Hauke Fehr · Stephan Elter

LET'S CODE Python

> Programmieren lernen ohne Vorkenntnisse

> Viele Beispiele mit Text und Grafik

> Mit allen Codebeispielen zum Download

Rheinwerk Computing

Kapitel 13
Funktionen selber schreiben

Du hast bereits eine Menge Funktionen in Python benutzt – sowohl solche, die fest eingebaut sind, als auch welche, die du vorher aus Modulen geladen hast. Funktionen machen das Programmieren übersichtlich. Jetzt ist es an der Zeit, eigene Funktionen zu basteln.

Eigene Funktionen zu schreiben ist einer der wichtigsten Schritte beim Programmieren – damit machst du deine Programme nicht nur kürzer, sondern auch klarer und leichter wartbar. Doch vorher schauen wir uns noch einmal an, was Funktionen eigentlich sind.

Was sind Funktionen noch mal genau?

Vielleicht kennst du Funktionen aus dem Matheunterricht, zum Beispiel:

$f(x) = x + 7$

Für jedes x definiert die Funktion f(x) einen Wert, der mit x+7 berechnet wird. Das ist auch die Grundidee von Funktionen in Python – allerdings können Funktionen viel mehr tun, als nur einen Wert zu berechnen.

Funktionen sind für die Programmiererin und den Programmierer wie Befehle, die etwas berechnen oder etwas ausführen. Selbst geschriebene Funktionen sind Befehle, die über die Standardbefehle von Python hinausgehen und spezielle Dinge leisten können. Eine Funktion führt ein kleines Programm aus und liefert bei Bedarf dann einen berechneten Wert zurück. Es gibt Funktionen, die bereits in Python eingebaut sind, wie zum Beispiel `print(wert)` oder `input("Text")`, und es gibt unzählige Funktionen, die man sich über Module hinzuladen kann, wie zum Beispiel `randint(x,y)` aus dem Modul `random` oder `sin(x)` aus dem Modul `math` oder `forward(x)` aus dem Modul `turtle`. Unbegrenzt werden die Möglichkeiten dadurch, dass man auch jederzeit einfach eigene Funktionen definieren kann.

In manchen Programmiersprachen wird zwischen Prozeduren und Funktionen unterschieden. Prozeduren sind wie Befehle, die nur etwas ausführen. Sie können allein aufgerufen werden:

```
print("Hallo")
```

gibt den Text »Hallo« aus.

»Echte« Funktionen werden hingegen wie Variablen verwendet, weil sie einen Wert zurückliefern:

```
x = input("Bitte Wert eingeben")
```

liefert den eingegebenen Wert zurück und schreibt ihn in die Variable `x`.

In Python unterscheidet man aber nicht zwei verschiedene Typen (Prozeduren und Funktionen). Beides sind dort einfach *Funktionen* und man kann sie mit oder ohne Rückgabewert verwenden – sie führen entweder nur einfach etwas aus oder sie liefern uns einen Wert zurück, den wir verwenden können. Zum Beispiel kannst du die `input()`-Funktion auch ohne Variable benutzen:

```
input("Das ist ein Eingabefenster")
```

Auch so öffnet sich ein Fenster – aber egal, was du darin eingibst, wird der Wert nicht gespeichert, weil er keiner Variablen zugewiesen wird.

Aufruf von Funktionen

Der Aufruf einer Funktion besteht immer aus dem *Funktionsnamen* mit einem darauffolgenden runden Klammernpaar, in dem die Werte stehen, die man der Funktion mitgibt (*Parameter*). Es kann ein Wert in den Klammern stehen, wie bei `sin(0.8)`. Es können mehrere Werte in den Klammern übergeben werden, die man durch ein Komma voneinander trennt, wie bei `randint(1,6)`. Oder die Klammern können auch leer bleiben, wenn man der Funktion keinen Wert übergeben will, weil sie vielleicht ohnehin immer nur dasselbe macht oder keinen Wert braucht. Das hängt natürlich immer davon ab, wie die Funktion programmiert ist und welche Parameter sie erwartet.

```
input()
```

ohne Zuweisung und ohne Parameter würde übrigens auch funktionieren. Dann öffnet sich eben ein Eingabefenster ohne Text und der eingegebene Wert wird auch nicht gespeichert. Macht eigentlich keinen Sinn, ist aber erlaubt.

Eigene Funktionen schreiben

Es gibt unzählige Funktionen in den Python-Modulen, die du jederzeit verwenden kannst. Damit du aber richtig verstehst, was Funktionen genau sind, ist es sinnvoll, dass du eigene Funktionen schreibst, die du dann auch verwenden kannst. Später wird das für dich ohnehin ein ganz selbstverständlicher Teil des Programmierens sein.

Fangen wir ganz einfach an: Wir möchten eine Funktion mit dem Namen »verdoppeln« schreiben, die einen Zahlenwert ganz einfach verdoppelt. Wie definieren wir nun diese Funktion?

Dazu gibt es einen Python-Befehl, der heißt

```
def Funktionsname(Variablen):
```

Mit `def` definiert man eine Funktion. Der Doppelpunkt am Ende weist darauf hin, dass danach ein eingerückter Block steht, der zur Funktion gehört.

```
def verdoppeln(wert):
```

So definierst du eine neue Funktion mit dem Namen `verdoppeln`, die eine Variable mit dem Namen `wert` erhält. Man nennt die Werte, die man einer Funktion übergibt, auch Übergabeparameter bzw. Parameter oder Argument. Hier haben wir also einen Parameter, der der Funktion als Variable `wert` übergeben wird.

Danach folgt das, was die Funktion macht. Was macht sie? Sie nimmt die Variable `wert` mal 2 und liefert diesen doppelten Wert dann zurück. Das war's.

So sieht das in Python zum Beispiel aus:

Definiton
Berechnung
Rückgabe des Ergebnisses

```
def verdoppeln(wert):
    doppelwert = wert*2
    return doppelwert
```

Mit `def` wird die Funktion also definiert, mit `return` wird das Ergebnis der Funktion zurückgegeben.

Sobald du diese Funktionsdefinition eingegeben hast, kannst du die Funktion `verdoppeln` wie einen Befehl in Python verwenden, an jeder Stelle in deinem Programm und beliebig oft. Du musst nur beachten: Erst nachdem eine Funktion im Programm definiert wurde, kannst du sie verwenden. Sonst gibt es einen Fehler, weil die Funktion noch unbekannt ist.

Das kann zum Beispiel so aussehen:

```
def verdoppeln(wert):
    doppelwert = int(wert)*2
    return doppelwert
x = input("Bitte Wert eingeben")
print(verdoppeln(x))
```

Alles klar? Es wird ein Wert eingegeben und in `x` gespeichert, dann wird das Doppelte ausgegeben, indem die neue Funktion `verdoppeln` mit `x` als übergebenem Parameter verwendet wird. Wie gesagt kann eine Funktion auch mit mehreren Parametern (also übergebenen Werten) arbeiten, wenn sie so definiert wird. Schreib eine Funktion `summe`, die zwei Werte erhält, die sie dann addiert und als Summe zurückgibt. Schaffst du das selber?

Hier der mögliche Code für die Funktion:

```
def summe(wert1,wert2):
    zusammen = int(wert1)+int(wert2)
    return zusammen
```

Und im ganzen Programm wäre es dann so verwendbar:

```
def summe(wert1,wert2):
    zusammen = int(wert1)+int(wert2)
    return zusammen
x = input("Bitte Wert 1 eingeben")
y = input("Bitte Wert 2 eingeben")
print(summe(x,y))
```

Das ist verständlich, oder?

Eine einfache Funktion, die keinen Wert braucht, könntest du zum Beispiel so schreiben:

```
def warnung():
    print("Achtung, hier ist ein falscher Wert eingegeben worden!")

x = int(input("Bitte Zahl unter 10 eingeben"))
if x<10:
    print("Danke")
else:
    warnung()
```

In diesem Programm haben wir uns also am Anfang einen neuen Befehl definiert: die Funktion `warnung()`.

Diese Funktion braucht keinen Wert und sie liefert auch keinen zurück. Alles, was sie macht, ist die Ausgabe eines Warntextes.

Wozu brauchen wir also Funktionen?

Funktionen sind aus mehreren Gründen sehr sinnvoll beim Programmieren. Zum einen machen sie Programme viel übersichtlicher. Eine Funktion kann viele Befehle und Berechnungen nacheinander enthalten – und sie wird im Programm dann nur mit einem einzigen Befehl aufgerufen. Wenn der Funktionsname deutlich genug ist, sind die Programme viel einfacher zu lesen.

Beispiel:

```
if antwort == richtig:
    zeige_smiley()
else:
    spiele_falschsound()
```

Dieses Programm versteht man sofort, weil es zwei klar benannte Funktionen enthält: `zeige_smiley()` und `spiele_falschsound()`

Jede dieser beiden Funktionsdefinitionen (die hier nicht mit aufgeführt sind) ist vielleicht selbst lang und kompliziert, etwa weil möglicherweise ein Smiley aus mehreren Elementen auf eine Fläche gezeichnet und ein Klang berechnet und abgespielt wird. Aber das braucht uns nicht mehr zu interessieren, wenn die Funktionen erst mal geschrieben sind. Danach werden sie nur noch wie *neue Befehle* verwendet, die genau das machen, was wir festgelegt haben.

Der andere große Vorteil von Funktionen ist, dass man sie in einem Programm beliebig oft aufrufen kann, und das an den unterschiedlichsten Stellen. Man muss also einen Programmcode für eine bestimmte Aufgabe nur **einmal schreiben** und kann ihn dann als Funktionsbefehl überall immer wieder aufrufen. Das macht Programme natürlich auch kürzer und viel einfacher.

Funktionen in eigenem Modul speichern

zB

Jede selbst definierte Funktion (oder auch eine ganze Liste von definierten Funktionen) kannst du selbst als Modul in einer Datei speichern. Diese Datei musst du dann nur noch mit dem `import`-Befehl in dein Programm laden und kannst dann alle Funktionen, die darinstehen, sofort verwenden. Du hast das ja schon öfter gemacht, indem du die mitgelieferten Module `random`, `math` oder `turtle` importiert hast. Ebenso geht es auch mit eigenen, selbst erfundenen Modulen.

Eigene Funktion »zahlwort«

Jetzt weißt du, wie Funktionen selbst geschrieben werden können. Die Beispiele waren natürlich sehr simpel und dienten nur dem Verständnis. Richtig nützliche Funktionen sind etwas länger und komplizierter – sonst bräuchte man sie nicht.

Daher nun eine kleine Herausforderung: Wir wollen eine Funktion schreiben, die eine Zahl in ein deutsches Zahlwort umwandelt. Also aus 1 wird »eins«, aus 2 wird »zwei« usw. bis 100.

Los geht's, wir definieren eine Funktion:

```
def zahlwort(zahl):
```

Die Funktion heißt also `zahlwort` und sie erhält als Parameter eine Zahl in der Variablen `zahl`, die zwischen 1 und 100 liegen muss.

Ups … wie macht man das denn jetzt? Wie kann man eine Zahl zwischen 1 und 100 in ein deutsches Wort umwandeln?

Du kommst sicher nach etwas Überlegung von selbst darauf: Wir brauchen hier auch wieder Listen. Klar, am simpelsten wäre eine Liste, die alle Zahlen von 1 bis 100 als Wörter enthält. Aber das wäre sehr viel Tipparbeit und würde eine ziemlich lange Liste ergeben.

Programmierer sind faul … ich meine effizient. Das heißt, wir sollten das Ganze vereinfachen, um es kürzer zu machen.

Die Zahlen von 1 bis 19 brauchen wir als Liste, da kommen wir nicht drumherum, denn die sind nicht einfach zusammensetzbar.

Ab 20 brauchen wir nur noch die Zehner und die Einer und können die Zahl daraus zusammensetzen (also `"fünf"+"dreißig"` oder `"neun"+"neunzig"`).

Ich zeige dir hier mal eine Möglichkeit, wie die Listen aussehen könnten, mit denen wir dann arbeiten werden:

```
einer = ["null", "ein", "zwei", "drei", "vier", "fünf", "sechs", "sieben",
         "acht", "neun", "zehn", "elf", "zwölf", "dreizehn", "vierzehn",
         "fünfzehn", "sechzehn", "siebzehn", "achtzehn", "neunzehn"]
zehner = ["null", "zehn", "zwanzig", "dreißig", "vierzig", "fünfzig",
          "sechzig", "siebzig", "achtzig", "neunzig", "hundert"]
```

Wenn die Zahl unter 20 ist, wird das dazugehörige Wort einfach aus der »einer«-Liste genommen. Das ist simpel. Nur einen Sonderfall gibt es – die eins, die lautet dann »eins« und nicht »ein«.

```
if zahl < 20:
    wort = einer[zahl]
    if zahl == 1:
        wort = "eins"

return wort
```

Alles klar?

Aber das reicht ja noch nicht. Was ist, wenn die Zahl größer als 19 ist? Sie muss dann in Zehner und Einer auseinandergenommen werden. Daraus setzt sich dann das Wort zusammen.

Wie nimmt man eine Zahl in Zehner und Einer auseinander?

Ganz einfach, indem man sie mit einer Ganzzahldivision durch zehn teilt, das sind die Zehner, und dann noch einmal durch Modulo-Division den Rest ermittelt, das sind die Einer. Wir hatten so etwas Ähnliches schon ganz am Anfang in unserem Rechner mit Rest.

```
z = zahl // 10
e = zahl % 10
```

Die Variable `z` enthält jetzt also die Zehner der Zahl, die Variable `e` enthält die Einer. Damit können wir jetzt das Zahlwort zusammensetzen:

```
wort = einer[e]+"und"+zehner[z]
```

Das Einerwort, dann ein »und« und das Zehnerwort – so werden Zahlen auf Deutsch gebildet. Also »fünf« + »und« + »zwanzig« – oder »neun« + »und« + »neunzig«.

Auch hier gibt es wieder eine Ausnahme: Was, wenn die Einer null sind? »Nullundvierzig« ist kein korrektes Zahlwort.

Also muss dieser Fall jetzt auch noch geprüft und gesondert behandelt werden:

```
    if e == 0:
        wort = zehner[z]
```

Ganz einfach: Wenn die Einer 0 sind, dann besteht das Zahlwort nur aus den Zehnern.

Am Schluss gibt unsere Funktion die zusammengebaute Variable `wort` zurück und ist fertig. So sieht die gesamte Funktion aus:

```
def zahlwort(zahl):

  einer = ["null", "ein", "zwei", "drei", "vier", "fünf", "sechs", "sieben",
        "acht", "neun", "zehn", "elf", "zwölf", "dreizehn", "vierzehn",
        "fünfzehn", "sechzehn", "siebzehn", "achtzehn", "neunzehn"]
  zehner = ["null", "zehn", "zwanzig", "dreißig", "vierzig", "fünfzig",
         "sechzig","siebzig", "achtzig", "neunzig", "hundert"]

  if zahl < 20:
        wort = einer[zahl]
        if zahl == 1:
            wort = "eins"
  else:
        z = zahl // 10
        e = zahl % 10
```

```
        wort = einer[e]+"und"+zehner[z]
        if e == 0:
            wort = zehner[z]
    return wort
```

Einrückungen mit Leerzeichen

Achte immer darauf, dass alle Einrückungen, die zu einer Anweisung gehören, gleich sind. Empfohlen sind jeweils vier Leerzeichen, aber das ist eben nur eine Empfehlung.

Jetzt kannst du die Funktion benutzen. Schreib unter die Funktion ein ganz kurzes Programm, in dem du eine Zahl `x` eingibst und als Ausgabe dann das `zahlwort` von `x` erhältst, zum Beispiel so:

```
x = int(input("Zahl:"))
print(str(x) + " ist in Worten: " + zahlwort(x))
```

Und jetzt geht's ans Testen: Gib verschiedene Zahlen zwischen 0 und 100 ein.

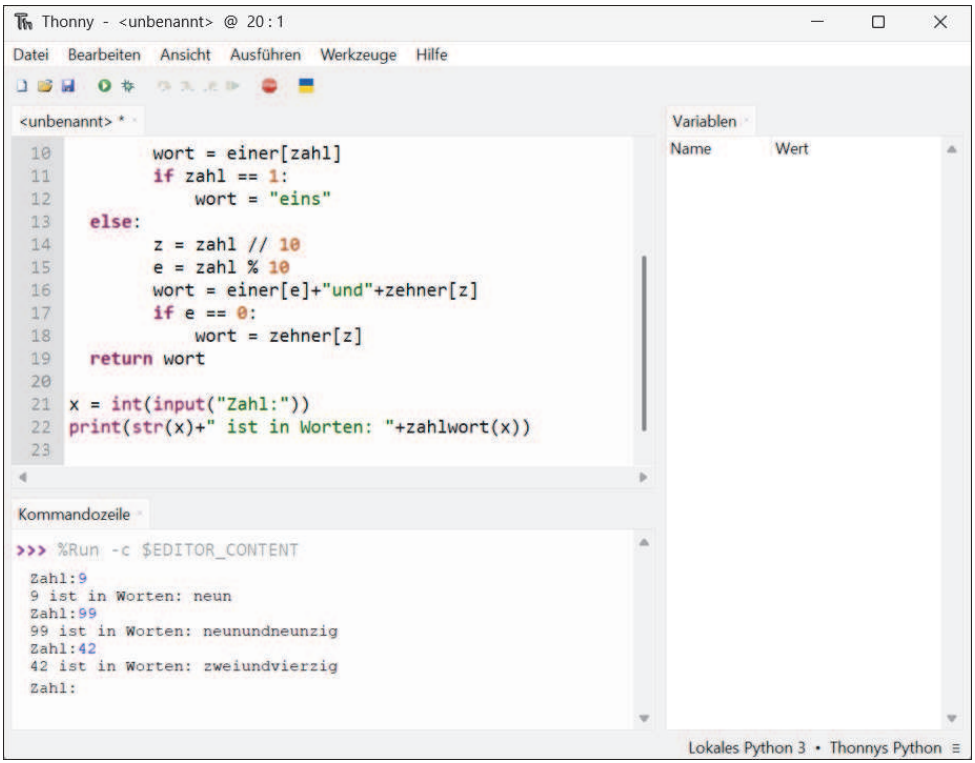


Abbildung 13.1: Zahlen in Wörter verwandeln mit der selbst gemachten Funktion »zahlwort«

Toll! Es funktioniert, solange man Zahlen zwischen 0 und 100 eingibt. Natürlich könntest du die Funktion jetzt erweitern, zum Beispiel auf Zahlen bis 1.000 oder bis 1 Million. Dann müsstest du noch etwas mehr Aufwand betreiben, denn ab 100 werden die Zahlen wieder anders zusammengesetzt. Du könntest die Funktion auch einfach absichern: Wenn die Zahl größer als 100 ist, liefert die Funktion »Zahl zu groß« zurück.

Du könntest auch prüfen, ob die Zahl kleiner als 0 ist – dann wird sie einfach in eine positive Zahl umgewandelt (mit –1 multipliziert) und ein »minus« dem Zahlwort vorangestellt. Du kannst hier frei experimentieren, wenn du möchtest.

Ein eigenes Modul erstellen

Wenn deine Funktion fertig ist und du sie vielleicht auch in anderen Programmen verwenden möchtest, kannst du dir ein eigenes Modul basteln. Das kannst du dann später in jedes beliebige Programm importieren und damit die Funktion daraus verwenden. Auf diese Weise kannst du dir allmählich deinen eigenen »Baukasten« zusammensetzen, der für jedes neue Programm wieder einsetzbar ist. Wir probieren das hier einmal mit unserer `zahlwort`-Funktion aus.

Der Code sieht jetzt also so aus:

```
def zahlwort(zahl):

  einer = ["null", "ein", "zwei", "drei", "vier", "fünf", "sechs",
           "sieben", "acht", "neun", "zehn", "elf", "zwölf", "dreizehn",
           "vierzehn", "fünfzehn", "sechzehn", "siebzehn", "achtzehn",
           "neunzehn"]

  zehner = ["null", "zehn", "zwanzig", "dreißig", "vierzig",
            "fünfzig","sechzig","siebzig","achtzig", "neunzig","hundert"]

  if zahl < 20:
        wort = einer[zahl]
        if zahl == 1:
          wort = "eins"
        return wort
  else:
        z = zahl // 10
        e = zahl % 10
        wort = einer[e]+"und"+zehner[z]
        if e == 0:
            wort = zehner[z]
  return wort
```

Diesen Code speicherst du jetzt als Datei unter dem Namen *zwort.py* ab. *zwort* ist dabei der Name des Moduls. Die Endung *.py* kennzeichnet, dass es ein Python-Programm ist. Achte darauf, dass du die Datei im selben Verzeichnis speicherst wie alle deine anderen Python-Programme, insbesondere das, das dieses Modul importieren will.

Um dein eigenes Modul jetzt in einem neuen Programm zu verwenden, gehst du genauso vor wie bei der Verwendung von `random`, `math` oder `turtle`, zum Beispiel so:

```
import zwort
x = int(input("Zahl eingeben: "))
print(zwort.zahlwort(x))
```

Oder so:

```
from zwort import *
x = int(input("Zahl eingeben"))
print(zahlwort(x))
```

Bei der ersten Lösung wird dein Modul `zwort` Python als Bibliothek zur Verfügung gestellt. Um die Funktion `zahlwort()` zu verwenden, musst du immer zuerst den Modulnamen schreiben, dann einen Punkt, dann den Funktionsnamen, also

```
zwort.zahlwort(x)
```

Bei der zweiten Variante importierst du alle Funktionen aus dem Modul `zwort` direkt in dein Python-Programm (in diesem Fall ist es ja nur eine Funktion) – deswegen musst du den Modulnamen `zwort` nicht mehr voranstellen.

Also alles genauso, wie du es schon kennst – nur eben mit deinem eigenen Modul.

Achtung: Das Modul im selben Verzeichnis speichern

Damit Python das Modul wirklich findet, musst du das neue Programm vor dem Ausführen erst einmal abspeichern, und zwar im selben Verzeichnis wie die Moduldatei. Dann kann das Modul problemlos importiert werden, weil Python nun weiß, wo es gesucht werden muss.

Zeichnen mit Funktionen

Zum Abschluss der Funktionen probieren wir noch einmal aus, wie man auch beim Zeichnen mit `turtle` eigene Funktionen nutzen kann.

Erinnerst du dich, wie man ein Quadrat zeichnet? Es werden da einfach vier Linien von gleicher Länge gemalt, nach jeder Linie dreht sich die Schildkröte um 90 Grad nach rechts. Machen wir eine Funktion daraus:

```
def quadrat(laenge):
    for i in range(4):
        t.forward(laenge)
        t.right(90)
```

So, mit dieser Funktion kannst du jetzt jederzeit ein Quadrat mit beliebiger Seitenlänge zeichnen.

```
import turtle as t
quadrat(100)
t.done()
```

Das zeichnet ein Quadrat mit der Seitenlänge 100 Pixel ab der Stelle, an der die Schildkröte gerade steht. Ein Dreieck könnte man ebenso als Funktion definieren:

```
def dreieck(laenge):
    for i in range(3):
        t.forward(laenge)
        t.right(120)
```

Die Funktion macht genau das Gleiche wie die letzte, nur für ein Dreieck.

Wie wäre es jetzt mit einer universellen Funktion, also einer Vieleckfunktion, die Dreiecke, Vierecke, Fünfecke, Sechsecke usw. zeichnen kann? Das hatten wir in Kapitel 12, »Die Schildkröte – ein grafischer Roboter«, auch schon einmal, nur dass wir das Ganze jetzt als Funktion definieren.

Diesmal brauchen wir zwei Parameter, die wir der Funktion mitgeben: einmal die Anzahl der Ecken und einmal die Länge der Seiten.

```
def vieleck(ecken,laenge):
    winkel = 360/ecken
    for i in range(ecken):
        t.forward(laenge)
        t.right(winkel)
```

Der erforderliche Winkel wird berechnet, indem man eine ganze Umdrehung (360 Grad) durch die Anzahl der Ecken teilt, daraus ergibt sich der Rest.

Nun kannst du ein Programm schreiben, das 100 Zufallsobjekte zeichnet. Jedes Objekt hat zwischen drei und sieben Ecken und eine Seitenlänge zwischen 10 und 100. Auch die Posi-

tion und die Farbe können zufällig gesetzt werden (mit den Funktionen von `turtle`). Und zum Zeichnen wird einfach immer nur die Funktion `vieleck()` verwendet. Versuch mal, so ein Programm zu schreiben. Erst mal nur in ganz einfacher Fassung, danach kannst du es weiter ausbauen.

```
import turtle as t
import random
t.hideturtle()
t.pensize(3)

def vieleck(ecken, laenge):
    winkel = 360 / ecken
    for _ in range(ecken):
        t.forward(laenge)
        t.right(winkel)

for _ in range(100):
    # Zufallsfarbe
    r = random.random()
    g = random.random()
    b = random.random()
    t.color(r, g, b)

    # zufällige Startrichtung
    t.setheading(random.randint(0, 359))

    # zufällige Position
    x = random.randint(-400, 400)
    y = random.randint(-300, 300)
    t.penup()
    t.goto(x, y)
    t.pendown()

    # zufällige Parameter
    laenge = random.randint(10, 100)
    ecken = random.randint(3, 7)
    vieleck(ecken, laenge)

t.done()
```

Diese Variante ist zum Beispiel eine Möglichkeit. Wir verwenden dabei eine zufällige Richtung und eine passende Stiftdicke, damit die Zeichnung lebendiger wirkt. Du kannst das natürlich auch ganz anders gestalten, wenn du möchtest. Am Ende entsteht auf jeden Fall ein farbenfrohes Bild aus vielen unterschiedlichen Figuren.

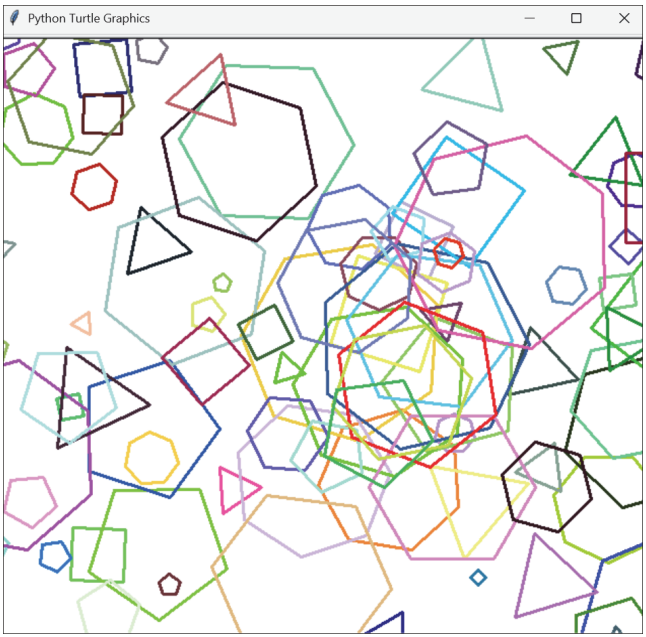


Abbildung 13.2: Moderne Kunst? Zufällige Vielecke in verschiedenen Farben, Positionen und Winkeln

Rekursive Funktionen

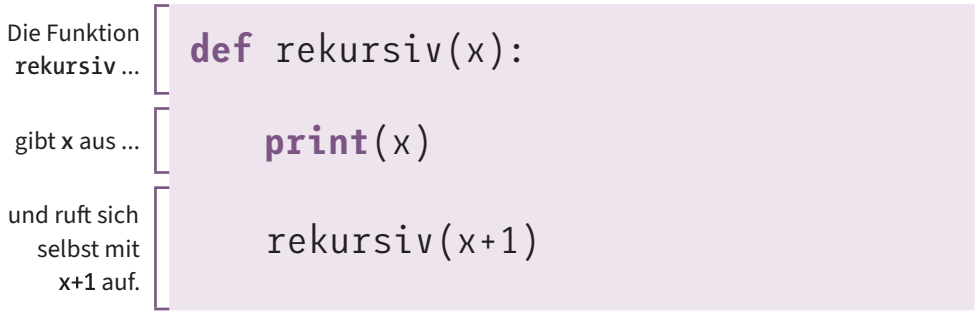
Zum Abschluss der Funktionen in Python noch eine Besonderheit, mit der professionelle Programmierer regelmäßig zu tun haben: sogenannte *rekursive Funktionen*. Was ist das?

Rekursive Funktionen sind Funktionen, die sich selbst aufrufen.

Wie? Sie rufen sich selbst auf? Wie soll denn das gehen?

Das geht sehr einfach, aber es gibt ein paar Dinge zu beachten und es ist nicht immer ganz leicht, sich klarzuwerden, was genau da passiert.

Hier ist eine einfache Funktion, die sich selbst aufruft:



Die Funktion `rekursiv()` wird mit dem Übergabewert `x` aufgerufen. Nehmen wir an, `x` ist beim Aufruf 1. Nun wird `x` ausgegeben (also 1), dann wird die Funktion selbst (`rekursiv(x+1)`) erneut aufgerufen, allerdings diesmal mit dem Wert 2. Die Funktion `rekursiv()` ruft sich also selbst auf und ist damit nicht beendet, sondern der nächste Aufruf gibt eine 2 aus, ruft sich dann wieder selbst auf, diesmal mit dem Wert 3 usw.

Die Funktion ist also ein Zähler, der immer eine Zahl höher zählt und sich selbst dann weiter aufruft. Du siehst wahrscheinlich schon, was das Problem dabei ist. Die Funktion wird nie beendet – stattdessen ruft sie sich bis ins Unendliche immer wieder selbst auf …

Bis ins Unendliche? Wie oft kann eine Funktion sich selbst aufrufen, ohne dass die Programm-Engine ein Problem bekommt? Denn sie muss sich ja schließlich immer »merken«, wo sie gerade ist, welche Funktion gerade welche aufgerufen hat … Das kann nicht unendlich lange gut gehen.

Probiere es einmal aus: Gib die Funktion von oben ein und füge am Ende noch einen Aufruf der Funktion hinzu:

```
def rekursiv(x):
    print(x)
    rekursiv(x+1)

rekursiv(1)
```

Was passiert? Der Zähler zählt hoch bis ca. 1.000, dann kann das Programm nicht mehr und bricht ab:

```
RuntimeError: maximum recursion depth exceeded
```

Also: *Laufzeitfehler. Die maximale Rekursionstiefe wurde überschritten*. Irgendwann reicht es dem Python-Interpreter und er hat »keine Lust mehr«, sich ewig selbst aufzurufen und sich dabei stets zu merken, auf welcher Ebene er sich jetzt befindet. Ganz konkret gibt es irgendwann ernste Speicherprobleme und der Computer würde »abstürzen«, wenn er nicht abbricht.

Wie löst du das Problem?

Indem du eine Bedingung einbaust, wann weiter aufgerufen und wann abgebrochen wird. Nehmen wir an, der Zähler soll bis 100 zählen – die Funktion soll sich also nur so lange selbst aufrufen, bis die 100 erreicht ist.

```
def rekursiv(x):
    print(x)
    if x<100:
        rekursiv(x+1)

rekursiv(1)
```

Schon besser, jetzt läuft das Programm schnell durch und zählt von 1 bis 100.

Natürlich wollte ich mit diesem Programm nur zeigen, was Rekursion ist. In der Praxis würde man so einen Zähler wohl anders programmieren. Aber es gibt Aufgaben, bei denen Rekursion durchaus sinnvoll ist. Manche Probleme lassen sich sogar nur durch Rekursion angemessen lösen – oder zumindest viel einfacher.

Aufgabe

Eine neue Aufgabe: Schreib eine rekursive Funktion, die die mathematische Fakultät einer Zahl berechnet.

Was ist die Fakultät?

Fakultät bedeutet, dass eine Zahl mit allen positiven ganzen Zahlen über 0 multipliziert wird, die kleiner sind als sie. Die Fakultät von 5 ist somit 5 × 4 × 3 × 2 × 1 = 120.

Klar, man könnte das in Python auch mit einer Zählerschleife lösen, aber die rekursive Version ist elegant und etwas raffinierter.

```
def fakultaet(x):
    if x == 1:
        return 1
    else:
        return (x * fakultaet(x-1))
```

Verstehst du, wie das funktioniert? Man muss sich beim Denken ein bisschen verrenken. Nehmen wir an, die Funktion wird mit dem Wert 3 aufgerufen. Da 3 nicht gleich 1 ist, wird das `if`-Kommando übersprungen. Jetzt liefert die Funktion mit `return` etwas zurück, aber sie endet noch nicht, denn für die Ermittlung des Rückgabewertes muss sich die Funktion erst einmal selbst aufrufen, diesmal mit der 2 als Wert. Dort passiert wieder das Gleiche, die Funktion ruft sich beim `return` wiederum selbst auf und dann startet sie mit der 1 als Wert.

Hier liefert sie nun einfach die 1 zurück und ruft sich nicht mehr selbst auf. Damit wird der zweite Aufruf auch beendet (1 × 2) und der dritte ebenso (2 × 3) – und die Funktion kann endlich beendet werden, das Ergebnis 6 kommt zurück.

Um die Funktion mit verschiedenen Werten auszuprobieren, kannst du die folgenden Zeilen dazuschreiben:

```
f = int(input("Zahl eingeben: "))
print("Die Fakultät von",f,"ist",fakultaet(f))
```

Vorsicht: Die Ergebnisse werden schnell sehr hoch!

Für Mathe-Profis: Wozu braucht man die Fakultätsberechnung eigentlich?

Die Fakultät spielt eine wichtige Rolle in der Wahrscheinlichkeitsrechnung. Man kann hiermit zum Beispiel die möglichen Anordnungen einer Anzahl von Dingen berechnen. Wenn ich fünf Personen habe, gibt es 120 (Fakultät von 5) Reihenfolgen, in denen sie sich auf eine Bank setzen können. Zehn Personen haben bereits über 3,5 Millionen Möglichkeiten, wie sie sich auf der Bank anordnen können. Unglaublich, aber beweisbar wahr!

Kapitel 14

Sound programmieren

Bevor es an die Objektprogrammierung und dann an die ersten richtigen Spiele geht, wollen wir uns zunächst mit Klang beschäftigen – denn durch Klänge werden Programme erst richtig lebendig!

Nicht nur Spiele, sondern auch jedes Programm kann Klänge und Geräusche oder auch Musik abspielen und manche können sogar sprechen. Ein Programm, das nicht nur das Auge anspricht, sondern das man auch hören kann, macht einfach mehr her.

Sound in Python abspielen

Alles, was du dazu brauchst, ist eine Klangdatei mit dem Sound, den du haben willst – selbst aufgenommen geht auch! Die kannst du dann in dein eigenes Programm einbinden, und schon kann dein Programm mithilfe des Moduls `pygame-ce` jedes beliebige Geräusch oder Musik und Sprache abspielen.

Dazu solltest du am besten etwas mehr darüber wissen, was eine Klangdatei auf dem Computer ist. Wenn du das alles schon weißt, kannst du den nächsten Abschnitt gern überspringen.

Was sind denn eigentlich Klangdateien?

Klangdateien sind digitalisierte Geräusche, die zu einer Einheit aus Amplitudenwerten zusammengefasst wurden. Um Klänge zu erzeugen, muss ein Lautsprecher in einer bestimmten Geschwindigkeit (das ist die Frequenz) hin- und herschwingen (die Stärke der Schwingung nennt man Amplitude). Wenn du vor einer empfindlichen Membran singst, dann wird diese anfangen zu schwingen, je nachdem, welche Töne du singst – je lauter, desto stärker, je höher, desto schneller. Wenn diese Membran ihre Schwingungen in elektrische Spannungen umsetzt, die man messen und aufzeichnen kann, hast du ein Mikrofon. Wenn ein Lautsprecher nun diese Schwingungen genauso, wie das Mikrofon sie aufgezeichnet hat, wieder in seinen Membranen erzeugen kann, dann wird eine Aufnahme abgespielt. Das war früher analog: Auf Schallplatten ritzt eine Nadel während der Schwingungen mehr oder weniger tief in das Vinyl, auf Tonbändern werden die Schwingungen, während das Band vorbeiläuft, durch unterschiedlich starke Magnetisierung festgehalten.

Digital läuft das alles prinzipiell genauso, wobei die Schwingungen des Mikrofons oder Aufnahmegeräts bei der Aufnahme in Zahlen umgesetzt werden, die in extrem schneller Folge, zum Beispiel 44.100-mal pro Sekunde, gemessen und gespeichert werden (das entspricht dem Standard einer Musik-CD, 44,1 kHz). Das heißt, 1 Sekunde Klang besteht aus 44.100 Zahlen, die nacheinander genau beschreiben, wie weit die Membran in welche Richtung ausschlägt. Wenn der Computer diese 44.100 Zahlen nun in der gleichen Geschwindigkeit wieder an den Lautsprecher sendet und dieser die Zahlen wieder in Membranausschläge umsetzt, ertönt 1 Sekunde lang wieder exakt der aufgenommene Klang.

Das Dateiformat, bei dem jede Zahl genau einem Membranzustand entspricht, nennt man auch WAV-Format, das steht für *wave* (Welle). Klangdateien im WAV-Format erkennst du in Windows an der Endung *.wav*; sie waren früher die Standarddateien, um Klänge zu speichern oder abzuspielen.

Da WAV-Dateien aber bei langen Musikstücken sehr viel Speicherplatz einnehmen (wie gesagt für jede Sekunde 44.100 Zahlen und für Stereo genau doppelt so viel, denn da spielen ja zwei Lautsprecher unterschiedliche Versionen des Klanges ab), hat man zunächst vor allem für die Übertragung über das Internet irgendwann ein Format erfunden, das den (nahezu) gleichen Klang mit viel weniger Daten abspielen kann. Die Klangdaten werden komprimiert – wiederkehrende Muster in den Daten werden zusammengefasst gespeichert und mit einem komplizierten Verfahren werden die Klangdaten intern genauso weit vereinfacht, dass das menschliche Ohr den Unterschied praktisch nicht wahrnimmt. Dieses Format wurde unter dem Namen *MP3* bekannt. Heute ist das längst der Standard für Audiodateien. Der Computer oder das Gerät, das diese Dateien abspielt, muss sie intern erst einmal wieder dekomprimieren, das heißt, aus einer MP3-Datei wird intern wieder eine WAV-Datei gemacht, die dann (über die Soundkarte) an den Lautsprecher gesendet wird.

Egal, ob du WAV-Dateien oder MP3-Dateien hast, kannst du beide mit Python abspielen. Du benötigst dafür lediglich ein Modul namens `pygame-ce`. Wie der Name vermuten lässt, kann dieses Modul mehr, als nur Sounddateien abzuspielen, aber dazu später mehr!

Wie du das Modul mithilfe von Thonny installierst und verwendest, zeige ich dir jetzt.

In Sekunden zum richtigen Modul

Python stellt dir eine ganze Menge an nützlichen Modulen zur Verfügung, die du einfach mit `import` einbinden und verwenden kannst. Aber es gibt noch viel mehr Module für alle möglichen (und unmöglichen) Verwendungszwecke. Da gibt es `pandas` für Tabellen und Datenanalysen, `numpy` für Berechnungen und wissenschaftliches Rechnen. Da ist eben auch `pygame-ce` für Sound, Spiele und Animationen oder `beautifulsoup`, um Webseiten auslesen zu können. Diese ganzen Module für Python werden zentral über `pip` bereitgestellt. `pip`

ist das zentrale Paket-Installationsprogramm von Python. Es greift auf die Daten zurück, die von *PyPI* zur Verfügung gestellt werden. Das ist so etwas wie ein zentraler App-Store für Python-Module. Die Anzahl der bereitgestellten Module liegt inzwischen bei über einer halben Million. Betrieben und verwaltet wird das Ganze von der *Python Software Foundation*.

Im Normalfall werden neue Module über die Kommandozeile deines Systems installiert:

- Unter Windows: `pip install paketname`
- Mit MacOS oder Linux: `pip3 install paketname`

Mit Thonny ist das viel einfacher. Über die Menüleiste **Werkzeuge • Verwalte Pakete ...** kannst du ganz einfach nach Modulen suchen und sie installieren. Das kannst du gleich mit »pygame-ce« ausprobieren. Gib den Begriff oben im Suchfeld ein und klick auf **Suche im PyPI**. Nach einem kurzen Moment werden dir passende und auch manchmal nicht ganz so passende Module vorgeschlagen.

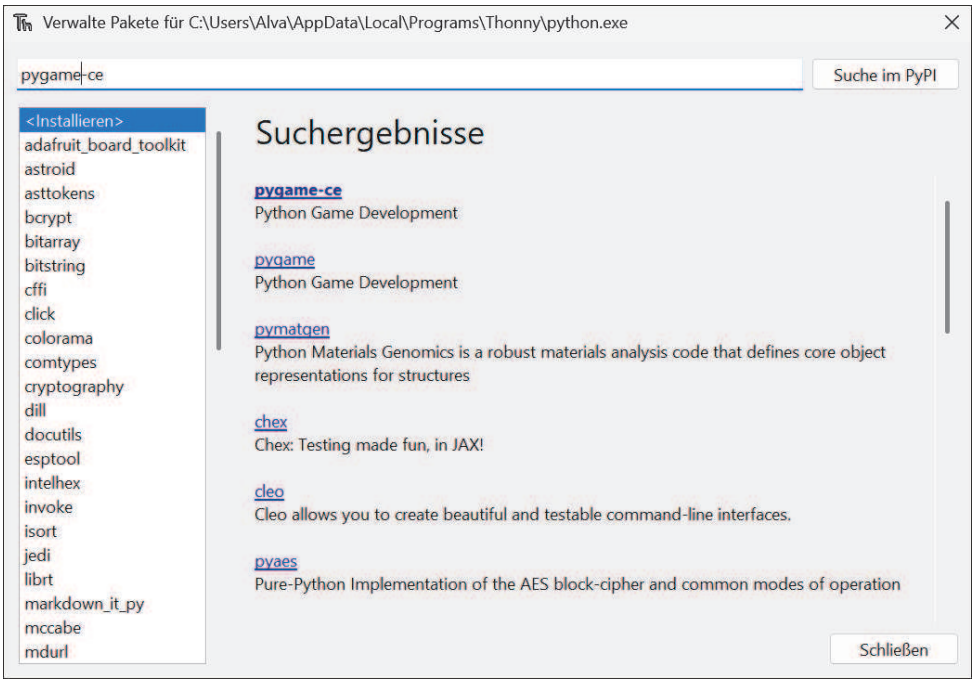


Abbildung 14.1: »pygame-ce« wird dir ganz oben als Suchergebnis angezeigt.

Klick jetzt bitte auf den blau unterstrichenen Namen **pygame-ce**. Auf der folgenden Seite bekommst du kurze Informationen zu dem Programm. Klick jetzt bitte auf **Installieren**.

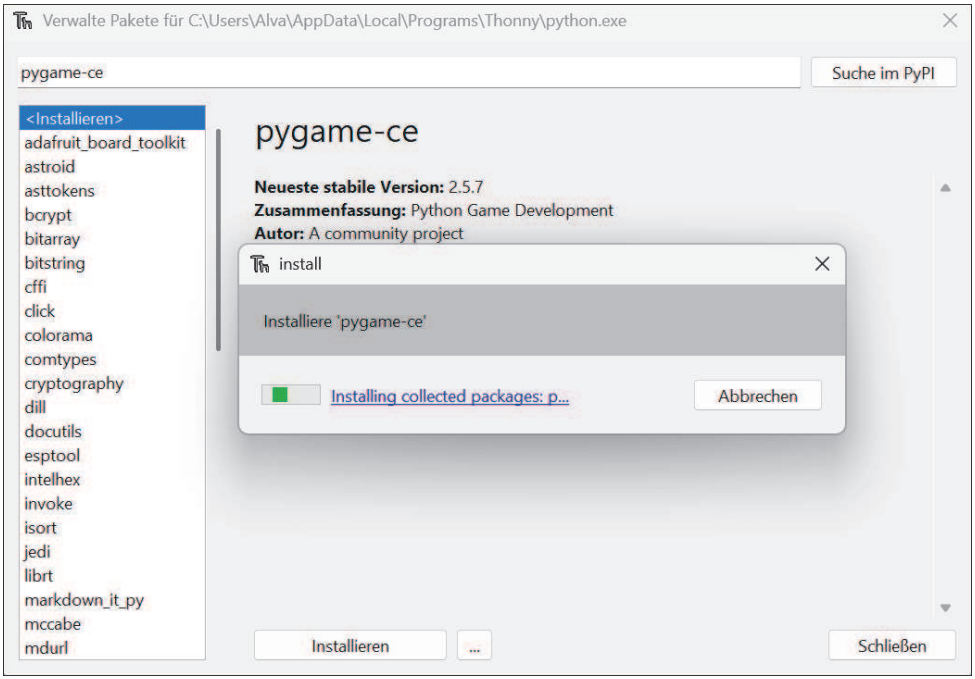


Abbildung 14.2: Die Installation ist schnell gemacht …

Das war es schon. Jetzt kannst du `pygame-ce` auch schon verwenden wie jedes andere Modul, mit dem du bereits gearbeitet hast.

`pygame-ce` ist übrigens eine Abspaltung von dem bekannten Modul `pygame`. Der ursprüngliche Entwickler von `pygame` will mit `pygame-ce` einen offeneren und aktiveren Zweig bereitstellen. Noch sind beide Versionen fast identisch, die Waagschale neigt sich aber doch mehr in Richtung `pygame-ce`.

WAV-Dateien abspielen

Um Klangdateien in einem Programm verwenden zu können, musst du immer als Erstes das Modul `pygame-ce` einbinden. Das geschieht am einfachsten über den folgenden Befehl:

```
import pygame
```

Da `pygame-ce` so etwas wie ein Zweig von `pygame` ist, lautet der Importbefehl tatsächlich nur `pygame`. Das Abspielen ist ganz einfach. Als Erstes wird der Soundplayer aktiviert und es wird ihm eine Datei gegeben, die er zum Abspielen bereit macht. Danach wird der Player initialisiert, das heißt sehr bildlich gesprochen, du schaltest schon mal die Boxen ein.

```
pygame.mixer.init()
```

Jetzt musst du dem Mixer nur noch mitteilen, welche Datei er abspielen soll. Die weist du einer Variablen zu und startest mit `sound.play()` die Wiedergabe:

```
sound = pygame.mixer.Sound("cosmic.wav")
sound.play()
```

Das war's, damit kannst du eine WAV-Datei namens *cosmic.wav* abspielen. Wichtig ist nur, dass sich dein abgespeichertes Python-Programm und die Sounddatei im gleichen Ordner befinden.

Du hast gerade keine WAV-Datei zur Hand? Zum Glück gibt es zahlreiche Anbieter wie Pixabay, die kostenlose Sounddateien zum Herunterladen anbieten. Mit denen kannst du dann ganz einfach testen.

Thonny sorgt nach dem Start dafür, dass die Sounddatei bis zum Ende gespielt wird. Damit das Programm aber auch ohne Thonny ganz bis zum Ende läuft, kannst du das Programm durch diese Schleife aktiv halten:

```
while pygame.mixer.get_busy():
    pass
```

Solange der Mixer noch beschäftigt ist, wird weiterhin diese Schleife durchlaufen und der Sound kann zu Ende gespielt werden. Die Anweisung `pass` ist dabei eigentlich nur ein Platzhalter, denn jede Schleife braucht zwingend mindestens eine untergeordnete Anweisung. `pass` ist ideal dafür, denn es ist eine gültige Anweisung und macht ansonsten sprichwörtlich gar nichts.

MP3-Dateien abspielen

Ich nehme an, du hast Musikdateien (sofern du welche hast) vielleicht eher im Format MP3 bei dir vorliegen? Das Abspielen von MP3-Dateien funktioniert prinzipiell genauso wie das Abspielen von WAV-Dateien. Nur muss das Programm natürlich wissen, dass es eine Datei im MP3-Format erhält und diese erst einmal decodieren muss. Deshalb werden MP3-Dateien etwas anders geöffnet.

Der Import und die Initialisierung unterscheiden sich dagegen nicht:

```
import pygame

pygame.mixer.init()
```

Danach wird die MP3-Datei geladen und mit `play()` abgespielt:

```
pygame.mixer.music.load("club-techno.mp3")
pygame.mixer.music.play()

while pygame.mixer.music.get_busy():
    pass
```

Du brauchst dafür nur eine MP3-Datei, zum Beispiel deinen Lieblingssong oder eine beliebige Datei, wie du sie massenweise legal aus dem Internet herunterladen kannst.

Du kannst dir in deinem Python-Ordner auch ein Verzeichnis namens *mp3* anlegen und deine Dateien dorthin kopieren. Nun kannst du sie mit Python abspielen unter der Pfadangabe:

```
pygame.mixer.music.load("mp3/club-techno.mp3")
```

Denk daran, du musst dein Programm zuvor erst einmal im Python-Ordner speichern, damit Python den MP3-Ordner findet.

Eigene Musik machen

Du kannst mit `pygame` bzw. `pygame-ce` sogar ganz eigene Musik machen. Gut, vielleicht keine Musik, aber mit unserem Programm kannst du zumindest drei unterschiedliche Töne generieren, indem du die Tasten 1, 2 oder 3 drückst. Natürlich kannst du das Programm auch auf acht oder mehr Töne erweitern.

Zuerst importieren wir die benötigten Module: `pygame` und `math` kennst du ja schon.

```
import pygame, math, array
```

Nun folgt die Initialisierung von `pygame`. Das sieht wilder aus, als es ist. Für den `mixer` werden die Abtastrate, also die Qualität, die Bittiefe des Tons und die Kanäle festgelegt – hier mal kein Stereo, Mono reicht.

```
pygame.init()
pygame.mixer.init(frequency=44100, size=-16, channels=1)
```

Die folgende Funktion baut aus einer Frequenz (zum Beispiel 440 Hz) einen kurzen Sinuston. Da wir uns aus Zeit- und Platzgründen nicht intensiver mit der Technik der Tonerzeugung auseinandersetzen wollen, halten wir ganz pragmatisch fest: Hier macht `pygame` aus einer angegebenen Frequenz einen Ton.

```
def ein_ton(freq):
    buf = array.array("h")
    for i in range(9000):
        t = i / 44100
        buf.append(int(32767 * math.sin(2 * math.pi * freq * t)))
    return pygame.mixer.Sound(buffer=buf)
```

Jetzt belegen wir drei Tasten 1, 2 und 3 mit unterschiedlichen Tönen. Das kannst du beliebig verändern und erweitern.

```
tones = {
    pygame.K_1: ein_ton(440),
    pygame.K_2: ein_ton(490),
    pygame.K_3: ein_ton(525),
}
```

Hier öffnen wir ein kleines Fenster, damit das Programm sichtbar läuft:

```
screen = pygame.display.set_mode((400, 200))
pygame.display.set_caption("1,2,3 Töne")
```

In dieser Schleife wartet das Programm auf Tastendrücke und erzeugt die gewünschten Töne:

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.KEYDOWN:
            if event.key in tones:
                tones[event.key].play()
```

Sprachsynthese – lass den Computer sprechen!

Jetzt wird es richtig cool! Mit Python und einer Bibliothek namens `pyttsx3` kannst du den Computer auch richtig sprechen lassen, und zwar jeden Text, den du möchtest!

Zuerst musst du das Modul `pyttsx3` installieren. Wie das geht, weißt du ja: Über die Menüleiste **Werkzeuge • Verwalte Pakete …** kannst du ganz einfach nach Modulen suchen und sie installieren. Ist das Modul installiert, brauchst du nur ein paar Zeilen Code für eine Sprachausgabe.

Probiere mal das hier aus:

```
import pyttsx3

engine = pyttsx3.init()
```

```
engine.say("Hallo, hier spricht Python.")
engine.runAndWait()
```

Einfacher geht es kaum. Willst du mehrfach Texte ausgeben, solltest du dir eine Funktion dafür schreiben:

```
import pyttsx3

def sprich(text):
    engine = pyttsx3.init()
    engine.say(text)
    engine.runAndWait()

sprich("Hallo")
sprich("Ich habe viel zu sagen")
```

Einmaleins

Wie wäre es mit einem sprechenden Einmaleins? Der Computer nennt dir zwei Zahlen als Aufgabe, wartet dann 3 Sekunden, bevor er das Ergebnis nennt. Zwar gibt es keine Rückmeldung oder Kontrolle, was du gerechnet hast, aber mit Freunden und auch älteren Bekannten ist es einfach eine »Gaudi«, gemeinsam als Erster das Ergebnis in den Raum zu rufen.

Drei Module brauchen wir. Neu ist `time`, mit dem wir 3 Sekunden warten können.

```
import random
import time
import pyttsx3
```

Nun kommt wieder unsere sprechende Funktion:

```
def sprich(text):
    engine = pyttsx3.init()
    engine.say(text)
    engine.runAndWait()
```

Das Spiel läuft noch in einer Endlosschleife. Du könntest es als kleine Aufgabe nach dem 20. Durchlauf stoppen.

```
while True:
    #Zufallszahlen erzeugen
    a = random.randint(1, 10)
    b = random.randint(1, 10)
```

```
    #Aufgabe sagen
    sprich(f"{a} mal {b}")
```

Hier wartet das Programm mithilfe des Moduls `time` 3 Sekunden:

```
    time.sleep(3)

    #Ergebnis sprechen
    sprich(f"Das Ergebnis ist {a * b}")
```

Und jetzt? Na, viel Spaß beim gemeinsamen Rechnen und beim Verbessern des Programms! Vielleicht wandelst du es auch ab, sodass man Zahlen oder Worte vorgelesen bekommt und nach einer kurzen Wartezeit alles Gemerkte aufschreiben muss? Du hast bestimmt ein paar spannende Ideen …

Inhalt

Index

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

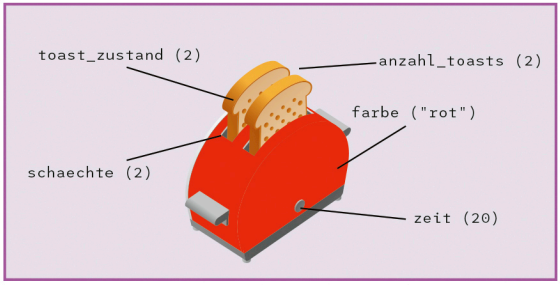
Z

<LET'S CODE> Python

Python macht Spaß! Überall in unserer digitalen Welt steckt Python: vom Raspberry Pi bis zur künstlichen Intelligenz. Steige jetzt ohne Vorkenntnisse ein, Schritt für Schritt, mit verschiedenen kleinen Programmen und eigenen Spielen. Mit der einsteigerfreundlichen Entwicklungsumgebung Thonny legst du sofort mit Python 3 los. Du brauchst nur noch deinen Computer, und los geht's!

- Installation und erste Schritte
- Dem Computer Befehle geben
- Quiz, Würfel-Poker und Co.
- Spiele-Klassiker nachbauen
- Grafik und Sound
- Geschwindigkeit, Schwerkraft und Logik programmieren
- Datenbanken und Tabellen mit SQL

Mit allen Codebeispielen zum Download



Hauke Fehr entwickelt seit vielen Jahren Software und weiß als Autor und Dozent, worauf es beim Erklären ankommt. **Stephan Elter** ist Softwareentwickler und programmiert seit vielen Jahren mit Python. Er liebt witzige Beispiele und Fachbücher, die sich selbst nicht zu ernst nehmen.

Rheinwerk Computing

Gedruckt in Deutschland
Mineralölfreie Druckfarben
Zertifiziertes Papier

€ 29,90 [D] € 30,80 [A]
Für Windows, macOS und Linux
Programmierung
ISBN 978-3-367-11517-4