Diese Leseprobe haben Sie beim edv-buchversand.de heruntergeladen. Das Buch können Sie online in unserem Shop bestellen.

Hier zum Shop

Kapitel 5

# Zeichenketten



Auch wenn die mathematischen Features von Python beeindruckend sind – bei vielen Anwendungen haben Sie öfter mit Zeichenketten zu tun als mit Zahlen. Der Umgang mit Zeichenketten (Datentyp `str`) zählt zu den Grundfertigkeiten aller in der Programmierung.

Python macht Ihnen diesbezüglich das Leben relativ leicht: In wenigen Programmiersprachen gelingt der Zugriff auf (Teil-)Zeichenketten so unkompliziert wie in Python, in wenigen Sprachen gibt es eine derart reiche Auswahl an Verarbeitungs- und Analysefunktionen.

Eine vollständige Referenz aller Funktionen ist hier aus Platzgründen unmöglich. Ich werde aber versuchen, Ihnen auf den folgenden Seiten die wichtigsten Features zu erläutern.

## 5.1 Grundregeln

Zeichenketten werden wahlweise in einfache oder doppelte Apostrophe gestellt, also `'abc'` oder `"abc"`. Beide Varianten sind gleichwertig.

```
s='abc'
type(s)
  <class 'str'>
```

Sollen Zeichenketten im Code über mehrere Zeilen reichen, müssen sie in dreifache Apostrophe gestellt werden, also wahlweise `'''` oder `"""`. In eingerücktem Code enthalten derartige Zeichenketten leider eine Menge überflüssiger Leerzeichen:

```
s = """Das ist
       eine lange
       Zeichenkette."""
```

```
print(s)

Das ist
       eine lange
       Zeichenkette.
```

Natürlich können Sie die Leerzeichen vermeiden, in dem Sie die Zeichenkette einfach nicht einrücken. Innerhalb von Abfragen, Schleifen, Funktionen etc. sieht der Code dann allerdings grauenhaft aus und zerstört die optische Struktur:

```
if True:
    s = """Das ist
eine lange
Zeichenkette."""
    print(s)
```

### Zeichenketten aneinanderfügen und vervielfältigen

Mit dem Operator + fügen Sie Zeichenketten aneinander. * vervielfacht Zeichenketten.

```
s1 = 'abc'
s2 = 'efg'
s3 = s1 + s2 + s1   # Ergebnis 'abcefgabc'
s4 = s1*3 + 'x'*2   # Ergebnis 'abcabcabcxx'
```

### Sonderzeichen

Zeichenketten werden intern in Unicode dargestellt. Bei Codedateien nimmt der Python-Interpreter standardmäßig an, dass sie in UTF-8-Codierung vorliegen. Ist das nicht der Fall, können Sie die tatsächliche Codierung in der zweiten Zeile der Codedatei in der folgenden Form angeben:

```
# -*- coding: <encoding name> -*-
```





Also beispielsweise:

```
# -*- coding: latin-1 -*-
```

In Zeichenketten können mit \ markierte Sonderzeichen eingebettet werden (sogenannte *Escape-Sequenzen*), z. B. `\n` für einen Zeilenumbruch (siehe Tabelle 5.1). Eine vollständige Referenz aller String-Literale finden Sie hier:

*https://docs.python.org/3/reference/lexical_analysis.html#literals*



| Zeichensequenz | Bedeutung |
|---|---|
| \a | Bell (Signalton) |
| \f | Formfeed (neue Seite) |
| \n | Zeilenumbruch |
| \r | Wagenrücklauf (für Windows-Textdateien) |
| \t | Tabulatorzeichen |
| \unnnn | Unicode-Zeichen mit dem Hexcode &xnnnn |
| \' | das Zeichen ' |
| \" | das Zeichen " |
| \\ | das Zeichen \ |

**Tabelle 5.1** Ausgewählte Escape-Sequenzen

```
s = "Erste\nzweite\ndritte Zeile."
print(s)

Erste
zweite
dritte Zeile.
```

### Raw-Zeichenketten

Python interpretiert \-Sequenzen als Sonderzeichen (siehe Tabelle 5.1). Wenn Sie das nicht möchten und jedes \-Zeichen als solches gewertet werden soll, stellen Sie der gesamten Zeichenkette den Buchstaben `r` (*raw*) voran:

```
latexcode = r'\section{Überschrift}'
```

### »chr«- und »ord«-Funktion

Die Funktion `chr` liefert ein Zeichen, das dem übergebenen ASCII-Code bzw. Unicode entspricht:

```
chr(65)      # 'A'
chr(8364)    # '€'
```

Die Umkehrfunktion `ord` liefert den ASCII-Code bzw. Unicode zu einem einzelnen Zeichen:

```
ord('A')     # 65
ord('€')     # 8364
ord('abc')   # TypeError, expected a character
```

## 5.2 Zugriff auf Teilzeichenketten

Für den Zugriff auf Teile einer Zeichenkette gilt die sogenannte *Slicing*-Syntax. `s[n]` gibt das `n`-te Zeichen zurück, wobei `n=0` das erste Zeichen repräsentiert. `s[start:ende]` liefert die Zeichen von `start` (inklusive) bis `ende` (exklusive). Mit negativen Werten geben Sie den Offset vom Ende der Zeichenkette an. Verglichen mit vielen anderen Programmiersprachen ist diese Syntax genial einfach und praktisch. Probieren Sie die folgenden Beispiele im Python-Interpreter aus!

```
s='abcdefghijklmnopqrstuvwxyz'
s[3]  # das vierte Zeichen, weil die Zählung bei 0 beginnt
  'd'
```





```
s[3:6]
  'def'
s[:3] # alles bis einschließlich des dritten Zeichens
  'abc'
s[3:] # alles ab dem dritten Zeichen (exklusive)
  'defghijklmnopqrstuvwxyz'
s[-4] # das viertletzte Zeichen
  'w'
s[-4:] # alles ab dem viertletzten Zeichen
  'wxyz'
```

### »IndexError«

Wenn Sie in der Slicing-Schreibweise Indizes angeben, die die Länge der Zeichenkette überschreiten, tritt ein Fehler auf:

```
s[25]    # ok, letztes Zeichen
  'z'
s[-1]    # auch ok, letztes Zeichen
  'z'
s[26]    # Fehler
  IndexError: string index out of range
s[-26]   # ok, erstes Zeichen
  'a'
s[-27]   # Fehler
  IndexError: string index out of range
```

### Schrittweite (Stride)

Durch einen dritten Parameter (dem *Stride-Parameter*) kann eine Art Schrittweite angegeben werden (also `s[start:ende:schrittweite]`). Da alle Parameter optional sind, ergeben sich viele interessante Varianten:

```
s[::2]      # jedes zweite Zeichen
  'acegikmoqsuwy'
s[:10:2]    # von den ersten 10 Zeichen jedes zweite
  'acegi'
```

```
s[10::2]    # ab dem 10. Zeichen jedes zweite
  'kmoqsuwy'
```

Mit einer negativen Schrittweite kehren Sie die Reihenfolge einer Zeichenkette um:

```
s[::-1]     # alles in umgekehrter Reihenfolge
  'zyxwvutsrqponmlkjihgfedcba'
s[::-2]     # jedes 2. Zeichen in umgekehrter Reihenfolge
  'zxvtrpnljhfdb'
```

Unerwartete Ergebnisse erhalten Sie, wenn Sie einen negativen Stride-Parameter mit Start- und Endposition kombinieren:

```
s[0:10]
  'abcdefghij'
s[0:10:-1]  # leeres Ergebnis
  ''
s[10:0:-1]  # vom 11. zum 2. Zeichen
  'kjihgfedcb'
```

Die Ergebnisse haben damit zu tun, wie Python intern die Schleife ausführt, um auf die Elemente zuzugreifen. Wenn Sie einen Teil einer Zeichenkette in umgekehrter Reihenfolge brauchen, ist es zumeist einfacher, zwei []-Konstruktionen hintereinanderzustellen. Die erste wählt die Zeichen aus, die zweite dreht die Reihenfolge um:

```
s[:10][::-1]  # die ersten 10 Zeichen in inverser
              # Reihenfolge
  'jihgfedcba'
```

## 5.3 Zeichenkettenfunktionen

Zeichenketten können mit diversen Funktionen und Methoden bearbeitet werden (siehe Tabelle 5.2). Eine vollständige und detaillierte Beschreibung aller String-Methoden finden Sie hier:

*https://docs.python.org/3/library/stdtypes.html#string-methods*





| Methode | Funktion |
|---|---|
| `len(s)` | Ermittelt die Anzahl der Zeichen. |
| `str(x)` | Wandelt `x` in eine Zeichenkette um. |
| `sub in s` | Testet, ob `sub` in `s` vorkommt. |
| `s.count(sub)` | Ermittelt, wie oft `sub` in `s` vorkommt. |
| `s.endswith(sub)` | Testet, ob `s` mit `sub` endet. |
| `s.expandtabs()` | Ersetzt Tabulatorzeichen durch Leerzeichen. |
| `s.find(sub)` | Sucht `sub` in `s` und liefert die Startposition oder `-1` zurück. |
| `str.isxxx(s)` | Testet Eigenschaften von `s`: `islower()`, `isdigit()` etc. |
| `s.join(x)` | Verbindet die Zeichenketten in `x` (Liste, Set, Tupel). |
| `s.lower()` | Liefert `s` mit lauter Kleinbuchstaben zurück. |
| `s.partition(sub)` | Trennt `s` auf und liefert drei Teile als Tupel zurück. |
| `s.replace(old, new)` | Liefert `s` zurück, wobei `old` jeweils durch `new` ersetzt wird. |
| `s.rfind(sub)` | Wie `find`, aber beginnt die Suche am Ende der Zeichenkette. |
| `s.split(sub)` | Zerlegt `s` bei jedem Vorkommen von `sub`, liefert eine Liste. |
| `s.splitlines()` | Zerlegt `s` zeilenweise, liefert eine Liste. |
| `s.startswith(sub)` | Testet, ob `s` mit `sub` beginnt. |
| `s.strip()` | Entfernt Whitespace vom Beginn und Ende. |
| `s.upper()` | Liefert `s` mit lauter Großbuchstaben zurück. |

**Tabelle 5.2** Ausgewählte Methoden und Funktionen für Zeichenketten

**Funktionen versus Methoden**

Methoden sind so etwas Ähnliches wie Funktionen, sie werden aber direkt auf die zugrundeliegenden Daten (Objekte) angewendet. Das äußert sich in der Schreibweise:

`funktion(daten)`
`daten.methode()` bzw. `daten.methode(weitere, daten)`

Die Anwendung von Methoden ist also nicht schwierig. Da Python eine objektorientierte Programmiersprache ist, gibt es viel mehr Methoden als Funktionen. Hintergründe zu Objekten und Methoden sowie Wege, eigene Klassen samt Methoden selbst zu programmieren, werden Sie in Kapitel 11, »Objektorientierte Programmierung«, kennenlernen.

Die folgenden Zeilen zeigen einige Anwendungsbeispiele für diese Funktionen und Methoden:

```
s='abcdefghijklmnopqrstuvwxyz'

s.upper()        # in Großbuchstaben umwandeln
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

'efg' in s       # ist 'efg' enthalten?
  True

s.count('efg')   # wie oft ist 'efg' enthalten?
  1

s.partition('e')  # beim Zeichen 'e' auftrennen (Tupel)
  ('abcd', 'e', 'fghijklmnopqrstuvwxyz')

s.split('e')      # beim Zeichen 'e' auftrennen (Liste)
  ['abcd', 'fghijklmnopqrstuvwxyz']
```





### Eigenschaften von Zeichen(ketten) ermitteln

`len` ermittelt die Anzahl der Zeichen einer Zeichenkette. Mit diversen `str.isxxx`-Funktionen können Sie testen, ob ein Zeichen bzw. die gesamte Zeichenkette aus lauter Zeichen eines bestimmten Typs besteht:

- `str.isalpha(s)`: Besteht `s` aus Buchstaben (inklusive `'äöüß'`)? Ein Leerzeichen gilt nicht als Buchstabe!
- `str.isdigit(s)`: Enthält `s` ausschließlich Ziffern (0 bis 9)?
- `str.isalnum(s)`: Enthält `s` ausschließlich Ziffern oder Buchstaben?
- `str.isascii(s)`: Besteht `s` aus ASCII-Zeichen (Code zwischen 0 und 127, verfügbar erst ab Python 3.7)?
- `str.islower(s)`: Enthält `s` keine Großbuchstaben? (Die Zeichenkette darf andere Zeichen enthalten, aber alle Buchstaben müssen klein sein.)
- `str.isupper(s)`: Enthält `s` keine Kleinbuchstaben?

Alle obigen Funktionen liefern `False`, wenn die Zeichenkette leer ist.

```
len('abc')                # 3
str.isalpha('a')          # True
str.isalpha('abcäöü')     # True
str.isalpha('abc123')     # False
str.isalpha(' ')          # False
str.isdigit('123')        # True
str.isalnum('abc123')     # True
str.isalnum('!')          # False
str.isascii('abc123|$!') # True
str.islower('abc')        # True
str.islower('abc123_')    # True
str.islower('abcD')       # False
```

**Spezialfunktionen**

Python kennt auch die `str`-Funktionen `isnumeric` und `isdecimal`. Sie unterscheiden sich von `isdigit` insofern, als sie auch Spezialzeichen wie ³ oder ½ erkennen. Details sind hier dokumentiert:

*https://stackoverflow.com/questions/44891070*

### Suchen und ersetzen

find beginnt die Suche normalerweise am Beginn der Zeichenkette. Mit einem optionalen Parameter kann der Startpunkt der Suche angegeben werden, z. B. um in einer Schleife mehrere Vorkommen des Suchbegriffs zu finden.

```
s="abcdefghijklmnopqrstuvwxyz"
s.find('efg')     # 'efg' wird an der Position 4 gefunden
  4
s.find('efg', 4)  # auch dann, wenn die Suche an dieser
  4               # Position beginnt
s.find('efg', 5)  # aber nicht mehr, wenn die Suche eine
  -1              # Stelle weiter startet
```

Anstelle von find können Sie auch die verwandte Methode index verwenden. Sollte der Suchbegriff nicht zu finden sein, löst index einen Fehler aus, anstatt den Wert -1 zurückzugeben.

replace ersetzt alle Vorkommen einer Suchzeichenkette (im folgenden Beispiel 'e') durch eine andere Zeichenkette (hier 'X'):

```
s.replace('e', 'X')
  'abcdXfghijklmnopqrstuvwxyz'
```

## 5.4 Zeichenketten formatieren und konvertieren

Häufig müssen Sie aus Zahlen, Datums- und Zeitangaben etc. Zeichenketten bilden. Im einfachsten Fall verwenden Sie dazu die Funktionen str(x) oder repr(x), die jedes beliebige Objekt als Zeichenkette darstellen. Die Funktion repr geht dabei so vor, dass die resultierende Zeichenkette mit eval wieder eingelesen werden kann. str bemüht sich hingegen, die Zeichenketten so zu formatieren, dass sie für Menschen gut leserlich sind.

```
from fractions import Fraction
x=Fraction('1/3')  # der Bruch 1/3 als Fraction-Objekt
print(str(x))      # gut lesbare Darstellung
  1/3
```





```
s=repr(x)          # maschinenlesbare Form
print(s)
  'Fraction(1, 3)'
y=eval(s)          # Rückumwandlung in ein Objekt
print(y)
  1/3
```



### Daten formatieren

Weder str noch repr gibt Ihnen Einfluss auf die Formatierung. Wenn Sie Zahlen rechtsbündig ausgeben oder mit Tausendertrennung darstellen möchten, dann benötigen Sie spezielle Formatierungsfunktionen. Unter Python haben Sie die Wahl zwischen mehreren Verfahren:

- formatzeichenkette % (daten, daten, daten): Hier wird die formatzeichenkette in der Syntax der printf-Funktion der Programmiersprache C formuliert. Innerhalb dieser Zeichenkette geben %-Zeichen die Position der einzusetzenden Daten an.
- formatzeichenkette.format(daten, daten, daten): Bei dieser Variante hat der Aufbau der formatzeichenkette große Ähnlichkeiten mit dem Aufbau der gleichnamigen Methode des .NET-Frameworks von Microsoft. Innerhalb dieser Zeichenkette geben {}-Klammernpaare die Position der Parameter an.
- Bei der Kurzschreibweise f'{varname}' mit vorangestelltem f wird {varname} durch den Inhalt der Variablen ersetzt.
- locale.format_string(...): Diese Methode aus dem locale-Modul berücksichtigt bei der Formatierung landesspezifische Einstellungen und verwendet z. B. ein Komma anstelle des Dezimalpunkts.

### Formatierung mit dem %-Operator

Zuerst drei Beispiele für das %-Verfahren, die Sie im Python-Interpreter ausprobieren können:

```
print('%s ist %d Jahre alt.' % ('Matthias', 11))
  'Matthias ist 11 Jahre alt.'
```

```
print('1/7 mit drei Nachkommastellen: %.3f' % (1/7))
  '1/7 mit drei Nachkommastellen: 0.143'
print('<img src="%s" alt="%s" width="%d">' %
      ('foto.jpg', 'Porträt', 200))
  '<img src="foto.jpg" alt="Porträt" width="200">'
```

Es gibt unzählige Codes zum Aufbau der Zeichenketten für die beiden Formatierungssysteme. Die wichtigsten sind in Tabelle 5.3 zusammengefasst, einige weitere auf der folgenden Webseite dokumentiert:

*https://docs.python.org/3/library/stdtypes.html#old-string-formatting-operations*

| Code | Bedeutung |
|---|---|
| `%d` | ganze Zahl (dezimal) |
| `%5d` | ganze Zahl mit fünf Stellen, rechtsbündig |
| `%-5d` | ganze Zahl mit fünf Stellen, linksbündig |
| `%f` | Fließkommazahl (*float*) |
| `%.2f` | Fließkommazahl mit zwei Nachkommastellen |
| `%r` | Zeichenkette, Python verwendet `repr`. |
| `%s` | Zeichenkette, Python verwendet `str`. |
| `%10s` | Zeichenkette mit zehn Zeichen, rechtsbündig |
| `%-10s` | Zeichenkette mit zehn Zeichen, linksbündig |
| `%x` | ganze Zahl hexadezimal ausgeben |

**Tabelle 5.3** Ausgewählte Codes für die %-Formatierung (»printf«-Syntax)

### Formatierung mit der »format«-Methode

Der größte Vorteil der neueren `format`-Methode besteht darin, dass die Platzhalterreihenfolge durch `{n}` frei gewählt werden kann und daher unabhängig von der Reihenfolge der Parameter ist. Für eine vollständige





Referenz der vielen Formatierungscodes fehlt abermals der Platz. Die wichtigsten Codes sind in Tabelle 5.4 aufgelistet. Weitere Codes finden Sie hier:

*https://docs.python.org/3/library/string.html#format-string-syntax*

| Code | Bedeutung |
|---|---|
| `{}` | Parameter, beliebiger Datentyp |
| `{0}, {1}, ...` | nummerierte Parameter |
| `{eins}, {zwei}, ...` | benannte Parameter |
| `{:d}` | ganze Zahl |
| `{:<7d}` | ganze Zahl mit sieben Stellen, linksbündig |
| `{:>7d}` | ganze Zahl mit sieben Stellen, rechtsbündig |
| `{:^7d}` | ganze Zahl mit sieben Stellen, zentriert |
| `{:f}` | Fließkommazahl |
| `{:,f}` | Fließkommazahl mit Tausendertrennung |
| `{:.5f}` | Fließkommazahl mit fünf Nachkommastellen |
| `{:n}` | Zahl mit Lokalisierung |
| `{:s}` | Zeichenkette |

**Tabelle 5.4** Ausgewählte Codes für die »format«-Methode

Beachten Sie in den folgenden Beispielen die eigenwillige Syntax in der Form `fmtstr.format(daten, daten, daten)`!

```
print('{} ist {} Jahre alt.'.format('Sebastian', 13))
  'Sebastian ist 13 Jahre alt.'

# umgekehrte Reihenfolge der Parameter!
print('{1} ist {0} Jahre alt.'.format(13, 'Sebastian'))
  'Sebastian ist 13 Jahre alt.'
```

```
# benannte Parameter
print('{name} ist {alter} Jahre alt.'.format(
       alter=13, name='Sebastian'))
  'Sebastian ist 13 Jahre alt.'

print('1/7 mit drei Nachkommastellen: {:.3f}'.format(1/7))
  '1/7 mit drei Nachkommastellen: 0.143'

print('SELECT * FROM table WHERE id={:d}'.format(324))
  'SELECT * FROM table WHERE id=324'
```

### »format«-Kurzschreibweise

Wenn Sie in der Formatierungszeichenkette Variablennamen verwenden und der Zeichenkette den Buchstaben `f` voranstellen, dann brauchen Sie weder `format` explizit aufzurufen noch irgendwelche Parameter zu übergeben:

```
alter=13
name='Sebastian'
print(f'{name} ist {alter} Jahre alt.')
  Sebastian ist 13 Jahre alt.
```

Die Schreibweise lässt sich auch mit den vorhin zusammengefassten Formatierungsparametern kombinieren:

```
x=1/7
print(f'{x:.2}')  # x mit zwei Nachkommastellen ausgeben
  0.14
```

Neu seit Python 3.8 ist die Schreibweise `{name=}`. Dabei wird zuerst der Text `name=` und dann der Inhalt der gleichnamigen Variablen ausgegeben.

```
print(f'{name=} {alter=}')
  name='Sebastian' alter=13
```





**Die Qual der Wahl**

Was ist nun besser, das `%`-Verfahren oder die `format`-Methode, egal, ob in der herkömmlichen Form oder in der Kurzschreibweise?

Sie kommen mit beiden Verfahren zum Ziel. Wenn Ihnen die `printf`-Syntax vertraut ist, spricht nichts dagegen, beim `%`-Verfahren zu bleiben. Bei aktuellen Python-Versionen führt die neue Kurzschreibweise `f'...'` zu dem am besten lesbaren Code.



## 5.5 Lokalisierung

Die vorhin präsentierten Formatierungsverfahren mit `%` und mit `format` berücksichtigen die Spracheinstellungen des Systems leider nicht. Fließkommazahlen werden immer mit dem US-typischen Dezimalpunkt ausgegeben:

```
x=2.5
print("%f" % x)
  2.500000
print(f'{x:f}')
  2.500000
```

Zur Veränderung des Defaultverhaltens verwenden Sie die Methoden des `locale`-Moduls. Die darin enthaltenen Methoden helfen bei der Lokalisierung eines Programms, also bei der Anpassung an landestypische Gepflogenheiten.

Laut Dokumentation sollte es ausreichen, `setlocale` auszuführen und als ersten Parameter `LC_ALL` zu übergeben. Damit werden alle Lokalisierungseinstellungen geändert, wobei die Einstellungen des Betriebssystems übernommen werden. Leider funktioniert dies vielfach nicht. Abhilfe schafft dann eine explizite Einstellung, die aber je nach Plattform unterschiedlich erfolgen muss (siehe das folgende Listing). Unter macOS und Linux können Sie die zur Auswahl stehenden Lokalisierungszeichenketten in einem Terminal mit dem Kommando `locale -a` feststellen.

Zur formatierten Ausgabe verwenden Sie dann `format_string(fmt, val)`, wobei die gleichen Syntaxregeln gelten wie für `fmt % val`:

```
import locale

# Einstellungen des Betriebssystems übernehmen
# (funktioniert oft nicht)
locale.setlocale(locale.LC_ALL)

# Lokalisierung explizit einstellen (Linux)
locale.setlocale(locale.LC_ALL, 'de_DE.utf-8')

# Lokalisierung explizit einstellen (macOS)
locale.setlocale(locale.LC_ALL, 'de_DE.UTF-8')

# Lokalisierung explizit einstellen (Windows)
locale.setlocale(locale.LC_ALL, 'german')

x=2.5
print(locale.format_string('%f', x))
  2,500000
```

Auch die »gewöhnliche« `format`-Methode liefert lokalisierte Zahlen, wenn Sie das Formatzeichen `n` für lokalisierte Zahlen verwenden:

```
print('{:n}'.format(x))
  2,5
```

### Lokalisierte Zeichenketten in Zahlen umwandeln

Wenn Sie bei der Konvertierung von Zeichenketten in Zahlen die Lokalisierung berücksichtigen möchten, verwenden Sie die Methode `atof`:

```
s='2,5'
x=locale.atof(s)
print(x*2)
  5
```





> **Lokalisierung von Zeitangaben**
>
> Die Methoden des `locale`-Moduls sind auch erforderlich, um Zeitangaben (z. B. Wochentage, Monatsnamen) landestypisch auszugeben. Entsprechende Beispiele folgen in Kapitel 6, »Datum und Zeit«.



### Lokalisierungsbeispiel

Das folgende Script erwartet die Eingabe von Länge und Breite eines Rechtecks, wobei das hierzulande übliche Komma verwendet werden darf. Das Programm berechnet den Flächeninhalt und gibt den Wert mit zwei Nachkommastellen ebenfalls in der bei uns üblichen Notation aus. Damit das Programm unter allen Betriebssystemen zufriedenstellend funktioniert, wird mit `platform.system` das Betriebssystem ermittelt und `setlocale` entsprechend ausgeführt. (`if` lernen Sie eigentlich erst in Kapitel 8, »Verzweigungen und Schleifen«, kennen, der Code sollte aber ohne weitere Erläuterungen verständlich sein.)

Beachten Sie, dass `input` in jedem Fall eine Zeichenkette zurückgibt. Für `input` ist daher keine Lokalisierung erforderlich. Wichtig ist aber, dass zur Konvertierung der Zeichenkette in eine Zahl die Methode `atof` verwendet wird.

```
# Beispielprogramm rechteck.py
import locale, platform
if platform.system() == 'Windows':
    locale.setlocale(locale.LC_ALL, 'german')
if platform.system() == 'Linux':
    locale.setlocale(locale.LC_ALL, 'de_DE.utf8')
else:
    locale.setlocale(locale.LC_ALL, 'de_DE.UTF8')

s = input("Länge des Rechtecks: ")        # str
laenge = locale.atof(s)                   # float
s = input("Breite des Rechtecks: ")       # str
breite = locale.atof(s)                   # float
```

```
flaeche = laenge * breite                 # float
s = locale.format_string('%.2f', flaeche) # str
print("Flächeninhalt: ", s)
```

Ein Probelauf des Programms sieht z. B. so aus:

```
./rechteck.py
  Länge des Rechtecks: 12,5
  Breite des Rechtecks: 7,8
  Flächeninhalt:  97,50
```

## 5.6 Reguläre Ausdrücke

*Reguläre Ausdrücke* bzw. *regular expressions* sind eine eigene Art von Sprache, die Suchmuster für Zeichenketten beschreibt. Sie können damit z. B. alle Links aus einem HTML-Dokument extrahieren oder komplexe Suchen-und-Ersetzen-Vorgänge durchführen. In Python sind die entsprechenden Funktionen im Modul `re` gebündelt (siehe Tabelle 5.5).

| Funktion | Bedeutung |
| --- | --- |
| `match(pattern, s)` | Testet, ob `s` dem Muster `pattern` entspricht. |
| `search(pattern, s)` | Liefert die Position, an der das Muster vorkommt. |
| `split(pattern, s)` | Zerlegt `s` bei jedem Vorkommen des Suchmusters |
| `sub(pattern, r, s)` | Ersetzt das erste gefundene Muster in `s` durch `r`. |

**Tabelle 5.5** Ausgewählte Funktionen des »re«-Moduls

Die folgenden Zeilen zeigen eine einfache Anwendung: Das Programm erwartet mit `input` die Eingabe einer E-Mail-Adresse. `pattern` enthält einen regulären Ausdruck zum Testen, ob die Adresse formalen Kriterien entspricht. Dazu muss der erste Teil der Adresse aus den Buchstaben a–z, den Ziffern 0–9 sowie einigen Sonderzeichen bestehen. Danach folgt ein @-Zeichen, dann ein weiterer Block aus Buchstaben, Ziffern und Zeichen, schließlich ein Punkt und zuletzt ein reiner Buchstabenblock für die Top-





Level-Domain (z. B. `info`). Damit die Zeichenkette korrekt interpretiert wird, muss sie als Raw-Zeichenkette angegeben werden (also `r'...'`).

```
import re
pattern =
  r'^[A-Za-z0-9\.\+_-]+@[A-Za-z0-9\._-]+\.[a-zA-Z]+$'
email = input("Geben Sie eine E-Mail-Adresse ein: ")

if re.match(pattern, email):
  print("Die E-Mail-Adresse ist OK.")
else:
  print("Die E-Mail-Adresse sieht fehlerhaft aus.")
```

Das Beispiel macht auch gleich klar, dass das Zusammenstellen regulärer Ausdrücke alles andere als einfach ist. Für gängige Problemstellungen finden Sie oft im Internet passende Lösungen. Der obige Code folgt z. B. einem Vorschlag von der folgenden Webseite:

*http://stackoverflow.com/questions/8022530*

Andernfalls müssen Sie den regulären Ausdruck selbst zusammenbasteln (siehe Tabelle 5.6). Das Muster `[a-z]+` trifft beispielsweise auf beliebige aus Kleinbuchstaben zusammengesetzte Zeichenketten zu, z. B. auf `abc` oder `x` oder `xxx`, nicht aber auf `a b` (Leerzeichen), `Abc` (Großbuchstabe), `a1` (Ziffer) oder `äöü` (internationale Buchstaben).

| Code | Bedeutung |
| --- | --- |
| `^` | Beginn der Zeichenkette |
| `$` | Ende der Zeichenkette |
| `.` | ein beliebiges Zeichen |
| `[a-z]` | ein Kleinbuchstabe zwischen `a` und `z` |
| `[a,b,f-h]` | ein Buchstabe aus `a`, `b`, `f`, `g` und `h` |
| `[^0-9]` | ein beliebiges Zeichen außer `0` bis `9` |

**Tabelle 5.6** Aufbau regulärer Ausdrücke

| Code | Bedeutung |
|---|---|
| `<muster>*` | Das Muster darf beliebig oft vorkommen (auch 0-mal). |
| `<muster>+` | Das Muster darf beliebig oft vorkommen (mindestens einmal). |
| `\x` | Spezialzeichen angeben (`\$` steht also für ein $-Zeichen.) |

**Tabelle 5.6** Aufbau regulärer Ausdrücke (Forts.)

Weitere Details und Beispiele können Sie in der Dokumentation zum `re`-Modul nachlesen:

*https://docs.python.org/3/library/re.html*

## 5.7 Wiederholungsfragen und Übungen

- **W1:** Wie bilden Sie eine Zeichenkette, die selbst ein Anführungszeichen enthält?
- **W2:** Wie bilden Sie Zeichenketten, die das Zeichen \ enthalten?
- **W3:** Extrahieren Sie aus der folgenden Zeichenkette das Tag zwischen den eckigen Klammern:

  ```
  bla [wichtig] mehr bla
  ```

- **W4:** Zerlegen Sie den folgenden Dateinamen in Linux-Notation in die Verzeichnisangabe (bis zum letzten /-Zeichen) und den eigentlichen Dateinamen (ab dieser Position):

  ```
  /home/kofler/Bilder/foto1.jpg
  ```

- **W5:** Fordern Sie den Anwender eines Scripts auf, seinen Namen einzugeben, und entfernen Sie dann alle Leerzeichen am Beginn und Ende der Eingabe.
- **W6:** Geben Sie drei maximal fünfstellige Zahlen rechtsbündig aus.
- **W7:** Geben Sie *Hello, World!* in umgekehrter Reihenfolge aus.

Kapitel 17

# Raspberry Pi

2012 stellte die Raspberry Pi Foundation einen kostengünstigen Minicomputer vor. Das Gerät ist vor allem zum Basteln sowie für den Unterricht konzipiert. Im Gegensatz zu »gewöhnlichen« Computern fehlen Gehäuse, Bildschirm und Tastatur. Dafür gibt es eine Pinleiste mit programmierbaren Ein- und Ausgängen (*General Purpose Input/Output*, kurz GPIO).

Was hat der Raspberry Pi mit Python zu tun? Python hat sich als *die* Programmiersprache etabliert, um den Raspberry Pi zu programmieren. Da auf dem Raspberry Pi eine Linux-Distribution (üblicherweise Raspberry Pi OS) läuft und unzählige andere Programmiersprachen zur Auswahl stellt, ist die Dominanz von Python keineswegs selbstverständlich.

Die Vorteile von Python auf dem Raspberry Pi sind die gleichen wie auf jedem anderen Computer: Die Einstiegshürde ist gering. Python ermöglicht es, viele Aufgaben mit wenigen Zeilen Code und minimalem Overhead zu lösen. Unzählige Anleitungen im Internet sowie eine große Auswahl von Modulen zur Steuerung diverser Hardwarekomponenten waren weitere Faktoren für den Erfolg von Python.

Ich habe sogar den Eindruck, dass die aktuelle Popularität von Python zumindest teilweise auf den Raspberry Pi zurückzuführen ist. In den vergangenen Jahren lernten Millionen Schüler, Studenten und Bastler über den Raspberry Pi die Programmiersprache Python kennen.

In diesem Kapitel stelle ich Ihnen grundlegende Python-Module zur Steuerung des Raspberry Pi vor. Aus Platzgründen kann ich hier aber weder auf Raspberry-Pi-Grundlagen eingehen noch eine Elektronikeinführung geben. All das finden Sie im Buch »Raspberry Pi – Das umfassende Handbuch«, das ich zusammen mit Christoph Scherbeck und Charly Kühnast verfasst und ebenfalls im Rheinwerk Verlag veröffentlicht habe.

## 17.1 GPIO-Zugriff mit RPi.GPIO

Es gibt mehrere Python-Module, die Ihnen beim Lesen bzw. Verändern des Status von GPIO-Pins helfen. Dieser Abschnitt stellt das seit mehreren Jahren bewährte Modul `RPi.GPIO` vor. Damit können Sie einzelne Pins auslesen oder verändern sowie die Software Pulse Width Modulation verwenden. *Nicht* unterstützt werden momentan serielle Schnittstellen, die Bussysteme SPI und $I^2C$, Hardware-PWM sowie das Auslesen von 1-Wire-Temperatursensoren. Die Onlinedokumentation des `RPi.GPIO`-Moduls finden Sie hier:

*https://pypi.python.org/pypi/RPi.GPIO*

Eine Alternative dazu ist das Modul `gpiozero`, das ich Ihnen in Abschnitt 17.3, »GPIO-Zugriff mit gpiozero«, näher vorstelle.

Den folgenden Beispielen liegt eine äußerst einfache Beschaltung des Raspberry Pi zugrunde (siehe Abbildung 17.1): Pin 26 des J8-Headers (GPIO 7 in BCM-Notation) ist über einen Vorwiderstand mit einer Leuchtdiode (*Light Emitting Diode*, kurz LED) verbunden, und Pin 21 (GPIO 9) dient als Signaleingang für einen Taster mit Pull-up-Widerstand.

**Pin-Nummern versus BCM-Nummern**

Der Raspberry Pi verfügt über eine Steckerleiste mit 40 Pins. Diese Leiste wird *J8-Header* genannt. Die Pins sind von 1 bis 40 durchnummeriert. Beim Aufbau einer Schaltung ist also die Pin-Nummer entscheidend.

Bei der CPU des Raspberry Pi handelt es sich um ein *System-on-a-Chip* (SOC) der Firma Broadcom. Dieser Chip stellt weit mehr als 40 GPIOs zur Verfügung. Diese sind intern ganz anders als die Pins nummeriert. Die BCM-Nummer bezieht sich auf die Dokumentation des Chips.

Bei der Programmierung des in diesem Abschnitt behandelten `RPi.GPIO`-Moduls können Sie wahlweise mit Pin- oder mit BCM-Nummern arbeiten. Das im nächsten Abschnitt vorgestellte `gpiozero`-Modul erwartet hingegen *immer* BCM-Nummern. Insofern ist es wichtig, dass Sie mit den Nummern nicht durcheinanderkommen.





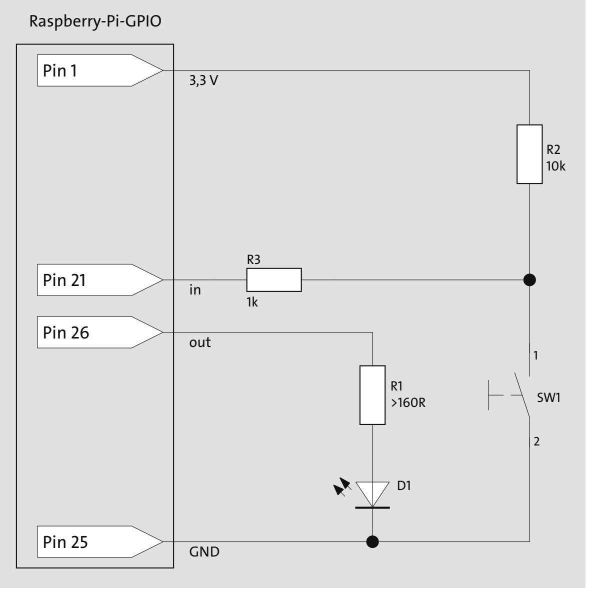


**Abbildung 17.1** Versuchsaufbau zum Test der RPi.GPIO-Funktionen

### GPIO-Setup

Das Modul `RPi.GPIO` ist unter Raspberry Pi OS standardmäßig installiert, d. h., Sie können auf das sonst übliche Kommando `pip3 install <modulname>` verzichten. Wegen des langen Modulnamens ist es empfehlenswert, mit `as gpio` ein Kürzel zu definieren.

Vor dem ersten Zugriff auf GPIOs müssen Sie festlegen, ob Sie mit den internen GPIO-Nummern der BCM-CPU arbeiten möchten oder die Pin-Nummerierung des J8-Headers verwenden möchten. Dazu führen Sie *eine* der beiden folgenden `setmode`-Funktionen aus. Alle Beispiele in diesem Abschnitt verwenden die Pin-Nummern des J8-Headers (also `gpio.BOARD`).

```
import RPi.GPIO as gpio
gpio.setmode(gpio.BOARD)    # Pin-Nummern des J8-Headers
gpio.setmode(gpio.BCM)      # oder Broadcom-GPIO-Nummern
```

Schließlich müssen Sie jeden GPIO-Pin einrichten, den Sie in Ihrem Script nutzen wollen. Dazu geben Sie mit `setup` an, ob der Pin zur Ein- oder zur Ausgabe dient. Zulässige Einstellungen sind `IN` und `OUT`. Für die Schaltung aus Abbildung 17.1 sind die folgenden Einstellungen erforderlich:

```
# Beispielprogramm  gpio-intro.py
import RPi.GPIO as gpio
gpio.setmode(gpio.BOARD)   # Pin-Nummern verwenden
gpio.setup(26, gpio.OUT)   # Pin 26 zur Datenausgabe
gpio.setup(21, gpio.IN)    # Pin 21 zur Dateneingabe
```

Die `setup`-Funktion liefert unter Umständen eine Warnung, wenn ein anderes Programm den GPIO-Pin ebenfalls nutzt: *RuntimeWarning: This channel is already in use, continuing anyway.* Das Programm wird also trotz der Warnung fortgesetzt, zumal die Warnung oft nur ein Indiz dafür ist, dass beim letzten Durchlauf der erforderliche `cleanup`-Aufruf weder von Ihnen selbst noch von einem Python-Script ausgeführt wurde (siehe unten). Gegebenenfalls können Sie die Warnung unterdrücken, indem Sie vor dem `setup`-Aufruf die folgende Zeile einbauen:

```
gpio.setwarnings(False)     # Warnungen unterdrücken
gpio.setup(...)
```

Zum Programmende sollten Sie alle von Ihrem Script genutzten GPIO-Pins wieder freigeben. Dazu führen Sie normalerweise einfach `cleanup` aus. Stellen Sie durch `try/finally` sicher, dass `cleanup` auch dann ausgeführt wird, wenn im Script ein Fehler aufgetreten ist:

```
try:
    gpio-Code
finally:
    gpio.cleanup()
```

Sollten Sie nicht alle GPIO-Pins, sondern nur einen ausgewählten Pin freigeben wollen, übergeben Sie an `cleanup` je nach `setmode`-Einstellung die betreffende Pin- oder GPIO-Nummer:

```
gpio.cleanup(n)
```





### LED ein- und ausschalten

Nach diesen Vorbereitungsarbeiten können Sie nun endlich GPIOs verändern oder auslesen, je nachdem, ob diese zur Aus- oder Eingabe eingestellt wurden. Um den Ausgangszustand zu verändern, verwenden Sie die `output`-Funktion:

```
# Beispielprogramm  gpio-intro.py
gpio.output(26, gpio.HIGH)  # GPIO auf High stellen
                            # (LED leuchtet)
time.sleep(2)
gpio.output(26, gpio.LOW)   # GPIO auf Low stellen
                            # (LED leuchtet nicht mehr)
```

### LED-Helligkeit steuern

Eigentlich sehen die GPIOs des Raspberry Pi nur eine digitale Steuerung vor, also Ein oder Aus. Durch einen Trick können Sie aber auch die Helligkeit einer Leuchtdiode steuern. Dazu wird die LED pro Sekunde Hunderte Male ein- und wieder ausgeschaltet. Das Verhältnis der Zeitanteile, während der die LED leuchtet und während der sie dunkel ist, bestimmt die Helligkeit, die Sie wahrnehmen. Dieses Verfahren wird *Pulsweitenmodulation* (PWM) genannt:

*https://de.wikipedia.org/wiki/Pulsweitenmodulation*

Mit dem `RPi.GPIO`-Modul können Sie über ein `PWM`-Objekt eine softwaregesteuerte PWM für alle Signalausgänge einstellen. Beim Erzeugen des `PWM`-Objekts geben Sie die gewünschte Frequenz an. `start` aktiviert PWM, wobei Sie in einem Parameter zwischen 0 und 100 die Leuchtstärke (*duty*) angeben. Später können Sie die Helligkeit mit `ChangeDutyCycle` ändern oder mit `stop` die Modulation beenden.

Der folgende Beispielcode macht eine Leuchtdiode über den Verlauf von 4 Sekunden zuerst immer heller und dann immer dunkler. Die zweite `for`-Schleife durchläuft dabei die Werte von 100 bis 0 absteigend.

```
# Beispielprogramm  gpio-intro.py (Fortsetzung)
... (Import, Setup)
gpio.setup(26, gpio.OUT)    # Pin 26 zur Datenausgabe
pwm = gpio.PWM(26, 1000)    # Frequenz: 1000 Hertz
pwm.start(0)                # Duty 0 (dunkel)
print('LED wird immer heller')
for duty in range(0, 101):
    pwm.ChangeDutyCycle(duty)
    time.sleep(0.02)        # 20 ms warten
print('LED wird immer dunkler')
for duty in range(100, -1, -1):
    pwm.ChangeDutyCycle(duty)
    time.sleep(0.02)        # 20 ms warten
pwm.stop()
```

### Zustand eines Tasters auswerten

Bei Signaleingängen liefert die `input`-Funktion den aktuellen Zustand `0` oder `1` (entspricht `LOW` oder `HIGH`). Aufgrund der Beschaltung mit den Pull-up-Widerständen (siehe Abbildung 17.1) bedeutet der Zustand `1`, dass der Taster *nicht* gedrückt ist.

```
status = gpio.input(21)     # 0 = gedrückt
                            # 1 = nicht gedrückt
```

**Pull-up- und Pull-down-Widerstände**

Pull-up- bzw. Pull-down-Widerstände verhindern, dass bei einer irrtümlichen Verwendung des Eingang-Pins als Ausgang (Output) ein hoher Strom fließt und unter Umständen den Raspberry Pi beschädigt. Eine Erläuterung dieser Schaltungstechnik können Sie in der Wikipedia nachlesen:

*https://de.wikipedia.org/wiki/Open_circuit#Beschaltung_der_Signalleitungen*





### Taster entprellen

Die Aufgabenstellung klingt trivial: Sie wollen einen Taster dazu verwenden, eine Leuchtdiode ein- und beim nächsten Drücken wieder auszuschalten. Dabei gibt es aber gleich zwei Probleme:

- Wie überwachen Sie den Zustand des Schalters? (Eine Schleife, die den Zustand ständig abfragt, würde viel zu viel CPU-Leistung verbrauchen.)
- Mechanische Schalter prellen, d. h. ein Metallplättchen schlägt beim Drücken innerhalb von Millisekunden mehrfach an den Kontakt. Wie verhindern Sie, dass das einmalige Drücken des Tasters von Ihrem Programm mehrfach gezählt wird?

Für das erste Problem sieht das `RPi.GPIO`-Modul zwei Funktionen vor. Zuerst geben Sie mit `add_event_detect` an, welchen Input-Pin Sie überwachen möchten. Im zweiten Schritt übergeben Sie an `add_event_callback` eine Funktion, die automatisch aufgerufen wird, wenn ein Pegelwechsel festgestellt wird. An diese Funktion wird je nach Setup die Pin-Nummer oder die BCM-Nummer übergeben. Eine erste Version des Programms kann so aussehen:

```
import RPi.GPIO as gpio

# LED ein- oder ausschalten
def turnLedOnOff(pin):
    # Pin-Nummer wird hier nicht ausgewertet
    global ledStatus
    ledStatus = 1 - ledStatus
    gpio.output(26, ledStatus)

# Setup
gpio.setmode(gpio.BOARD)    # Pin-Nummern verwenden
gpio.setup(26, gpio.OUT)    # Pin 26: LED
gpio.setup(21, gpio.IN)     # Pin 21: Schalter
ledStatus = 0
gpio.add_event_detect(21, gpio.FALLING, bouncetime=50)
gpio.add_event_callback(21, turnLedOnOff)
```

```
# auf Tastendruck warten
print('Programm endet nach 10 Sekunden')
time.sleep(10)
gpio.cleanup()
```

Während einer Zeit von 10 Sekunden wartet das Programm darauf, dass der Signalpegel von Pin 21 fällt (also von HIGH auf LOW wechselt), wenn Sie den Taster drücken. Bei jedem derartigen Pegelwechsel wird die Funktion `turnLedOnOff` aufgerufen. Sie invertiert die globale Variable `status` und schaltet die LED entsprechend ein oder aus. Beachten Sie, dass Sie bei der Funktion `turnLedOnOff` einen Parameter für die Pin-Nummer vorsehen müssen, auch wenn er wie im obigen Code nicht ausgewertet wird.

Allerdings werden Sie feststellen, dass das Programm wegen des Prellens des Tasters unzuverlässig funktioniert. Die LED wird jedes Mal (für das Auge kaum wahrnehmbar) ein paar Mal ein- und ausgeschaltet, und es ist vom Zufall abhängig, ob sie zum Schluss leuchtet oder nicht.

Die Funktion `add_event_detect` sieht einen optionalen Parameter `bounce=n` vor, um die Zeitspanne für den Entprellvorgang in Millisekunden anzugeben. Dieser Parameter funktioniert aber leider nicht wie vorgesehen.

Als Programmierer oder Programmiererin können Sie das Entprellen aber leicht selbst lösen – und das Beispielprogramm auf diese Weise ein wenig interessanter machen. Die Idee ist simpel: Sie merken sich den Zeitpunkt des letzten Schaltvorgangs und ignorieren alle weiteren Ereignisse, bis eine vorgegebene Entprellzeit vergangen ist. Diese Technik ist in der verbesserten Variante des Beispielprogramms realisiert.

```
# Beispieldatei gpio-bounce.py
import RPi.GPIO as gpio
import sys, time
from datetime import datetime, timedelta

# LED ein-/ausschalten
def turnLedOnOff(pin):
    global ledStatus, lastTime
    now = datetime.now()
```





```
    # 500 ms Entprellzeit
    if now - lastTime > timedelta(milliseconds=500):
        ledStatus = 1 - ledStatus
        gpio.output(26, ledStatus)
        lastTime = now

# Setup
gpio.setmode(gpio.BOARD)    # Pin-Nummern verwenden
gpio.setup(26, gpio.OUT)    # Pin 26 zur Datenausgabe
gpio.setup(21, gpio.IN)     # Pin 21 zur Dateneingabe
ledStatus = 0
gpio.output(26, ledStatus)  # LED anfänglich aus
lastTime = datetime.now()
gpio.add_event_detect(21, gpio.FALLING, bouncetime=50)
gpio.add_event_callback(21, turnLedOnOff)

# auf Tastendrücke warten
print('Programm endet nach 10 Sekunden')
time.sleep(20)
gpio.cleanup()
```

## 17.2 LED-Ampel für die CPU-Temperatur



Ausgangspunkt für dieses Beispiel ist eine Art LED-Ampel, die anzeigt, wie heiß die CPU des Raspberry Pi ist. Sie müssen dazu drei Leuchtdioden – nach Möglichkeit eine grüne (leuchtet bis 50 Grad), eine gelbe und eine rote (leuchtet ab 60 Grad) – mit drei GPIOs des Raspberry Pi verbinden. Ich habe die Pins 22, 24 und 26 mit den BCM-Nummern 25, 8 und 7 verwendet (siehe Abbildung 17.2).

Auf dem Raspberry Pi können Sie die CPU-Temperatur einer Systemdatei entnehmen. Diese enthält den Zahlenwert in Tausendstel Grad:

```
cat /sys/class/thermal/thermal_zone0/temp
  48312
```

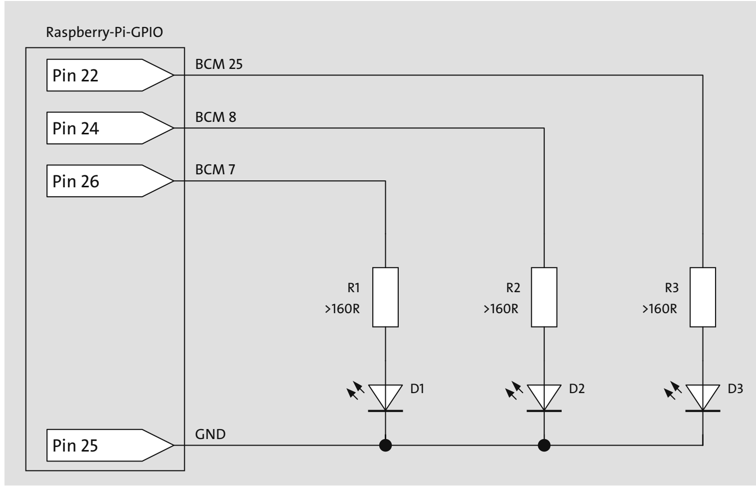


**Abbildung 17.2** Eine Leuchtdiodenampel

### Code

Das folgende Programm liest die Temperaturdatei einmal pro Sekunde ein und ruft dann die Funktion `turnOnOffLEDS` auf. Dort wird der gewünschte Status für die drei Leuchtdioden zuerst in den Variablen `statR`, `statY` und `statG` gespeichert. Diese Variablen werden dann in drei `output`-Aufrufen ausgewertet.

Das Programm läuft nach dem Start unbegrenzt weiter, kann aber mit `Strg`+`C` jederzeit beendet werden. Die `try/except`-Konstruktion kümmert sich darum, dass dabei keine unschönen Fehlermeldungen auftreten und alle GPIO-Pins wieder freigegeben werden.

```
# Beispieldatei gpio-cputemp.py
import RPi.GPIO as gpio
import time
```





```
# je nach Temperatur die entsprechende LED einschalten
def turnOnOffLEDs(temp):
    if temp < 50:
      (statR, statY, statG) = (0, 0, 1)
    elif temp > 60:
      (statR, statY, statG) = (1, 0, 0)
    else:
      (statR, statY, statG) = (0, 1, 0)
    gpio.output(r, statR)     # rot
    gpio.output(y, statY)     # gelb
    gpio.output(g, statG)     # grün

# Setup
gpio.setmode(gpio.BOARD)      # Pin-Nummern verwenden
(r, y, g) = (22, 24, 26)
for pin in [r, y, g]:
  gpio.setup(pin, gpio.OUT)   # als Ausgang verwenden
  gpio.output(pin, gpio.LOW)  # ausschalten

# LEDs einmal pro Sekunde aktualisieren
print('Programmende mit Strg+C')
fname = '/sys/class/thermal/thermal_zone0/temp'
try:
    while True:
        with open(fname, 'rt') as f:
            temp = int(f.readline()) / 1000
        print(temp)
        turnOnOffLEDs(temp)
        time.sleep(1)
except KeyboardInterrupt:
    print('Programmende')
finally:
    # GPIOs wieder freigeben
    gpio.cleanup()
```

### Test

Um das Programm auszuprobieren, installieren und starten Sie in einem zweiten Terminalfenster das Programm `sysbench`. Dieses Benchmark-Programm lastet die CPU für eine Weile voll aus. Die CPU-Temperatur steigt beinahe sofort auf über 50 Grad und überschreitet nach einigen Sekunden auch 60 Grad.

```
sudo apt install sysbench
sysbench --test=cpu --cpu-max-prime=20000 --num-threads=4
    run
```

## 17.3 GPIO-Zugriff mit »gpiozero«

Das `gpiozero`-Modul ist eine Alternative zu dem in den vorigen beiden Abschnitten präsentierten `RPi.GPIO`-Modul. Sein Hauptvorteil besteht darin, dass das Modul viel stärker objektorientiert ist. Für viele häufig benötigte Hardwarekomponenten gibt es eigene Klassen, beispielsweise:

- `LED` (Leuchtdiode)
- `PWMLED` (Leuchtdiode mit Software Pulse Width Modulation)
- `RGBLED` (dreifarbige LED, die über drei GPIO-Ausgänge gesteuert wird)
- `TrafficLights` (Kombination aus einer roten, gelben und grünen LED)
- `MotionSensor` (für PIR-Bewegungssensoren)
- `LightSensor` (Lichtdetektor)
- `Button` (Taster)
- `Buzzer` (Summer)
- `Motor` (zur Steuerung von zwei GPIOs für Vorwärts- und Rückwärts-Signale)
- `Robot` (zur Steuerung mehrerer Motoren)
- `MCP3008` (für den gleichnamigen A/D-Konverter)

Einen vollständigen Klassenüberblick finden Sie hier:

*https://gpiozero.readthedocs.io/en/stable/api_generic.html*





> **BCM-Nummern statt Pins**
>
> Beim `gpiozero`-Modul werden die GPIOs ausschließlich über die Nummern der BCM-Dokumentation adressiert, nicht über die Pin-Nummern!

Ob Sie die `gpiozero`-Bibliothek oder `RPi.GPIO` vorziehen, ist letztlich eine Geschmacksfrage. Im Internet dominieren Anleitungen für `RPi.GPIO`. Das hat damit zu tun, dass `RPi.GPIO` von Anfang an zur Verfügung stand, während die `gpiozero`-Bibliothek erst viel später entwickelt wurde.

### Nochmals die CPU-Temperatur visualisieren

Für eine ausführliche Beschreibung des `gpiozero`-Moduls fehlt hier der Platz. Stattdessen zeige ich Ihnen eine `gpiozero`-Variante des Programms aus Abschnitt 17.2, »LED-Ampel für die CPU-Temperatur«.

Das Listing demonstriert die Anwendung der `LED`-Klasse. Beim Erzeugen der Objekte geben Sie die BCM-Nummer an. In der Folge können Sie die Leuchtdiode mit den Methoden `on` und `off` ein- und ausschalten.

```
from gpiozero import LED
import time

# je nach Temperatur die entsprechende LED einschalten
def turnOnOffLEDs(temp):
    if temp < 50:
      ledG.on();  ledY.off(); ledR.off()
    elif temp > 60:
      ledG.off(); ledY.off(); ledR.on()
    else:
      ledG.off(); ledY.on();  ledR.off()

# Setup
ledR = LED(25)     # BCM-Nummern, entspricht Pin 22
ledY = LED(8)      # ... Pin 24
ledG = LED(7)      # ... Pin 26
```

17

```
for led in [ledR, ledY, ledG]:
  led.off()

# LEDs einmal pro Sekunde aktualisieren
fname = '/sys/class/thermal/thermal_zone0/temp'
try:
    while True:
        with open(fname, 'rt') as f:
            temp = int(f.readline()) / 1000
        turnOnOffLEDs(temp)
        time.sleep(1)
except KeyboardInterrupt:
    print('Programmende')
```

## 17.4 Sense HAT

Im letzten Abschnitt dieses Kapitels erkläre ich Ihnen, wie Sie das *Sense HAT* mit Python steuern. Dabei handelt es sich um ein Erweiterungs-Board für den Raspberry Pi (siehe Abbildung 17.3). Es stellt die folgenden Funktionen zur Verfügung:

- eine Matrix von 8 × 8 RGB-Leuchtdioden, die in verschiedenen Farben leuchten können
- einen Mini-Joystick
- diverse Sensoren für Beschleunigung, Rotation, Magnetismus, Temperatur, Luftfeuchtigkeit und Luftdruck

Die Abkürzung HAT steht für *Hardware Attached on Top*. Das bedeutet, dass das Board direkt auf die Steckerleiste des Raspberry Pi gesteckt werden kann. Die zur Ansteuerung des Sense HAT erforderlichen Bibliotheken inklusive des `sense_hat`-Moduls sind bei aktuellen Raspberry-Pi-OS-Versionen standardmäßig installiert.





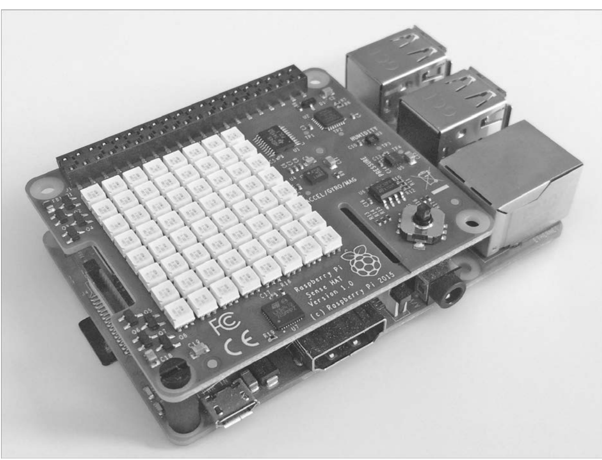

**Abbildung 17.3** Raspberry Pi mit aufgesetztem Sense HAT

**Sense-HAT-Simulator**

Sie können die Sense-HAT-Programmierung ganz ohne Raspberry Pi und Sense HAT auch in einem Simulator ausprobieren:

*https://trinket.io/sense-hat*



### »Hello, Sense HAT!«

Das `sense_hat`-Modul macht erste Tests des Boards denkbar einfach. Sie erzeugen ein Objekt der `SenseHat`-Klasse und können dann mit `show_message` eine Laufschrift anzeigen. Je nachdem, wie der Raspberry Pi Ihnen gegenüber positioniert ist, können Sie das Display durch die Veränderung der `rotation`-Eigenschaft in 90-Grad-Schritten rotieren.

```
# Beispieldatei hello-sense.py
from sense_hat import SenseHat
sense = SenseHat()
sense.rotation = 180  # Display-Rotation
sense.show_message('Hello, Sense HAT!')
```

#### Das Display Pixel für Pixel steuern

Mit `clear` können Sie das gesamte Display in einer Farbe zum Leuchten bringen. Die Farbe übergeben Sie als Tupel oder Liste mit drei Werten von 0 bis 255 für den Rot-, Grün- und Blau-Farbanteil. Das Display leuchtet nach dem Programmende weiter. Wenn Sie das nicht möchten, führen Sie `clear` ohne Parameter aus und schalten so alle LEDs aus.

```
# Beispieldatei sense-pixel.py
from sense_hat import SenseHat
import time
sense = SenseHat()
blue = (0, 0, 255)
sense.clear(blue)   # ganzes Display leuchtet blau
time.sleep(2)
sense.clear()       # Display ausschalten
```

Mit `set_pixel(x, y, farbe)` bringen Sie ein Pixel an einem beliebigen Koordinatenpunkt in der gewünschten Farbe zum Leuchten. Das Koordinatensystem hat seinen Ursprung links oben, die Achsen zeigen nach rechts und nach unten.

```
# LED links oben rot
red = (255, 0, 0)
sense.set_pixel(0, 0, red)
```

Wenn Sie alle 64 Pixel auf einmal verändern möchten, verwenden Sie die Methode `set_pixels`. Sie erwartet eine Liste mit 64 Farbtupeln. Die ersten 8 Listeneinträge gelten für die erste Zeile, die nächsten 8 für die zweite Zeile usw.

In den folgenden beiden Schleifen wird eine geeignete Liste zusammengestellt. Dabei ergibt sich ein Verlauf zwischen den Farben Schwarz, Rot, Grün und Weiß.

```
lst = []
for row in range(8):
    for col in range(8):
        lst += [(row*32, col*32, 0)]
sense.set_pixels(lst)
```





#### Pixel mit Joystick bewegen

Die Auswertung von Joystick-Bewegungen ist genauso einfach wie die Steuerung der LEDs. Mit `stick.wait_for_event` warten Sie darauf, dass ein Joystick-Ereignis eintritt. Anschließend werten Sie das `InputEvent`-Objekt aus, das `wait_for_event` zurückliefert:

```
from sense_hat import SenseHat
sense = SenseHat()
while True:
    event = sense.stick.wait_for_event()
    print('joystick %s %s' %
          (event.action, event.direction))
# Ausgabe beispielsweise:
#   joystick pressed right
#   joystick released right
#   joystick pressed middle
#   joystick released middle
```

Die Eigenschaften `action` und `direction` liefern simple Zeichenketten. Beachten Sie, dass die `direction`-Eigenschaft unabhängig von der im vorigen Abschnitt vorgestellten `rotation`-Eigenschaft ist. »Rechts« bezieht sich immer auf die Normallage des Raspberry Pi (Stromversorgung unten, USB- und Netzwerkanschlüsse rechts, GPIO-Leiste oben).

Das folgende Beispielprogramm beginnt damit, dass in der Mitte des Displays ein Pixel gesetzt wird. Mit dem Joystick können Sie dieses Pixel nun bis an den Rand des Displays bewegen – aber nicht darüber hinaus. Die Funktion `setLED` ist dafür zuständig, die Leuchtdiode an der gerade aktuellen Position rot leuchten zu lassen, diejenige an der bisherigen Position aber auszuschalten.

In der `while`-Schleife wartet das Programm auf das nächste Joystick-Ereignis, wobei das Loslassen (`'released'`) ignoriert wird. Sofern `x` und `y` den zulässigen Wertebereich nicht über- oder unterschreiten, werden die Koordinaten wunschgemäß angepasst.

```
# Beispielprogramm sense-joystick.py
from sense_hat import SenseHat
from time import sleep

# LED an aktueller Position einschalten, an der
# bisherigen ausschalten
def setLED(x, y):
    global oldX, oldY
    sense.set_pixel(oldX, oldY, (0, 0, 0)) # aus
    sense.set_pixel(x, y, (255, 0, 0))     # rot
    # neue Position merken
    (oldX, oldY) = (x, y)

# Setup
sense = SenseHat()
sense.clear()
(x, y)       = (4, 4)  # Startposition
(oldX, oldY) = (0, 0)  # vorige Position
setLED(x, y)

# Event-Loop, bis Strg+C gedrückt wird
try:
    while True:
        event = sense.stick.wait_for_event()
        if event.action == 'released':
            continue
        direct = event.direction
        if direct == 'left' and x>0:
            x -= 1
        if direct == 'right' and x<7:
            x += 1
        if direct == 'up' and y>0:
            y -= 1
        if direct == 'down' and y<7:
            y += 1
```





```
        # LED an neuer Position einschalten
        setLED(x, y)
except KeyboardInterrupt:
    sense.clear()
```

#### Pixel mit dem Gyroskop-Sensor bewegen

Das Sense HAT enthält ein Gyroskop, das die aktuelle Drehung des Raspberry Pi um dessen Achsen feststellt. Wenn Sie die Methode `get_orientation` ausführen, erhalten Sie drei Winkel mit einem Wertebereich von jeweils 0 bis 360 Grad. Uns interessieren hier nur zwei: `roll` gibt die Drehung um die Längsachse an, `pitch` die Rotation um die Querachse. Hintergrundinformationen zu diesen Winkeln finden Sie hier:

*https://projects.raspberrypi.org/en/projects/sense-hat-marble-maze/7*
*https://de.wikipedia.org/wiki/Roll-Nick-Gier-Winkel*

`roll` und `pitch` verraten uns also, ob der Raspberry Pi samt dem aufgesteckten Sense HAT eben liegt (dann sind beide Winkel nahe 0 bzw. 360) oder um seine Achsen gekippt wurde.

Zur anschaulichen Interpretation dieser Daten greifen wir das vorhin präsentierte Beispielprogramm nochmals auf. Aber anstatt die leuchtende LED nun durch den Joystick zu steuern, kippen Sie den Raspberry Pi in die entsprechende Richtung. Die leuchtende LED »rollt« wie eine Kugel in diese Richtung, bis das Ende des Displays erreicht ist. Das funktioniert auch diagonal, wenn der Minicomputer in beiden Achsen verdreht ist.



Am Beispielprogramm ändert sich nur die `while`-Schleife mit der Auswertung der `get_orientation`-Ergebisse:

```
# Beispieldatei sensor-motion.py
# ...
# Importe, setLED() und Setup wie in sensor-joystick.py
while True:
    o = sense.get_orientation()
    # Raspberry Pi nach links gekippt
    if 20 < o['pitch'] < 90 and x>0:
        x -= 1
```

```
# Raspberry Pi nach rechts gekippt
if 270 < o['pitch'] < 340 and x<7:
    x += 1
# nach hinten gekippt
if 270 < o['roll'] < 340 and y>0:
    y -= 1
# nach vorne gekippt
if 20 < o['roll'] < 90 and y<7:
    y += 1
# LED an neuer Position einschalten
setLED(x, y)
sleep(0.3)
```