollen? Das wäre etwa der Fall, wenn wir die meisten der bisher definierten Variablen in eine Konsolenausgabe integrieren wollen.

Betrachten wir zunächst ein vereinfachtes Beispiel bei dem wir – wie möglicherweise von anderen Programmiersprachen gewohnt – die auszugebenden Bestandteile mit + verbinden:

```
>>> anzahl = 3
>>> print("Bitte " + anzahl + " Mal klingeln!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Das führt zu einem Programmabbruch. Wir erkennen hier, dass nur textuelle Bestandteile mit + verknüpft werden können – ein simpler Cast mit `str()` löst das Problem:

```
>>> print("Bitte " + str(anzahl) + " Mal klingeln!")
Bitte 3 Mal klingeln!
```

Weil die Verknüpfung von textuellen Bestandteilen zur Ausgabe recht gebräuchlich ist, ist eine Alternative in Python integriert: An `print()` kann man eine kommaseparierte Folge von Werten übergeben, die dann automatisch in Strings umgewandelt und mit einem Abstand von einem Leerzeichen ausgegeben werden:

```
>>> print("Hello", name, "the answer is", the_answer, "and not",
      age)
Hello Michael the answer is 42 and not 18
```

Aber Moment: Wo sind denn die anfangs erwähnten Typen? Darauf kommen wir zurück, nachdem wir uns die Konsolenausgabe etwas genauer angeschaut haben.

Besonderheit: Separator Wenn die Werte nicht mit einem Abstand von einem Leerzeichen ausgeben werden sollen, sondern gegebenenfalls direkt nacheinander oder mit speziellen Trennzeichenfolgen, bietet `print()` den Parameter `sep`:

```
>>> print("Hello", name, "no_space", "in_between", sep="")
HelloMichaelno_spacein_between
>>> print("Important", "Message", "INFO", sep=" --- ")
Important --- Message --- INFO
```

Besonderheit: Zeilenumbruch Bislang werden die Ausgaben mit `print()` automatisch mit Zeilenumbruch beendet. Wenn man allerdings mehrere Werte hintereinander in einer Zeile darstellen möchte, dann ist das Standardverhalten unpraktisch. Dieses lässt sich mit Angabe eines leeren Endezeichens in Form von `end=""` abwandeln – dadurch erfolgt auf der Konsole kein Zeilenumbruch mehr, weshalb die Eingabeaufforderung (>>>) direkt nach dem letzten ausgegebenen Zeichen steht:

```
>>> print("Hello " + name, end="")
Hello Michael>>>
```

Varianten der formatierten Ausgabe

Zur Ausgabe von Werten mit `print()` müssen diese in eine textuelle Form überführt werden. Das geschieht automatisch, wenn man die zuvor gezeigte kommaseparierte Variante nutzt. Alternativ kann man die Bestandteile durch einen Aufruf der Python-Standardfunktionalität `str()` in einen String wandeln und dann mit + verknüpfen. Dabei muss man aber auf die korrekte Angabe der Leerzeichen zum Abstand achten.

Egal welche der Varianten man wählt, alle haben so ihre Tücken. Als Abhilfe bietet Python zur formatierten Aufbereitung diverse Möglichkeiten.

```
>>> print("Hello %s the answer is %d and not %d" % (name,
      the_answer, age))
Hello Michael the answer is 42 and not 18
>>>
>>> print("Hello {} the answer is {} and not {}".format(name,
      the_answer, age))
Hello Michael the answer is 42 and not 18
>>>
>>> print(f"Hello {name} the answer is {the_answer} and not {age}")
Hello Michael the answer is 42 and not 18
```

Zunächst sehen wir die Variante mit %, dann `{}` in Kombination mit `format()` sowie schließlich die sogenannten f-Strings, wo man benannte Platzhalter in einem String nutzt, dem ein `f` vorangestellt ist – Details zu diesen Möglichkeiten erfahren Sie später in Abschnitt 3.1.3.

2.2.4 Bezeichner (Variablennamen)

Ohne es explizit zu erwähnen, haben wir uns bei der Benennung von Variablen an ein paar Regeln gehalten. Zunächst einmal muss jede Variable (und auch die später vorgestellten Funktionen, Methoden und Klassen) durch einen eindeutigen Namen, auch Identifier oder Bezeichner genannt, gekennzeichnet werden.

Bei der Benennung sollte man folgende Regeln und Hinweise beachten:

- Namen sollten mit einem Buchstaben beginnen[1] – Ziffern sind als erstes Zeichen eines Variablennamens nicht erlaubt.
- Namen können danach aus einem beliebigen Mix aus Buchstaben (auch Umlauten), Ziffern und _ bestehen, dürfen aber keine Leerzeichen enthalten.
- Bestehen Namen aus mehreren Wörtern, dann ist es guter Stil, die sogenannte SnakeCase-Schreibweise zu verwenden, bei der jedes neue Wort getrennt durch _ startet, etwa `arrival_time` oder `estimated_duration`.
- Die Groß- und Kleinschreibung von Namen spielt eine Rolle: `arrival_Time` und `arrival_time` bezeichnen unterschiedliche Variablen. Allerdings sollten per Konvention Namen von Variablen bevorzugt kleingeschrieben werden.
- Namen dürfen nicht mit den in Python vordefinierten und im Anhang aufgelisteten Schlüsselwörtern übereinstimmen, somit sind `class` oder `True` keine gültigen Namen – die Begriffe `public_transport` oder `winter` hingegen sind erlaubt.

Übrigens spielt es für Python keine Rolle, ob Sie sehr kurze (`i`, `m`, `p`), kryptische (`dpy`, `mtr`) oder aber gut verständliche Namen (`days_per_year`, `max_table_rows`) nutzen. Für Sie und das Verständnis beim Nachvollziehen eines Python-Programms macht das aber einen himmelweiten Unterschied. Natürlich sind kurze Variablennamen wie etwa `x` und `y` zur Bezeichnung von Koordinaten vollkommen verständlich, aber was sagt beispielsweise ein einzelner Buchstabe `m` aus? Schauen wir uns ein Beispiel an, das Ihnen das Gesagte verdeutlichen soll:

```
>>> # Gut verständliche Namen
>>> minutes_per_hour = 60
>>> days_per_year = 365
>>>
>>> # Nicht wirklich verständlich, was die Abkürzungen bedeuten
>>> m = 60
>>> dpy = 365
```

[1] Tatsächlich dürfen Namen auch mit _ beginnen, jedoch ist das eher selten nützlich.

```
>>> # ACHTUNG: ziemlich schwierig unterscheidbar
>>> arrival_Time = "16:50"
>>> arrival_time = 16.50
```

Die letzten beiden Variablendefinitionen sind ziemlich ungünstig, da man sich beim Unterscheiden schwertut. Dadurch kommt es vermutlich leichter zu Fehlverwendungen.

Im Listing sehen wir sogenannte Kommentare. Damit kann man erklärende Zusatzinformationen hinterlegen. In diesem Fall werden mit # einzeilige Kommentare eingeleitet. Alles, was nach dem # bis zum Ende der Zeile folgt, wird bei der Ausführung von Python ignoriert (vgl. Abschnitt 2.7).

2.3 Operatoren im Überblick

Operatoren beschreiben Aktionen (Operationen) zwischen Variablen und/oder Werten. Die Addition ist ein Beispiel, nämlich für den Operator +. Das lässt sich wunderbar mit der Python-Kommandozeile nachprüfen:

```
>>> sum1 = 200 + 50
>>> sum2 = sum1 + 500
>>> sum3 = sum2 + 750
>>> print("Summen: 1:", sum1, "/ 2:", sum2, "/ 3:", sum3)
Summen: 1: 250 / 2: 750 / 3: 1500
```

Generell bietet Python folgende Varianten von Operatoren:

- Arithmetische Operatoren
- Zuweisungsoperatoren
- Vergleichsoperatoren
- Logische Operatoren
- Bit-Operatoren – Diese sind nur für ganz spezielle Anwendungsfälle von Relevanz und werden in diesem Buch deshalb nicht weiter behandelt.

2.3.1 Arithmetische Operatoren

Die arithmetischen Operatoren sind uns größtenteils aus der Schule bekannt. Neben den Grundrechenarten Addition, Subtraktion, Multiplikation und Division gibt es noch die Modulo-Berechnung sowie die Potenzierung. Tabelle 2–1 bietet einen Überblick.

Beispiele für Grundrechenarten

Betrachten wir einfache Beispiele für diese Operationen, exemplarisch für Ganzzahlen als Parameter. Beachtenswert dabei ist insbesondere, dass die einfache Division eine Gleitkommazahl als Ergebnis liefert:

```
>>> print(9 - 7)
2
>>> print(7 + 2)
9
>>> print(7 * 2)
14
>>> print(15 / 4)
3.75
```

Schauen wir uns ergänzend die in anderen Sprachen (oft) unbekannten Operatoren an, nämlich ** zur Potenzierung und // zur Ganzzahldivision:

```
>>> print(2 ** 8)
256
>>> print(10 // 7)
1
>>> print(1 + 6 * 7 - 7 // 6)
42
```

Das Ergebnis der letzten Berechnung könnte Sie vielleicht überraschen. Überlegen wir kurz, um es zu verstehen: In Python gelten genau wie in der Mathematik die Vorrangregeln, also Punkt- vor Strichrechnung. Dadurch kommt es zu folgender Auswertung: $1+(6*7)-1$. Damit wir keine Nachkommastellen erhalten, nutzen wir mit // die Ganzzahldivision.

Tab. 2–1: Arithmetische Operatoren

Operator	Name	Beschreibung und Beispiel
+	Addition	Addiert zwei Werte: `x + y`, z. B. `2 + 5 = 7`
-	Subtraktion	Subtrahiert zwei Werte: `x - y`, z. B. `9 - 2 = 7`
*	Multiplikation	Multipliziert zwei Werte: `x * y`, z. B. `2 * 7 = 14`
**	Potenzierung	Potenziert zwei Werte (x^y): `x "* y`, z. B. `2 ** 8 = 256`
/	Division	Dividiert zwei Werte: `x / y`, z. B. `15 / 4 = 3.75`
//	Ganzzahl-division	Dividiert zwei Werte ohne Rest: `x // y`, z. B. `9 // 7 = 1`
%	Modulo	Berechnet den Rest der Division zweier Werte: `x % y`, z. B. `9 % 7 = 2`

Spezialfall: Division durch 0

Bei der Division erinnern wir uns vielleicht an den Spezialfall der verbotenen Teilung durch 0. Das ist in der Mathematik nicht erlaubt bzw. liefert den Wert ∞ (unendlich). Mit diesem Wert kann man aber nicht mehr weiterrechnen. Auch Python verbietet eine derartige Division und reagiert darauf mit dem Auslösen eines `ZeroDivisionError`. Das Thema Fehlerbehandlung wird später in Kapitel 9 vertieft.

```
>>> 7.0 / 0.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division by zero
>>> 0.0 / 0.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division by zero
```

Hier reicht uns erst einmal das Wissen, dass nach einer solchen Fehlersituation die Ausführung eines Python-Programms gestoppt wird. In der Kommandozeile hingegen können Sie nach einem solchen Fehler problemlos weitere Kommandos eintippen.

Modulo-Operator

Schauen wir uns für einige Zahlen noch die Modulo-Berechnung an, die den Divisionsrest ermittelt:

```
>>> 5 % 1
0
>>> 5 % 2
1
>>> 5 % 3
2
>>> 5 % 4
1
>>> 5 % 5
0
```

Anstelle der hier gewählten Kurzform können wir natürlich auch erst Zahlen einer Variablen zuweisen und danach dann die Operationen ausführen:

```
>>> five = 5
>>> remainder = 9 % five
>>> remainder
4
```

Übrigens ...

Der Modulo-Operator arbeitet auch auf Gleitkommazahlen. Wenn man beispielsweise ermitteln möchten, welche Restzeit sich bei Etappen von 45 Minuten und einer Fahrtzeit von 6,5 Stunden ergibt, dann kann der Modulo-Operator gute Dienste leisten:

```
>>> 6.5 % 0.75
0.5
```

2.3.2 Zuweisungsoperatoren

Bei den Zuweisungsoperatoren denkt man natürlich zunächst an die einfache Zuweisung mit dem Operator =, den wir nun schon einige Male genutzt haben und der sich ziemlich natürlich erschließt: Er weist einer Variablen, hier `clicks_per_hour`, den rechts hinter dem = stehenden Wert, in diesem Fall `1234567`, zu:

```
>>> clicks_per_hour = 1234567
```

Nehmen wir nun die Variable `x` und die Wertzuweisung von 100 als Beispiel:

```
>>> x = 100
```

Um einen Wert, etwa 10, hinzuzufügen, können wir Folgendes schreiben:

```
>>> x = x + 10
>>> x
110
```

Kurzschreibweisen

Manchmal möchte man die Addition eines Werts kompakter als beispielsweise

```
>>> x = x + 10
```

notieren. Dazu existiert als Kurzform der Additionszuweisungsoperator +=:

```
>>> x += 10
>>> x
120
```

Für diese Kurzschreibweise gibt es noch die im Anschluss weiter unten demonstrierten praxisrelevanten Varianten +=, -=, *=, **=, /=, //= und %=, die vorab in der folgenden Tabelle aufgeführt sind.

Tab. 2–2: Zuweisungsoperatoren

Operator	Name	Beschreibung
`+=`	Addition	Addiert zwei Werte: `x += y` ⇒ `x = x + y`
`-=`	Subtraktion	Subtrahiert zwei Werte: `x -= y` ⇒ `x = x - y`
`*=`	Multiplikation	Multipliziert zwei Werte: `x *= y` ⇒ `x = x * y`
`**=`	Potenzierung	Potenziert zwei Werte (x^y): `x **= y` ⇒ `x = x ** y`
`/=`	Division	Dividiert zwei Werte: `x /= y` ⇒ `x = x / y`
`//=`	Ganzzahl-division	Dividiert zwei Werte ohne Rest: `x //= y` ⇒ `x = x // y`
`%=`	Modulo	Berechnet den Rest der Division zweier Werte: `x %= y` ⇒ `x = x % y`

Viele der Operationen verdeutlicht folgendes Beispiel:

```
>>> counter = 0
>>> counter += 100
>>> counter
100
>>> counter -= 10
>>> counter
90
>>> counter /= 9
>>> counter
10.0
>>> counter *= 7
>>> counter
70.0
>>> counter %= 9
>>> counter
7.0
>>> counter **= 2
>>> counter
49.0
```

2.3.3 Vergleichsoperatoren

Aus der Mathematik kennen wir diverse Operatoren, um Werte miteinander zu vergleichen. Exakt so funktioniert das auch in Python. Man möchte beispielsweise als wahr oder falsch abbilden können, ob zwei Werte gleich oder größer gleich sind. Für derartige Aussagen dienen die Wahrheitswerte `True` oder `False` vom Typ `bool`.

Tab. 2–3: Vergleichsoperatoren

Operator	Name	Beschreibung
==	gleich	`x == y`
!=	ungleich	`x != y`
>	größer	`x > y`
>=	größer gleich	`x >= y`
<	kleiner	`x < y`
<=	kleiner gleich	`x <= y`

Schauen wir uns wieder eine Abfolge von Aktionen zur Verdeutlichung an:

```
>>> 7 == 8
False
>>> 7 != 8
True
>>> 7 < 8
True
>>> 7 <= 8
True
>>> 7 > 8
False
>>> 7 >= 8
False
```

Wir können selbstverständlich Variablen vom Typ `bool` basierend auf den Ergebnissen erzeugen und abhängig vom Wert gewisse Aktionen ausführen. Dazu werden wir in Kürze Fallunterscheidungen und bedingte Ausführungen einsetzen.

```
>>> is_equal = 7 == 42
>>> is_equal
False
>>> age = 18
>>> younger_than30 = age < 30
>>> younger_than30
True
```

2.3.4 Logische Operatoren

Um einfache Bedingungen zu prüfen, haben wir gerade die Vergleichsoperatoren kennengelernt. Oftmals soll nicht nur eine, sondern es müssen mehrere Bedingungen erfüllt sein, oder eine aus mehreren. Mitunter muss auch die Bedingung invertiert werden: Aus größer gleich 18 (`>= 18`) wird kleiner 18 (`< 18`). Mithilfe der logischen Operatoren lassen sich diese Aktionen in Python formulieren. Die Operatoren `and` und `or` arbeiten auf zwei booleschen Werten (`bool`), der Operator `not` auf einem.

Tab. 2–4: Logische Operatoren

Operator	Name	Beschreibung
`and`	UND	`True`, wenn beide Bedingungen erfüllt sind, ansonsten `False`
`or`	ODER	`True`, wenn mindestens eine der Bedingungen erfüllt ist, ansonsten `False`
`not`	NEGATION	`True`, wenn die Bedingung nicht erfüllt ist, ansonsten `False`

Schauen wir uns wieder eine Abfolge von Aktionen zur Verdeutlichung an. Zunächst definieren wir die zwei Ganzzahlvariablen `age` und `points` und kombinieren dann einige Abfragen – dabei sehen wir, dass bei `and` beide Bedingungen erfüllt sein müssen, damit `True` geliefert wird, und bei `or` zumindest eine der beiden:

```
>>> age = 27
>>> points = 127
>>>
>>> age > 20 and points > 100
True
>>> age > 20 and points > 200
False
>>>
>>> age > 30 or points > 100
True
>>> age > 20 or points > 200
True
>>> age > 30 or points > 200
False
>>>
>>> not points > 500
True
```

Mehrere Verknüpfungen

Es lassen sich nicht nur einzelne Werte miteinander verknüpfen, sondern auch ganze Ausdrücke, z. B. zur Prüfung der Möglichkeit, ein weiteres Bier in einer Bar zu bestellen. Dies soll gegeben sein, wenn das Alter >= 18 ist und man zudem noch genug Bares hat oder der Sitznachbar die Runde übernimmt:

```
one_more_beer = (age >= 18) and (credit > 0 or paid_by_neighbour)
```

Besonderheit bei Verknüpfungen

Wenn man eine Variable auf zwei Grenzen prüfen möchte, dann lässt sich das mit zwei Vergleichen kombiniert mit `and` lösen:

```
>>> age = 22
>>> age >= 18 and age <= 65
True
```

In Python gibt es noch eine sehr praktische Besonderheit, die gerade bei Vergleichen auf untere und obere Grenzen recht hilfreich und deutlich besser lesbar ist:

```
>>> 18 <= age <= 65
True
```