4 Datentypen und Variablen

4.1 Einführung

Sie glauben, weil Sie bereits in Sprachen wie C und C++ programmiert haben, dass Sie bereits genug über Datentypen und Variablen wissen? Vorsicht ist geboten! Sie wissen sicherlich viel, aber wahrscheinlich nicht genug, wenn es um Python geht. Deshalb lohnt es sich auf jeden Fall, hier weiterzulesen. Es gibt gravierende Unterschiede zwischen Python und anderen Programmiersprachen in der Art, wie Variablen behandelt werden. Vertraute Datentypen wie Ganzzahlen (Integer), Fließkommazahlen (floating point numbers) und Strings (Zeichenketten) sind zwar in Python vorhanden, aber auch hier gibt es wesentliche Unterschiede zu C/C++ und Java. Dieses Kapitel ist also sowohl für Anfänger als auch für fortgeschrittene Programmierende zu empfehlen.

Eine Variable ist ein Name bzw. ein Bezeichner, mit dem man auf ein Objekt zugreifen kann.[1] Ein Objekt steht zwar an einem bestimmten Speicherplatz, aber in Python spricht und denkt man nicht in physikalischen Speicherplätzen. Eine Variable ist nicht einem festen Objekt oder Speicherplatz zugeordnet. Während des Programmablaufs können einem Variablennamen neue beliebige Objekte zugewiesen werden, d.h. die Variable referenziert nach jeder Zuweisung in den meisten Fällen einen neuen Speicherort. Diese Objekte können beliebige Datentypen sein. Eine Variable referenziert also ein Objekt in Python.

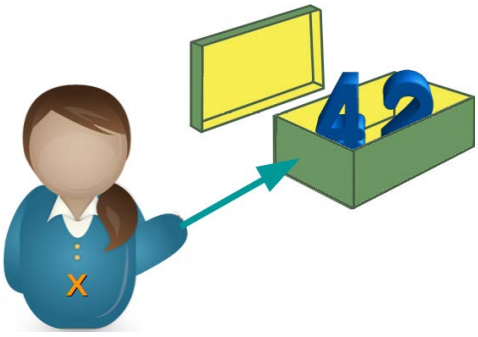


Bild 4.1 Variablen sind Referenzen auf Objekte.

[1] In den meisten Programmiersprachen ist es so, dass eine Variable einen festen Speicherplatz bezeichnet, in dem Werte eines bestimmten Datentyps abgelegt werden können. Während des Programmlaufs kann sich der Wert der Variable ändern, aber die Wertänderungen müssen vom gleichen Typ sein. Also kann man nicht in einer Variable zu einem bestimmten Zeitpunkt eine Integer-Zahl gespeichert haben und dann diesen Wert durch eine Fließkommazahl überschreiben. Ebenso ist der Speicherort der Variablen während des gesamten Laufs konstant, kann also nicht mehr geändert werden.

Im Folgenden kreieren wir eine Variable x in der interaktiven Python-Shell, indem wir ihr einfach den Wert 42 unter Verwendung eines Gleichheitszeichens zuweisen:

```
>>> x = 42
```

Man bezeichnet dies als Zuweisung oder auch als Definition einer Variablen. Genau genommen müsste man sagen, dass wir ein Integer-Objekt „42" erzeugen und es mit dem Namen x referenzieren.

Anmerkung: Obige Anweisung darf man nicht als mathematisches Gleichheitszeichen sehen, sondern als „der Variablen x wird der Wert 42 zugewiesen", d.h. der Inhalt von x ist nach der Zuweisung 42. Man kann die Anweisung also wie folgt „lesen": „x soll sein 42".

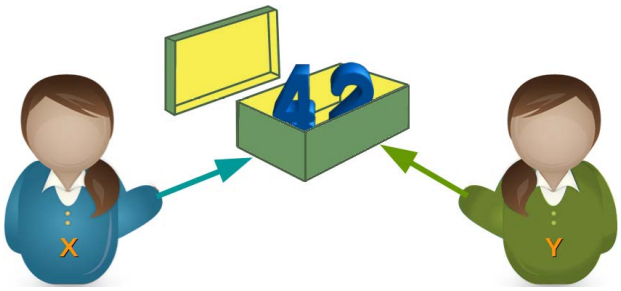
Wir können nun diese Variable zum Beispiel benutzen, indem wir ihren Wert mit der Funktion `print` ausgeben lassen:

```
>>> print("Wert von x: ", x)
Wert von x:  42
```

Wir können sie aber auch auf der rechten Seite einer Zuweisung verwenden:

```
>>> y = x
```

Beide Variablen zeigen nun auf das gleiche Objekt, d.h. das `int`-Objekt 42:



Woher wissen wir oder noch besser, wie kann man beweisen, dass beide Variablen auf das gleiche Objekt zeigen? Dazu bietet sich die `id`-Funktion an. Dies ist eine Funktion, die für ein Objekt die Identität eines Objektes zurückgibt. Erhält man für zwei Variablennamen die gleiche Identität, so weiß man, dass sie das gleiche Objekt referenzieren. Wir benutzen nun `id`, um die obige Behauptung zu beweisen:

```
>>> x = 42
>>> id(x)
9786176
>>> y = x
>>> id(y)
9786176
```

Der Ergebniswert von `id` interessiert in diesem Fall nicht. Wir möchten nur wissen, ob wir für beide Variablen die gleichen Werte erhalten:

```
>>> x = 42
>>> y = x
>>> id(x) == id(y)
True
```

Mit dem Ausdruck `id(x) == id(y)` können wir also prüfen, ob die beiden Variablen das gleiche Objekt referenzieren. Für einen solchen Test bietet Python auch den Operator `is`. Liefert `is` True, handelt es sich um das gleiche Objekt:

```
>>> pi1 = 3.141592653589793
>>> pi2 = 3.141592653589793
>>> pi1 is pi2  #  the same as id(pi1) == id(pi2)
False

>>> pi1 = 3.141592653589793
>>> pi2 = pi1
>>> pi1 is pi2
True
```

Den Operator `is` darf man keinesfalls mit dem Gleichheitsoperator == verwechseln. Mit == kann man überprüfen, ob Objekte gleich sind, was aber nicht impliziert, dass die Objekte identisch sind.[2] Ein Beispiel aus dem täglichen Leben kann dies illustrieren. Wenn jemand sagt: „Das selbe Auto ist schon wieder vorbeigefahren.", dann meint diese Person, dass es sich physikalisch um das gleiche Auto handelt, also gleiches Nummernschild beispielsweise. Wenn jemand sagt „Das ist das gleiche Auto, wie ich es habe!", dann meint die Person nur, dass sie das gleiche Modell besitzt. Im folgenden Code sehen wir, dass s1 und s2 sowohl gleich sind als auch das identische Objekt referenzieren:

```
>>> s1 = "Der Unterschied zwischen 'das Gleiche' und 'dasselbe'"
>>> s2 = s1
>>> s1 == s2
True
>>> s1 is s2
True
```

Nun sehen wir, dass Gleichheit nicht immer Identität impliziert:

```
>>> s2 = "Der Unterschied zwischen 'das Gleiche' und 'dasselbe'"
>>> s1 is s2
False
>>> s1 == s2
True
```

Bisher stand meist nur eine einzelne Variable rechts vom Gleichheitszeichen. Auf der rechten Seite einer Zuweisung kann aber auch ein beliebiger, zu berechnender Ausdruck stehen, dessen Ergebnis dann der Variablen auf der linken Seite zugewiesen wird. Ein solcher Ausdruck kann auch andere Variablennamen enthalten. Diesen muss jedoch zuvor ein Wert zugewiesen worden sein. Das bedeutet, dass anschließend die Variable y auf das Ergebnis der Addition des alten y-Wertes und 36 zeigt.

```
>>> a = 3
>>> b = 1
>>> c = a + b
```

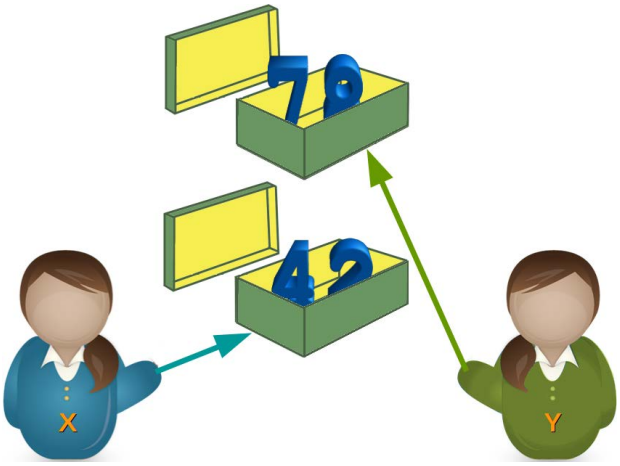
[2] In Gedanken kann man in den folgenden Beispielen auch den Begriff „Objekt" durch „Wert" ersetzen. Wir wollten von Anfang an die korrekten Begriffe, also „Objekt", verwenden.

```
>>> c
4
```

Wie man im nächsten Beispiel sehen kann, ist es möglich, eine Variable gleichzeitig auf der linken und rechten Seite einer Zuweisung zu benutzen.

```
>>> x = 42
>>> y = x
>>> y = y + 36
>>> y
78
```

Für die Schreibweise „`y = y + 36`" verwendet man üblicherweise die sogenannte erweiterte Zuweisung „`y += 36`".



Vom Ergebnis her gesehen sind die beiden Möglichkeiten identisch. Allerdings gibt es einen Implementierungsunterschied. Bei der erweiterten Zuweisung muss die Referenz der Variablen y nur einmal ausgewertet werden.[3]

Erweiterte Zuweisungen gibt es auch für die meisten anderen Operatoren wie „-", „*", „/", „**", „//" (ganzzahlige Division) und „%" (Modulo).

4.2 Variablennamen

Wenn wir uns beim Programmieren Variablennamen ausdenken, muss man zweierlei beachten: Zum einen muss der Name den syntaktischen Anforderungen der Programmiersprache genügen, und zum anderen muss er den üblichen Gepflogenheiten und Konventionen entsprechen.

4.2.1 Gültige Variablennamen

Variablennamen müssen mit einem Buchstaben oder Unterstrich „_" beginnen. Die folgenden Zeichen dürfen sich aus einer beliebigen Folge von Buchstaben, Ziffern und dem

[3] Bei Datentypen wie Listen kann es zu extremen Laufzeitproblemen kommen, wenn man keine erweiterte Zuweisung verwendet. Darauf gehen wir jedoch erst später ein.

Unterstrich zusammensetzen. Variablennamen sind case sensitive.[4] Dies bedeutet, dass Python zwischen Groß- und Kleinschreibung unterscheidet:

```
>>> person_42 = "Marvin"
>>> Person_42 = "Arthur"
>>> print(person_42)
Marvin
>>> print(Person_42)
Arthur
```

Zu den gültigen Buchstaben gehören aber nicht nur „lateinische" Buchstaben, sondern auch alle anderen Unicode-Buchstaben, wie beispielsweise kyrillische, griechische und so weiter.

4.2.2 Konventionen für Variablennamen

Man bevorzugt in der Python-Community Kleinschreibung bei Variablennamen, z.B. `minimum` statt `Minimum`. Außerdem werden Variablennamen, die aus mehreren Wörtern bestehen, mittels Unterstrich (_) separiert, also beispielsweise `minimale_breite`. Variablennamen mit Binnenversalien – bei Programmierenden besser als „camel case" oder „CamelCase" bekannt – sind in der Python-Welt nicht beliebt: `MinimaleBreite` oder `minimaleBreite`.

Ansonsten zeichnet sich ein guter Programmierstil dadurch aus, dass man möglichst sprechende Variablennamen verwendet. Also beispielsweise `aktueller_kontostand` oder `maximale_beschleunigung` statt nichtssagender Namen wie `ak` oder `mb`.

4.3 Datentypen

4.3.1 Ganze Zahlen

Ganze Zahlen werden in der Informatik üblicherweise als Integer bezeichnet. Es handelt sich dabei um die Zahlen ... -3, -2, -1, 0, 1, 2, 3, ...

Während Integer-Zahlen in den meisten Programmiersprachen durch einen Maximalwert begrenzt sind, sind sie in Python unbegrenzt, d.h. sie können beliebig groß bzw. beliebig klein werden. Die Variable `unsigned_long_max` bezeichnet in der folgenden interaktiven Python-Session den maximalen Integer-Wert in C für „unsigned long". Ein Wert, der in C für diesen Datentyp nicht überschritten werden kann. Wir sehen, dass wir in Python diesen Wert sogar ohne Fehler beispielsweise quadrieren können.

```
>>> unsigned_long_max = 4294967295
>>> unsigned_long_max * unsigned_long_max
18446744065119617025
>>> x = 787647382988789087878782366665654342334143008909100065 4
```

[4] Für diesen Begriff gibt es keine deutsche Übersetzung.

```
>>> x * x
620388399929088197813485589212087821899160651152524666622241420288301
↪ 9101976159306894814859277968375310284277 16
```

Bei Integer-Zahlen muss man beachten, dass sie nicht mit einer 0 beginnen dürfen, wenn es sich um eine Zahl im Dezimalsystem handeln soll:[5]

```
>>> 42
42
>>> 042
  File "<stdin>", line 1
    042
      ^
SyntaxError: leading zeros in decimal integer literals are not
↪ permitted; use an 0o prefix for octal integers
```

Die führende Null wird benötigt, wenn man Binär-, Oktal- oder Hexadezimalzahlen schreiben möchte. Literale für Binärzahlen beginnen mit der Sequenz `0b` oder `0B` und werden dann von der eigentlichen Binärzahl gefolgt. Die Binärzahl `101010` schreibt man also wie folgt:

```
>>> x = 0b101010
>>> x
42
```

Man sieht, dass dies keine Auswirkung auf die Interndarstellung der Zahl hat.

Literale für Oktalzahlen beginnen mit der Sequenz `0o` oder `0O` und werden dann von der eigentlichen Oktalzahl gefolgt:

```
>>> x = 0o10
>>> x
8
```

Literale für Hexadezimalzahlen beginnen mit der Sequenz `0x` oder `0X` und werden dann von der eigentlichen Hexzahl gefolgt:

```
>>> x = 0x10
>>> x
16
>>> x = 0x1A
>>> x
26
```

Mit den Funktionen `hex`, `bin`, `oct` kann man eine Integer-Zahl in einen String wandeln, der der Stringdarstellung der Zahl in der entsprechenden Basis entspricht:[6]

```
>>> x = hex(19)
>>> x
'0x13'
>>> type(x)
```

[5] Die Fehlermeldung im folgenden Beispiel gilt ab Python 3.8. Vorher wurde der Fehler "SyntaxError: invalid token" erhoben.

[6] Strings, also Zeichenketten, haben wir noch nicht eingeführt. Diese Umwandlung haben wir hier nur der Vollständigkeit wegen eingeführt!

```
<class 'str'>
>>> bin(65)
'0b1000001'
>>> oct(65)
'0o101'
>>> oct(0b101101)
'0o55'
```

4.3.2 Fließkommazahlen

Fließkommazahlen[7] entsprechen dem Datentyp `float` in Python. Es handelt sich dabei um Zahlen der Art 2.34, 27.87878 oder auch 3.15e2:

```
>>> x = 2.34
>>> y = 3.14e2
>>> y
314.0
```

Achtung: In deutschsprachigen Ländern werden die Nachkommastellen im täglichen Leben mit einem Komma eingeleitet. So schreibt man beispielsweise 1,99, wenn ein Artikel 1,99 € kostet. In Python und in anderen Programmiersprachen wird statt des Kommas immer ein Punkt, der sogenannte Dezimalpunkt, benutzt! Wie man aus obigem Beispiel ersehen kann, entspricht die Zahl „3.14e2" dem Ausdruck $3.14 \cdot 10^2$.

4.3.3 Zeichenketten

Wir werden im nachfolgenden Kapitel weiter auf Strings eingehen.

4.3.4 Boolesche Werte

Bei den booleschen Werten handelt es sich um einen Datentyp, der nur die zwei Werte True[8] (wahr) oder False (falsch) annehmen kann. Allerdings entspricht das im Prinzip den numerischen Werten 1 für True und 0 für False. Damit lässt sich mit booleschen Werten auch „numerisch" rechnen, auch wenn das nicht immer gutem Programmierstil entspricht:

```
>>> x = True
>>> x * 4
4
```

Für zwei boolesche Variablen x und y gilt Folgendes:

[7] Sie werden manchmal auch als Gleitpunktzahlen bezeichnet.
[8] Ein beliebter Anfangsfehler besteht darin, dass man nicht auf die Großschreibung der beiden Werte achtet!

Tabelle 4.1 Logisches UND und ODER

Operator	Erklärung
not y	Negierung von y
x and y	Logisches UND von x und y
x or y	Logisches ODER von x und y

Beispielanwendungen:

```
>>> x = True
>>> not x
False
>>> y = False
>>> x and y
False
>>> x or y
True
>>> x and not y
True
```

4.3.5 Komplexe Zahlen

Python bietet einen Datentyp `complex` für komplexe Zahlen, den man bei den meisten Programmiersprachen vergeblich sucht. Allerdings würden ihn die meisten Anwender von Python wohl auch nicht vermissen. Wenn Sie komplexe Zahlen nicht kennen oder nichts mit ihnen zu tun haben, können Sie das Folgende gerne überspringen. Mathematisch gesehen erweitern die komplexen Zahlen die reellen Zahlen derart, dass die Gleichung $x^2 + 1 = 0$ lösbar wird. In der Mathematik werden komplexe Zahlen meist in der Form $a + b \cdot i$ dargestellt, wobei a und b reelle Zahlen sind und i die imaginäre Einheit ist. In Python benutzt man, der Konvention in der Elektrotechnik folgend, ein „j" als imaginäre Einheit.

```
>>> x = 3 + 4j
>>> y = 2 - 4.5j
>>> x + y
(5-0.5j)
>>> x * y
(24-5.5j)
```

4.3.6 Operatoren

Eine Übersicht über die gängigsten Operatoren bietet die Tabelle 4.2. Zwei von diesen Operatoren werden wir nun im Detail vorstellen, nämlich „//" und „%":

„//" bezeichnet die ganzzahlige Division, d.h. `11 // 4` ergibt nicht den exakten Float-Wert `2.75` , sondern den Integer-Wert `2`, d.h. es wird immer auf die nächste ganzzahlige Inte-

Tabelle 4.2 Erklärung zu Operatoren

Operator	Erklärung
x + y	Summe von x und y
x - y	Differenz von x und y
x * y	Produkt von x und y
x / y	Quotient von x und y
x // y	Ganzzahlige Division
x % y	Modulo- oder Restdivision
abs(x)	Betrag von x
x ** y	Potenzieren, also x hoch y

ger „abgerundet". Dies gilt allerdings nur für den Fall, dass beide Operanden ganze Zahlen (`int`) sind! Ist mindestens einer der Operanden eine float-Zahl, wird auch auf die nächste ganzzahlige Integer-Zahl abgerundet, aber dann das Ergebnis in `float` gewandelt:

```
>>> 8 // 3
2
>>> 11 // 3
3
>>> 12 // 3
4
>>> 12.0 // 3
4.0
```

Mit `%` lässt sich der Rest bei der Integerdivison bestimmen: So ergibt `11 % 4` den Wert 3 und `10 % 4` den Wert 2.

```
>>> 8 % 3
2
>>> 9 % 3
0
>>> 8.0 % 3
2.0
```

Es gilt folgender Zusammenhang zwischen der ganzzahligen Division und dem Modulo-Operator:

```
>>> x = 24
>>> y = 7
>>> x == (x // y) * y + (x % y)
True
```

4.4 Statische und dynamische Typdeklaration

Wer Erfahrungen mit C, C++, Java oder ähnlichen Sprachen hat, hat dort gelernt, dass man einer Variablen einen Typ zuordnen muss, bevor man sie verwenden darf. Der Datentyp muss im Programm festgelegt werden – und zwar, bevor die Variable zum ersten Mal benutzt oder definiert wird. Dies sieht dann beispielsweise in einem C-Programm so aus:

```
int i, j;
float x;
x = i / 3.0 + 5.8;
```

Während des Programmlaufs können sich dann die Werte für i, j und x ändern, aber ihre Typen sind für die Dauer des Programmlaufs auf `int` im Falle von i und j und `float`für die Variable x festgelegt. Dies bezeichnet man als „statische Typdeklaration", da bereits der Compiler den Typ festlegt und während des Programmablaufs keine Änderungen des Typs mehr vorgenommen werden können.

In Python sieht dies anders aus, wie wir bereits die ganze Zeit gesehen haben. Zunächst einmal bezeichnen Variablen in Python keinen bestimmten Typ, und deshalb benötigt man in Python keine Typdeklaration. Benötigt man im Programm beispielsweise eine Variable i mit dem Wert 42 und dem Typ Integer, so erreicht man dies einfach mit der Anweisung „`i = 42`".

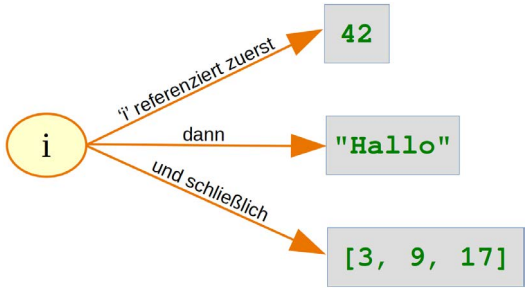


Bild 4.2 ‚i' während der Laufzeit

Damit hat man automatisch ein Objekt vom Typ Integer angelegt, welches dann von der Variablen i referenziert wird. Man kann dann im weiteren Verlauf des Programms der Variablen i auch andere Objekte mit anderen Datentypen zuweisen:

```
>>> i = 42
>>> i = "Hallo"
>>> i = [3, 9, 17]
```

Dennoch ordnet Python in jedem Fall einen speziellen Typ oder genauer gesagt eine spezielle Klasse der Variablen zu. Dieser Datentyp wird aber automatisch von Python erkannt. Diese automatische Typzuordnung, die auch während der Laufzeit erfolgen kann, bezeichnet man als „dynamische Typdeklaration". Mit der Funktion `type` können wir uns den jeweiligen Typ ausgeben lassen:

```
>>> i = 42
>>> type(i)
<class 'int'>
>>> i = "Hallo"
>>> type(i)
<class 'str'>
```

```
>>> i = [3, 9, 17]
>>> type(i)
<class 'list'>
```

Während sich bei statischen Typdeklarationen, also bei Sprachen wie C und C++, nur der Wert, aber nicht der Typ einer Variablen während eines Laufs ändert, kann sich bei dynamischer Typdeklaration, also in Python, sowohl der Wert als auch der Typ einer Variablen ändern.

Aber Python achtet auf Typverletzungen zur Laufzeit eines Programms. Man spricht von einer Typverletzung[9], wenn Datentypen beispielsweise aufgrund fehlender Zuweisungskompatibilität nicht regelgemäß verwendet werden. In Python kann es nur bei Ausdrücken zu Problemen kommen, da man ja einer Variablen einen beliebigen Typ zuordnen kann. Eine Typverletzung liegt beispielsweise vor, wenn man eine Variable vom Typ int zu einer Variablen vom Typ str addieren will. Es wird in diesem Fall ein TypeError generiert:

```
>>> x = "Ich bin ein String"
>>> y = 42
>>> z = x + y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Allerdings kann man Integer- und Float-Werte – auch wenn es sich um unterschiedliche Datentypen handelt – in einem Ausdruck mischen. Der Wert des Ausdrucks wird dann ein Float.

```
>>> x = 12
>>> y = 3.5
>>> z = x * y
>>> z
42.0
>>> type(x)
<class 'int'>
>>> type(y)
<class 'float'>
>>> type(z)
<class 'float'>
```

4.5 Typumwandlung

In der Programmierung bezeichnet man die Umwandlung eines Datentyps in einen anderen als Typumwandlung.[10] Man benötigt Typumwandlungen beispielsweise, wenn man Strings und numerische Werte mit dem Stringoperator „+” konkatenieren möchte.

[9] engl. type conflict
[10] engl. type conversion oder type cast

```
>>> first_name = "Henry"
>>> last_name = "Miller"
>>> age = 20
>>> print(first_name + " " + last_name + ": " + str(age))
Henry Miller: 20
```

In dem Beispiel haben wir den Wert von `age` explizit mit der Funktion `str` in einen String gewandelt. Man bezeichnet dies als explizite Typumwandlung. Hätten wir in obigem Beispiel nicht den Integer-Wert `age` in einen String gewandelt, hätte Python einen TypeError generiert:

```
>>> print(first_name + " " + last_name + ": " + age)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Der Text der Fehlermeldung besagt, dass Python keine implizite Typumwandlung von `int` nach `str` vornehmen kann. Prinzipiell unterstützt Python keine impliziten Typumwandlungen, wie sie in Perl oder PHP möglich sind. Dennoch gibt es Ausnahmen so wie unser Beispiel, in dem wir Integer- und Float-Werte in einem Ausdruck gemischt hatten. Dort wurde der Integer-Wert implizit in einen Float-Wert gewandelt.

4.6 Datentyp ermitteln

In vielen Anwendungen muss man für ein Objekt bestimmen, um welchen Typ bzw. um welche Klasse es sich handelt. Dafür bietet sich meistens die eingebaute Funktion `type` an. Man übergibt ihr als Argument einen Variablennamen und erhält den Typ des Objekts zurück, auf das die Variable zeigt:

```
>>> l = [3, 5, 4]
>>> type(l)
<class 'list'>
>>> x = 4
>>> type(x)
<class 'int'>
>>> x = 4.5
>>> type(x)
<class 'float'>
>>> x = "Ein String"
>>> type(x)
<class 'str'>
```

Als Alternative zur Funktion `type` gibt es noch die eingebaute Funktion `isinstance`, die einen Wahrheitswert „True" oder „False" zurückgibt.

```
isinstance(object, ct)
```

`object` ist das Objekt, das geprüft werden soll, und `ct` entspricht der Klasse oder dem Typ, auf den geprüft werden soll. Im folgenden Beispiel prüfen wir, ob es sich bei dem Objekt s um ein String handelt:

```
>>> s = "Ich bin ein String"
>>> isinstance(s, str)
True
```

In Programmen kommt es häufig vor, dass wir für ein Objekt wissen wollen, ob es sich um einen von vielen Typen handelt. Also zum Beispiel die Frage, ob eine Variable ein Integer oder ein Float ist. Dies kann man mit der Verknüpfung `or` lösen:

```
>>> x = 4
>>> isinstance(x, int) or isinstance(x, float)
True
>>> x = 4.8
>>> isinstance(x, int) or isinstance(x, float)
True
```

Allerdings bietet `isinstance` hierfür eine bequemere Möglichkeit. Statt eines Typs gibt man als zweites Argument ein Tupel von Typen an:

```
>>> x = 4.8
>>> isinstance(x, (int, float))
True
>>> x = (89, 123, 898)
>>> isinstance(x, (list, tuple))
True
>>> isinstance(x, (int, float))
False
```

Möchte man in einem Programm auf einen bestimmten Typ prüfen, so kann man obiges Vorgehen wählen, aber häufig lässt sich dies eleganter mit Ausnahmehandlungen lösen, die wir in Kapitel Ausnahmebehandlung, Seite 215 behandeln.