c't Python-Projekte

Von alltagstauglich bis völlig nerdig

www.ctspecial.de

Neuronale Netze

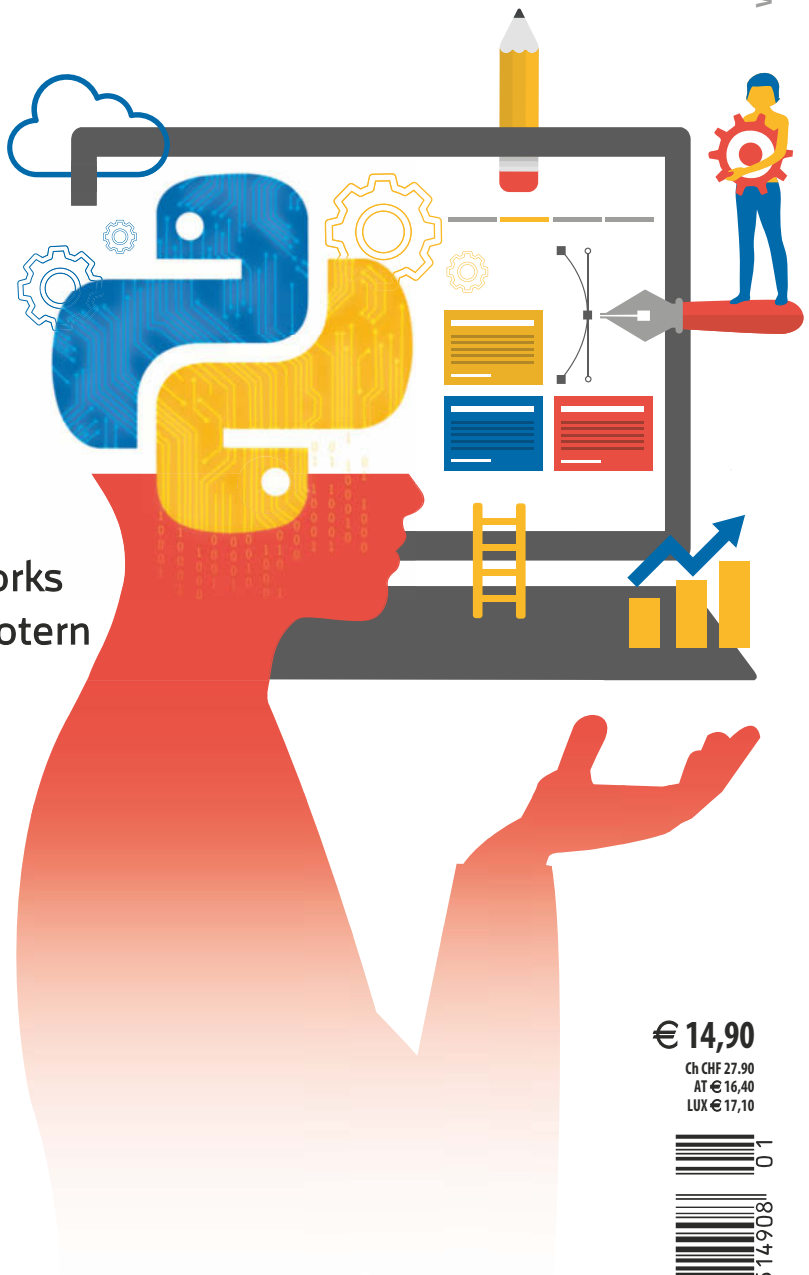
Bilder scharf hochskalieren
Schlechte Aufnahmen löschen
Van Gogh & Co. für Fotos
Textsorten unterscheiden

Automatische Tests

Systematisch Coden mit Test-Frameworks
Automatisch testen mit Software-Robotern
Selbstoptimierende Fehlersuche

Praktische Helfer

Schrifterkennung mit Tesseract
Daten verstecken in Google Fit
Bilder sortieren mit EXIF-Daten



€ 14,90
Ch CHF 27.90
AT € 16,40
LUX € 17,10

4 198661 514908 01

Profi-Python

Ruckelfreie GUI mit PyQt 5 · Passwortsuche in 25 GByte
Megatabellen mit Pandas · Schöne Diagramme mit Altair

Editorial

Liebe Leserinnen und Leser,

Sie halten kein Lehrbuch in Händen. Stattdessen präsentiert dieses Sonderheft Projekte, die jeweils für sich ein Problem vorstellen und es mit Python lösen. Trotzdem können Sie Ihre Python-Skills mit diesem Heft vom blutigen Anfänger bis zur Fachgröße steigern. Unsere Sammlung enthält nämlich vom Computerspiel „Knäueljagd“ (S. 28) für programmierende Kinder über schnelle Projekte wie ein Bilder-Sortier-Skript (S. 60) bis zum Details erfindenden tiefen Convolutional Network in der KI (S. 84) alle Schwierigkeitsgrade.

Statt einem Semester mit Vorlesungen haben Sie eine Zusammenstellung mit Beispielen ohne didaktisches Konzept. Mit denen können Sie einfach loslegen – ohne Studienordnung, ohne ein halbes Jahr Vorlauf, ohne Mathe-Vorkurs. Passen Sie unsere Projekte dabei gern an Ihre eigenen Bedürfnisse an: So lernen Sie nämlich am meisten. Das beste dabei ist aber, dass Sie an Ihren und unseren Projekten nicht nur lernen. Jedes Projekt löst auch ein konkretes Problem.

Ein bisschen haben wir aber dennoch für eine gute Lernkurve gesorgt: Nach dem Einstieg am Anfang des Heftes finden Sie bei den Alltagshelfern ab Seite 40 die richtigen Herausforderungen. Für ambitionierte Hobbyisten bieten unsere nerdigen Hardcore-Projekte ab Seite 154 gutes Futter und unsere Reihe zum automatischen Testen ab Seite 120 ist genau richtig für Berufsprogrammierer, denen Code-Qualität und Performance am Herzen liegen. Und mit unseren KI-Projekten ab Seite 62 werden Sie zum Data-Scientist.

Stürzen Sie sich also einfach in die Python-Projekte, die Ihnen am meisten Spaß machen! Lernen werden Sie dabei ganz automatisch.

Pina Merkert

Inhalt

PYTHON LERNEN

Mit einer klaren und übersichtlichen Syntax eignet sich Python hervorragend als Einstiegssprache. Wir erklären die ersten Schritte mit dem sicheren und alltagstauglichen Passwort-Manager c't SESAM.

PYTHON ALS ALLTAGSHELFER

Bereits mit wenigen Zeilen Code programmieren Sie nützliche Helfer. Mit diesen Projekten lernen Sie Frameworks kennen, die komplizierte Probleme ganz einfach lösen.

KI MIT NEURONALEN NETZEN

Ein Blick hinter den Hype ums maschinelle Lernen: Mit unseren KI-Projekten klappt der Einstieg in die Entwicklung künstlicher Intelligenzen. Die Projekte zeigen an konkret nutzbaren Beispielen, wie man KIs erfolgreich trainiert.

AUTOMATISCHES TESTEN

Automatische Tests finden Fehler im Code und sparen Zeit. Der zusätzliche Programmieraufwand lohnt sich schon nach wenigen Releases.

PYTHON FÜR PROFIS

Python vereinfacht viele aufwendige und komplexe Berechnungen. Mit den richtigen Tools gelingt alles von Wissenschaft bis Web-Anwendung.

ZUM HEFT

Elegante Entschleunigung

Python-Code ist kurz und verständlich. Mit dem Interpreter gehen Experimente schnell, während die Programme verglichen mit C langsam laufen. KI-Forscher entwerfen damit trotzdem die schnellsten neuronalen Netze.

Von Pina Merkert

Python begeistert Entwickler, die verständlichen Code schreiben möchten. Gut für die Übersicht: Einrückungen statt Klammern definieren Blöcke; in jeder Zeile steht nur ein Befehl ohne Semikolon. Besonders kompakter und lesbarer Code gilt als „pythonic“ und erntet das Lob der Community. Gleichzeitig kann man mit Python alles programmieren: Vom Bash-Skript-Ersatz über grafische Desktop-Programme bis zum neuronalen Netz – Python kann alles!

Das Programm `python` interpretiert den Code direkt und überspringt daher das Kompilieren in Maschinencode. Deshalb eignet sich Python besonders für Programmier-Experimente. Python-Code ist schnell geschrieben und schnell ausprobiert. `python` ohne Argument startet beispielsweise eine interaktive Konsole, die jede getippte Zeile sofort ausführt – toll zum Testen, ob eine Anweisung wie gedacht funktioniert. Noch bequemer geht das mit Jupyter-Notebooks (Python im Browser), die nebenbei auch Text und Grafik anzeigen und schon manche GUI überflüssig gemacht haben.

Der Preis für den Interpreter ist Rechenzeit: Der gleiche Algorithmus braucht in Python bis zu 100-mal so lang wie in C. Python selbst und die meisten Bibliotheken sind daher in C geschrieben und stellen nur ein „pythonic“ Interface bereit. Im Alltag merkt man das daran, dass Pythons Paketmanager `pip` öfter mal den C-Compiler des Systems anwirft – und scheitert, wenn es den nicht findet.

Angesichts von Pythons Langsamkeit überrascht es, dass Python die Sprache aller KI-Forscher ist. Die nutzen jedoch Frameworks wie Caffe oder TensorFlow, die ihre Berechnungen automatisch für die vorhandene Hardware optimieren und dabei beispielsweise den CUDA-Compiler anwerfen, um auf der Grafikkarte zu rechnen. Für die Auswertung der Ergebnisse nutzen die Wissenschaftler hoch optimierte Frameworks wie Numpy und Pandas. Dass ein paar Zeilen Python die aufwendigen Berechnungen anstoßen und verwalten, spielt für die Laufzeit letztlich keine Rolle. Die Bequemlichkeit beim Experimentieren bleibt jedoch.

Generell gibt es für fast jede C-Bibliothek einen Python-Wrapper. Dank derer sieht es so aus, als wäre jedes Kunststück, zu dem der Rechner fähig ist, auch Teil von Python. Damit steht man auf den Schultern der vielen Entwickler, die ihre Lösungen längst in Bibliotheken verpackt haben. Die passende Doku lagert meist bei readthedocs.org.

Für wen?

Python empfiehlt sich für Anfänger, die noch nie programmiert haben. Die Sprache erlaubt jede Art von Programm und die eigenen Kenntnisse wachsen mit jedem neuen Projekt. Auch für funktionale oder probabilistische Programmierung muss man die Python-Welt nicht verlassen.

Maschinelles Lernen und vor allem neuronale Netze sollte man gar nicht erst versuchen in einer anderen Sprache zu entwerfen: Das Python-API ist bei TensorFlow & Co. stets die vollständigste und verbreitetste Schnittstelle.

Es gibt Frameworks für alles: Mit Django gelingen große Webapplikationen, mit Flask schreibt man schnell HTTP-Microservices in Python. Das passende Web-Frontend sollte man allerdings mit ei-

Ungewohnt: Python nutzt Einrückungen statt Klammern, um im Code Blöcke zu bilden. Programme sind dadurch gut lesbar.

```
# -*- coding: utf-8 -*-
import os

print("Diese Anweisung wird ausgeführt, sobald Python die Datei importiert.")

def funktion_mit_parameter(x):
  print("Der Typ einer Variable kann sich ändern:", type(x), x)
  x = 2
  return "Die str()-Funktion wandelt ein " + str(type(x)) + "in einen String uma: " + str(x)

if __name__ == "__main__":
  print("Diese führt Python nur aus, wenn die Datei direkt aufgerufen wird.")
  print(funktion_mit_parameter({"Geschweifte Klammern": "definieren dictionaries.",
                "float": 2.3, "liste": [1, "zwei", int]}))
  typ_der_klammer, wort, _ =(tuple, 'Einrückungen', "definieren Blöcke im Code:")
  for i in range(4):
    print(wort, " " * i + "Zeile", i)
  wortliste = 'String-Funktionen~erleichtern~die~Arbeit~mit~Zeichenketten.'.split("~")
    # Kommentar: List-Comprehensions (for-Schleife definiert neue Liste) sparen Platz
  print(" ".join(["*" + wort + "*" for wort in wortliste]))
  print("Module nehmen viel Arbeit ab. Pfad:", os.path.abspath(os.path.curdir))
```

Python.org und Anaconda

ct.de/wf31

nem JavaScript-Framework wie React schreiben, da Browser kein Python verstehen. Auch Apps für iOS und Android entstehen besser in Swift oder Kotlin. Gegen Desktop-Anwendungen in Python spricht dagegen wieder nichts.

Python ist ein Schweizer Messer, mit einem behäbigen Interpreter, was man meist mit Bibliotheken ausgleichen kann. Die Community feiert Ästhetik und Verständlichkeit, was Anfänger wie Profis gleichermaßen beglückt. (pmk) ct

Programmieren lernen mit c't SESAM

Programmierkenntnisse eröffnen am Computer neue Möglichkeiten. Damit der Rechner folgt, muss man aber erst mal dessen Sprache lernen. Python macht den Einstieg leicht: Wenige Zeilen Code reichen, um einen sicheren und alltagstauglichen Passwort-Manager auf die Beine zu stellen.

Von Pina Merkert

Für den Bau einer Kathedrale bedarf es Hunderter hoch spezialisierter Baumeister und viel Zeit. Bei komplexer Software ist es ähnlich: Niemand programmiert mal eben so eine Photoshop-Alternative oder einen Browser. Aber man muss weder Baumeister noch Handwerker sein, um in der eigenen Wohnung einen Dübel zu setzen.

Ganz ähnlich verhält es sich beim Programmieren. Gut 20 Zeilen Python reichen zum Schreiben eines Passwort-Managers, der mehr als ein Spielzeug ist. Für den schnellen Einstieg ins Programmieren erklären wir anhand dieses Beispiels die wichtigsten Konzepte.

Ein Programm entwerfen

Ein Programm ist eine Anleitung, die der Computer von oben bis unten abarbeitet. Bevor Sie die erste Zeile Code schreiben, sollten Sie eine klare Vorstellung haben, was der Computer machen soll. Programmieren fordert Sie heraus, große Probleme so klein aufzuteilen, dass der Rechner die Teilprobleme lösen kann. Mit zunehmender Erfahrung erarbeiten Sie immer mehr Lösungen für solche Teilprobleme.

Jedes gelöste Teilproblem kann im nächsten Programm wieder eingesetzt werden, um ein komplexeres Problem zu lösen. Wir empfehlen Ihnen, mit einfachen Beispielen anzufangen und sich erst nach und nach an komplexere Probleme zu wagen. Wenn Sie mit Python gelernt haben, große Probleme in lösbare Teilprobleme aufzubrechen, können Sie diese Erfahrung bei allen anderen Programmiersprachen und auch abseits vom Computer nutzen.

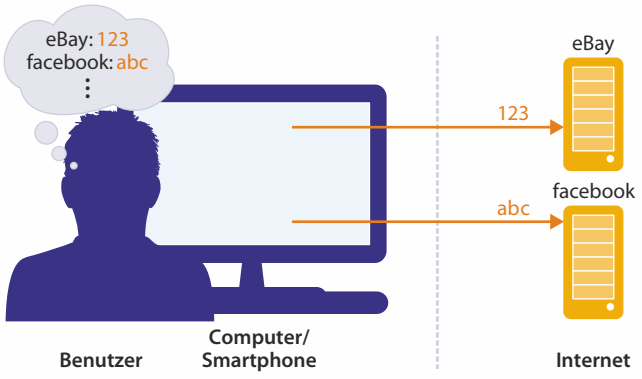
Für die Planung eines Programms empfehlen sich je nach Programm unterschiedliche Vorarbeiten. Für grafische Programme skizzieren Sie am besten zuerst die gewünschte Oberfläche. Für Datenbankanwendungen sollten Sie sich vorab überlegen, welche Daten das Programm speichert und unter welchen Bedingungen es darauf zugreift. Ein einfacher Passwort-Manager auf der Konsole muss nur die Eingabe und Verarbeitung der Daten in der richtigen Reihenfolge gewährleisten.

Passwort-Meister

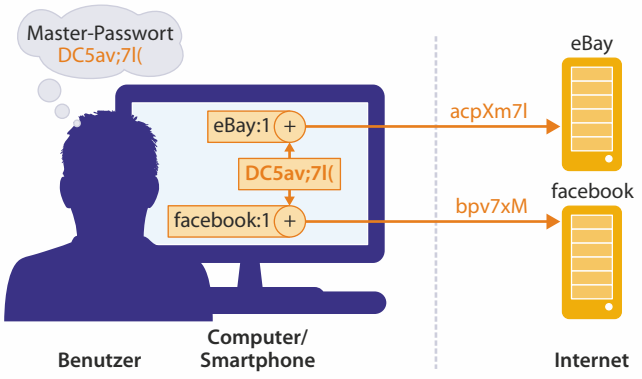
Sichere Passwörter sind ein Problem: Sie sollen länger als 10 Zeichen sein, dürfen in keinem Wörter-

Passwort-Konzepte

Normal: Der Anwender muss sich viele Passwörter merken und schickt sie dann übers Netz an die Dienste, um sich auszuweisen.



Master-Passwort: Eine kleine App erzeugt für jeden Dienst ein eigenes, unknackbares Passwort. Der Anwender muss sich nur noch sein Master-Passwort merken.



buch stehen, sollen Sonderzeichen und Zahlen enthalten. Ein solches Passwort zu lernen kostet Mühe. Wird es selten gebraucht, vergisst man es schnell. Zu allem Überfluss soll man für jeden Dienst und auf jeder Domain ein eigenes Passwort verwenden.

Mit einem Passwort-Generator wie c't SESAM lernen Sie nur ein einzelnes richtig langes und kompliziertes Masterpasswort auswendig. Das Programm berechnet aus dem Masterpasswort und dem Namen einer Webseite ein Passwort, um sich dort anzumelden. Jeder Dienst bekommt so sein eigenes Passwort. Alle Passwörter sind ausreichend lang und enthalten genug Sonderzeichen, sodass Angreifer sie nicht erraten.

Sollte eine Webseite gehackt werden oder ein Passwort in die Hände eines Angreifers fallen, kann er nicht vom Dienstpasswort auf das Masterpasswort schließen. Die Password-Based-Key-Derivation-Function-2 (PBKDF2) sorgt dafür, dass ein Angreifer nur Masterpasswörter ausprobieren kann, und jeder Versuch so lange dauert, dass er selbst mit einem Supercomputer nicht auf das Masterpasswort kommt.

Der für die Sicherheit verantwortliche Hash-Algorithmus (PBKDF2) ist in Python integriert. Dadurch eignet sich der nützliche Passwort-Manager als Einsteigerbeispiel. Kryptografische Algorithmen sollten Sie ohnehin nie selbst implementieren, da es dort besonders viele Fallstricke und mögliche Fehler gibt. Wir haben das Beispielprogramm c't SESAM genannt, was für „Sehr einfaches, sicheres Authentifizierungs-Management" oder englisch „super easy secure authentication management" steht.

c't SESAM soll Anwender nach einem Masterpasswort und einer Domain fragen. Ohne Masterpasswort und Domain kann das Programm nichts berechnen, also muss es diese Abfrage auf jeden Fall als Erstes ausführen. Anschließend verarbeitet PBKDF2 diese Daten so, dass ein Angreifer, der eines der generierten Passwörter erfährt, keine Möglichkeit hat, das Masterpasswort zu berechnen.

Danach muss c't SESAM das Ergebnis, das dieser Algorithmus berechnet, noch in ein Passwort der richtigen Länge verwandeln. Für dieses Teilproblem wählt das Programm einzelne, für ein Passwort geeignete Zeichen aus und hängt sie aneinander. Welche Zeichen das Programm auswählt, soll vom Ergebnis von PBKDF2 abhängen. Hat c't SESAM ein Passwort der gewünschten Länge zusammengebaut, kann es das Passwort ausgeben.

Python einrichten

Unter Linux gehört Python zur Standardausstattung. Achten Sie auf die Version: Die Anweisung python ruft oft noch `Python` in Version 2 auf. Der Befehl `python3` startet auf allen Distributionen die aktuelle Version der Sprache. Der hier vorgestellte Programmtext benötigt mindestens Python 3.4; aktuell ist 3.8.

Unter Windows müssen Sie Python erst installieren. Den Installer können Sie von python.org herunterladen. Auch hier sollten Sie die aktuelle Version von Python 3 installieren.

Wenn Sie kein Python installieren möchten, können Sie Jupyter Notebook im Browser verwenden. Dies ist ein Webdienst, mit dem man Python im Browser editieren und ausführen kann. Wenn Sie Mathematik-Programme kennen, werden Sie sich bei Jupyter Notebook sofort zu Hause fühlen. Jedes Notebook ist ein Dokument, das auf dem Server gespeichert wird. Es besteht aus Programmtext, den Ergebnissen der Ausführung und Text. Mit Maus oder Tastatur ausgewählte Abschnitte führen Sie mit dem Play-Symbol oben in der Menüleiste aus.

Im c't-Link finden Sie einen Dienst, der Ihnen virtuelle IPython-Notebook-Server zur Verfügung stellt. Echte Passwörter sollten über einen solchen Webdienst natürlich nicht generiert werden. Außerdem existieren die Testserver nur temporär, sodass die erzeugten Notebooks relativ schnell wieder gelöscht werden. Leider passiert das schon nach einigen Minuten Inaktivität, selbst wenn Sie das Browser-Fenster nicht schließen.

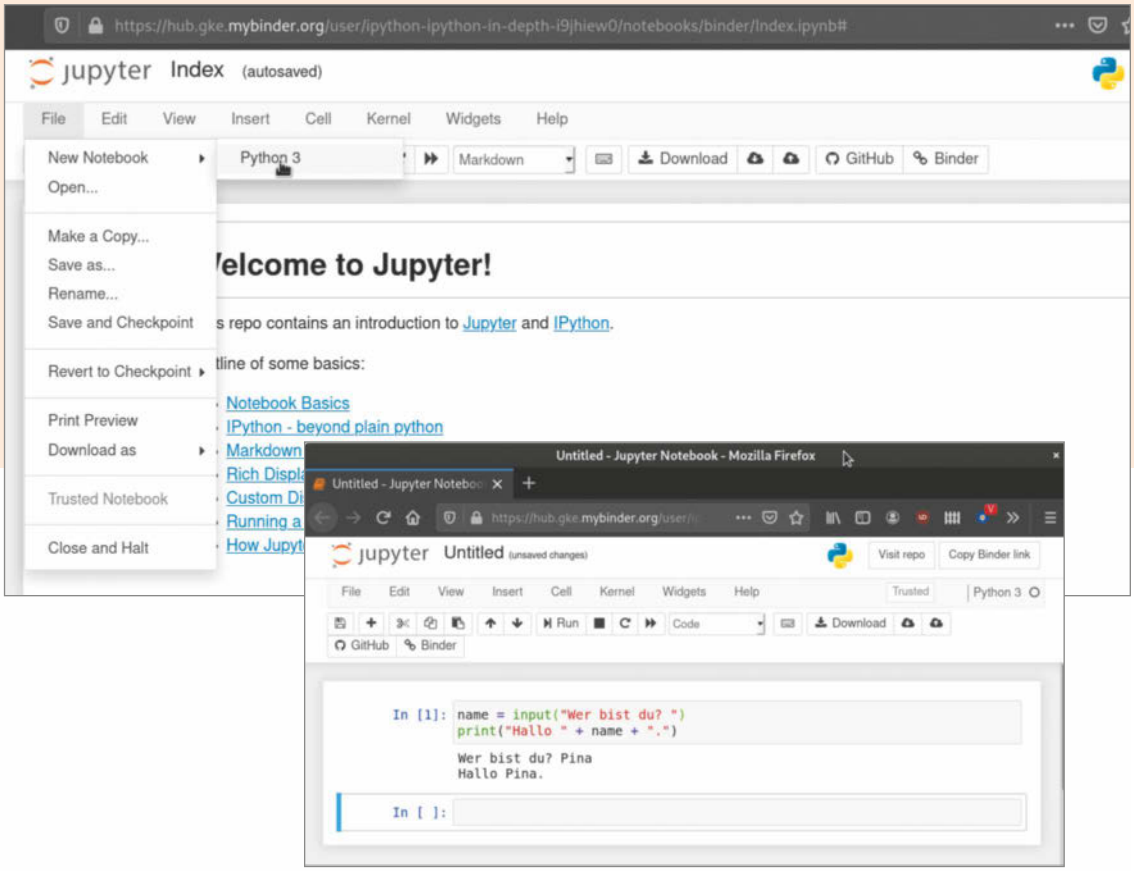
Programme speichern und ausführen

Ein Python-Programm ist eine Textdatei mit der Endung .py. Unter Linux führen Sie das Beispiel auf der Konsole mit `python3 ctSESAM.py` aus.

Unter Windows empfehlen wir eine Entwicklungsumgebung, aus der Sie das Programm starten. IDLE (Integrated DeveLopment Environment) ist bei Python unter Windows dabei und reicht für c't SESAM aus. Die IDE startet in einer interaktiven Konsole, in der Sie Python-Befehle testen können. Ausgaben Ihres Programms landen in diesem Fenster. Im Menü von IDLE legen Sie mit „File", „New File" eine neue Datei an und führen den Programmtext mit „Run", „Run Module" aus.

Damit das Python-Programm den Benutzer zu einer Eingabe auffordert, reicht der Befehl

Über das Web-basierte Jupyter Notebook können Sie die Programmiersprache Python ohne Installation ausprobieren.



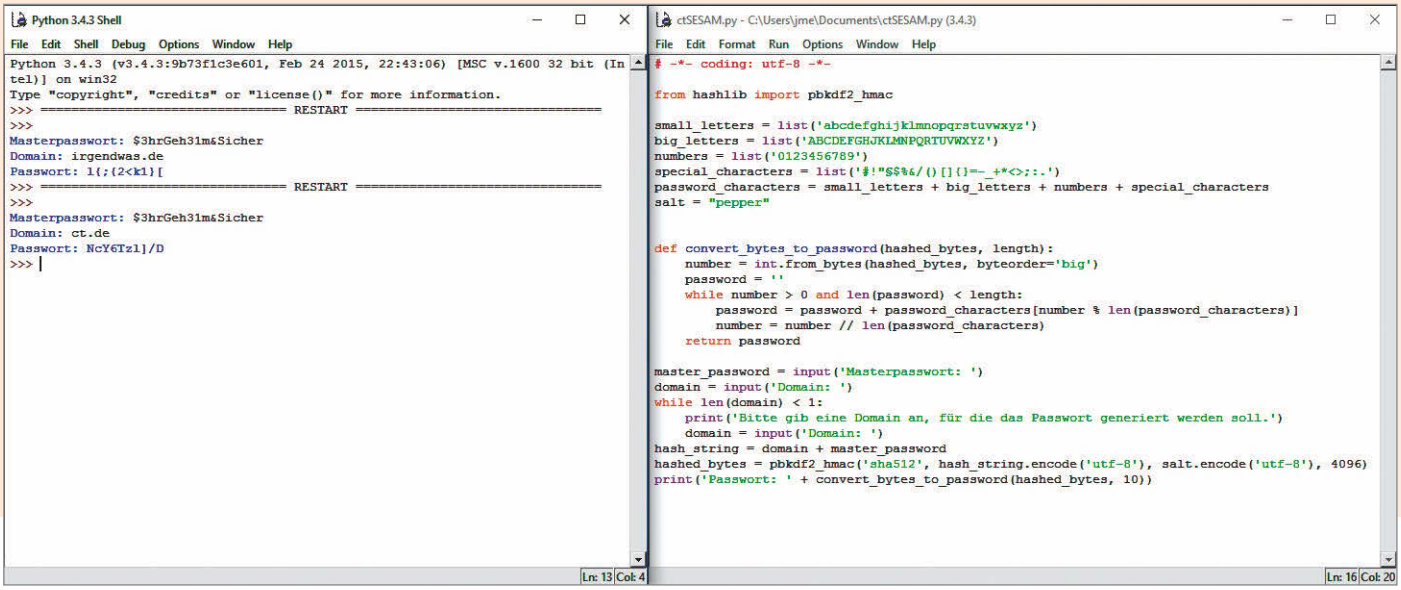
`input("Masterpasswort: ")`. An den runden Klammern nach dem Namen erkennen Sie, dass `input()` eine Funktion ist. Funktionen sind Teile von Programmen, die eine bestimmte Aufgabe erfüllen. `input()` ist eine in Python integrierte Funktion, die Tastatureingaben entgegennimmt, bis die Eingabe-Taste gedrückt wird. Wie Sie eigene Funktionen definieren, lernen Sie etwas später bei der Konstruktion des Passworts. Funktionen wollen mit Parametern gefüttert werden:

```
funktion(parameter1, parameter2)
```

Der Parameter `"Masterpasswort: "` ist ein String, also eine Zeichenkette. Python akzeptiert Strings sowohl mit einfachen als auch mit doppelten Anführungszeichen. Wenn Sie das Programm ausführen, zeigt `input()` den in den Klammern übergebenen String an. So weiß der Benutzer, welche Eingabe das Programm erwartet.

Wenn Funktionen ein Ergebnis produzieren, geben sie es nach dem Aufruf zurück. Die Zeile `wert = f()` speichert die Rückgabe der Funktion `f()` in der Variablen `wert`. Den Datentyp dieser Rückgabe bestimmt die Funktion. Beispielsweise gibt `input()` die eingegebenen Zeichen als String zurück.

Um den String für die spätere Verwendung zu speichern, müssen Sie eine Variable definieren. Variablenzuweisungen sehen aus wie mathematische Gleichungen, weisen den Rechner aber eigentlich nur an, in der Variablen links vom Gleichheitszeichen die Daten rechts vom = zu speichern.



Python für Windows enthält die Entwicklungsumgebung IDLE, die für c't SESAM ausreicht. Im rechten Fenster steht der Editor für das Programm, links die interaktive Konsole für die Ausgaben des Programms und kleine Tests.

Eine Zeile reicht, um die eingegebenen Zeichen des `input()`-Befehls, der das Masterpasswort abfragt, in der Variablen master_password zu speichern:

```
master_password = input("Masterpasswort: ")
```

Bei Python ist es üblich, Variablennamen, die aus mehreren Wörtern bestehen, durch Unterstriche zu verbinden. Falls Sie Wörter lieber `direktAneinander-Reihen` wollen, akzeptiert Python das auch.

Fehlerhafte Eingaben

Die Zuweisung `domain = input("Domain: ")` bittet um die Eingabe der Domain und speichert sie in der Variablen `domain`. Gibt ein Benutzer keine Domain ein und drückt direkt die Eingabe-Taste, kann das Programm nicht weiter laufen, schließlich berechnet c't SESAM seine Passwörter anhand des Domain-Namens. Also sollte das Programm überprüfen, dass der Nutzer mindestens ein Zeichen eingegeben hat, und ihn andernfalls erneut nach der Domain fragen.

Die Überprüfung einer Bedingung wird in Python mit `if Bedingung:` eingeleitet. Was passieren soll, wenn die Bedingung erfüllt ist, steht dann eingerückt in der nächsten Zeile. In Python werden zusammengehörende Programmabschnitte durch Einrückungen markiert. Es dürfen dafür Leerzeichen und Tabulatoren benutzt werden. Wichtig ist nur, dass ein Abschnitt einheitlich eingerückt wird.

Andere Programmiersprachen benutzen für die Markierung von zusammenhängenden Abschnitten meist geschweifte Klammern. Außerdem werden dort Befehle terminiert, häufig mit einem Strichpunkt. Der für Python typische verschwenderische Umgang mit Leerzeichen und Zeilensprüngen erzwingt lesbaren Programmtext, erfordert aber Disziplin beim Setzen von Leerzeichen.

Hinter dem `if` steht die Bedingung. Eine Bedingung ist eine Zeile Code, die entweder wahr (`True`) oder falsch (`False`) ist. Eine Variable vom Typ Boolean speichert genau einen dieser beiden Fälle. Im einfachsten Fall besteht eine Bedingung nur aus einer solchen Variablen. Häufig wird bei Bedingungen ein Vergleich benutzt wie `x < 1:` Wenn die Variable `x` eine Zahl kleiner als 1 ist, gibt der Vergleich True zurück.

Die Funktion `len()` gibt die Länge eines Strings als ganze Zahl zurück. Mit dieser Funktion kann c't SESAM prüfen, ob der Benutzer eine Domain eingegeben hat:

```
domain = input("Domain: ")
if len(domain) < 1:
    print("Bitte gib eine Domain an.")
    domain = input("Domain: ")
```

Die Funktion `print()` gibt ihren Parameter als eigene Zeile aus, ohne Eingaben zu erwarten.

c't SESAM stellt lediglich sicher, dass `domain` mindestens ein Zeichen lang ist, sodass der Benutzer auch etwas anderes als einen Domain-Namen eintippen kann. c't SESAM soll auch Eingaben verarbeiten, die nicht das Format von Internet-Domainnamen haben. Daher ist die Überprüfung bewusst minimalistisch. Wer statt „sparkasse-hannover.de" lieber „Bank" eintippt, kann den Passwort-Manager auch so nutzen.

Wenn ein Benutzer trotz des Hinweises den gleichen Fehler wiederholt, bliebe `domain` leer, obwohl die Überprüfung das eigentlich verhindern sollte. Wenn der Rechner etwas wiederholen soll, solange eine Bedingung gilt, hilft eine `while`-Schleife: Das Programm wiederholt den Programmtext in der Schleife, solange die Bedingung erfüllt bleibt. Um sicherzustellen, dass `domain` nicht leer bleibt, ersetzen Sie das `if` durch `while`. Die vier Zeilen zur Eingabe der Domain sehen dann so aus:

```
domain = input("Domain: ")
while len(domain) < 1:
    print("Bitte gib eine Domain an.")
    domain = input("Domain: ")
```

Daten vorbereiten

Der Algorithmus, den wir nicht selbst programmieren wollen (PBKDF2), erzeugt aus einer Folge von Bytes einen Schlüssel. Dazu muss das Programm die Domain und das Masterpasswort zu einem einzelnen String verbinden. In Python werden Strings mit + aneinandergehängt:

```
hash_string = domain + master_password
```

In der neuen Variablen `hash_string` stehen danach hintereinander die Domain und das Masterpasswort.

Der Aufruf von PBKDF2 erwartet vier Parameter: Den Anfang macht der Name des Hash-Algorithmus, den PBKDF2 intern verwendet. c't SESAM nutzt SHA512. Es folgt die Kombination aus Masterpasswort und Domain. Zusätzlich braucht PBKDF2 ein „Salz": Diese Bytefolge fließt in den ersten Hash-Durchgang des Algorithmus ein. Das Salz können Sie beliebig wählen, allerdings führt eine Änderung des Salzes zu komplett anderen Passwörtern. Wir haben „pepper" verwendet. Im Idealfall würde das Programm für jedes Passwort ein eigenes Salz generieren. Es müsste dieses Salz dann aber speichern und auf allen Geräten, die die richtigen Passwörter erzeugen sollen, synchronisieren. Für den Einstieg führt das zu weit. Auch mit einem festen Salz erzeugt das Programm sichere Passwörter. Die erweiterte Version im nächsten Artikel verwendet für jedes Passwort ein eigenes Salz.

Der letzte Parameter ist die Anzahl an Iterationen. Dieser Parameter bestimmt, wie schnell der Algorithmus läuft. Da Angreifer für jeden Versuch, Ihr Masterpasswort zu erraten, genauso lange brauchen wie Ihr Programm bei der Errechnung des Kennworts, sollte der Algorithmus nicht zu schnell laufen. Wir haben uns für 4096 Iterationen entschieden.

Normalerweise kann man es der Programmiersprache überlassen, wie die Zeichen einer Zeichenkette als Byte-Folgen dargestellt werden. Da PBKDF2 aber bei Masterpasswort plus Domain und beim Salz statt Strings eine Folge von Bytes als Ein-

Debugging-Ausgaben

Auf der Suche nach Fehlern oder zur Überprüfung der Zwischenschritte werden Sie häufig wissen wollen, was in einer Variablen steht. Das erreichen Sie am einfachsten, indem Sie die Variable auf der Konsole anzeigen lassen.

Die `print()`-Funktion gibt Strings genau auf diese Weise aus und konvertiert automatisch andere Datentypen. `print(5 / 2)` schreibt zum Beispiel `2,5` auf die Konsole. Fügen Sie ruhig in Ihren Programmtext zusätzliche Zeilen mit `print()`-Anweisungen ein. Läuft das Programm erst einmal fehlerfrei, lassen sich diese Zeilen schnell wieder entfernen.

gabe erwartet, müssen die Strings erst in Byte-Folgen konvertiert werden. `string.encode()` erledigt die Konvertierung. Damit die Funktion das richtige Encoding benutzt, müssen Sie den Namen des Encodings als Parameter übergeben:

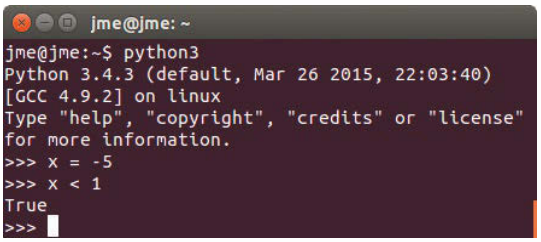
```
hash_string_bytes=hash_string.encode(
                                "utf-8")
salt="pepper"
salt_bytes=salt.encode("utf-8")
```

Das Encoding legt fest, welche Bytes im Speicher zu welchen Zeichen gehören. Gewöhnen Sie sich an, immer UTF-8 zu verwenden. Dieses Encoding umfasst alle Zeichen, was Probleme mit Sonderzeichen und Fremdsprachen vermeidet.

Bibliotheken nutzen

Machen Sie sich nicht selbst Arbeit, greifen Sie auf die Arbeit anderer Entwickler zurück! Python bietet Schnittstellen zu fast allen Bibliotheken. Eine Websuche nach dem Namen der Bibliothek und „Python“ fördert in vielen Fällen auch eine nutzbare Dokumentation zutage. PBKDF2 ist Teil der `hashlib`, die zum Standardumfang von Python gehört und daher nicht nachinstalliert werden muss. Diese Funktion importieren Sie im Programm mit diesem Befehl:

```
from hashlib import pbkdf2_hmac
```



Was eine Anweisung zurückgibt, lässt sich auf der interaktiven Konsole ausprobieren. Gestartet wird sie auf der Konsole mit „python3“ ohne Angabe eines Dateinamens oder unter Windows auch aus IDLE heraus (im Menü „Run“). Wer das IPython Notebook verwendet, muss nicht einmal eine Konsole starten, da die Anweisung dort direkt im Browser ausgeführt wird.

Danach lässt sich `pbkdf2_hmac` aufrufen, als sei die Funktion im Quelltext von c’t SESAM definiert.

Nach der umfangreichen Vorbereitung der Daten ist der Aufruf des Algorithmus eher unspektakulär:

```
hashed_bytes = pbkdf2_hmac(
                    "sha512",
                    hash_string_bytes,
                    salt_bytes,
                    4096)
```

Es spricht übrigens nichts dagegen, Platz zu sparen und die Strings direkt bei der Übergabe an den Algorithmus in Bytefolgen zu konvertieren:

```
hashed_bytes = pbkdf2_hmac(
          "sha512",
          hash_string.encode("utf-8"),
          salt.encode("utf-8"),
          4096)
```

So entfällt die Definition von `hash_string_bytes` und `salt_bytes`. Das macht den Code zwar nicht schneller, aber kompakter.

Passwort erzeugen

PBKDF2 erzeugt eine Ausgabe mit 512 Bit, die c’t SESAM in `hashed_bytes` als Folge von 64 Bytes speichert. Diese Bytes sind aber lediglich binäre Zahlen, die sich nicht einfach per Encoding in ein benutzbares Passwort verwandeln lassen.

Um ein Passwort zu erhalten, muss c’t SESAM aus diesen Bytes einen String konstruieren, der aus Zeichen besteht, die sich für ein Passwort eignen. Da dies eine in sich abgeschlossene Aufgabe ist, empfiehlt es sich, diesen String in einer eigenen Funktion zu erzeugen.

Zu Beginn des Programms haben Sie schon die vordefinierten Funktionen `input()`, `print()` und `len()` aufgerufen. Jetzt lernen Sie eigene Funktionen zu schreiben. Mit Funktionen können Sie Programmtext, der mehrmals benötigt wird, zentral an einer Stelle definieren. Außerdem eignen sich Funktionen zur Strukturierung. Sie werden in Python mit dem Schlüsselwort `def` eingeleitet. Nach einer Leerstelle folgen der Name der Funktion und in runden Klammern ihre Parameter. Kommas trennen mehrere Parameter voneinander, keine Parameter sind auch erlaubt. Nach der schließenden Klammer folgt ein Doppelpunkt. In der nächsten Zeile beginnt der eigentliche Programmtext der Funktion, der wie bei der `if`-Anweisung bezie-

hungsweise der `while`-Schleife eingerückt sein muss. Beispiel:

```
def addition(zahl1, zahl2):
    ergebnis = zahl1 + zahl2
    return ergebnis
```

Python kann den Aufruf einer Funktion erst verarbeiten, wenn sie vorher definiert oder importiert wurde. Daher müssen Sie den ganzen Block der Funktion zur Erzeugung des Passworts in Ihrem Programm oberhalb ihres Aufrufs einfügen. Zur besseren Übersicht fügen Sie sie vor der ersten Zeile Ihres Programms ein.

Die Funktion `convert_bytes_to_password()`, der komplexeste Teil von c't SESAM, soll jeweils ein Zeichen auswählen, das an das bisherige Passwort angehängt wird. Dafür muss für das Passwort zunächst die Variable als String definiert werden:

```
password = ""
```

c't SESAM verwendet für die Passwörter nur Zeichen, die üblicherweise von Websites angenommen werden. Wir haben das ganze Alphabet mit kleinen und großen Buchstaben ausgewählt, wobei wir zwei große Buchstaben ausgelassen haben, die leicht mit anderen Zeichen verwechselt werden. Das sind das I, da es leicht mit dem l verwechselt wird, und das O, da es der 0 ähnlich sieht. Zur Auswahl gehören zusätzlich die Ziffern 0 bis 9 und einige gebräuchliche Sonderzeichen. Sie können die Zeichenauswahl und Reihenfolge ändern, wenn Sie das möchten. Daraus ergäben sich aber Passwörter, die nicht mehr zu unserer Implementierung passen.

Damit `convert_bytes_to_password()` das Passwort Zeichen für Zeichen zusammensetzen kann, muss das Programm die Liste der Zeichen so vorbereiten, dass das leicht geht. Es bietet sich daher an, die Zeichen als Folge von Strings mit jeweils einem Zeichen vorzubereiten.

In Python verwendet man hierzu eine Liste. Diese wird in Python mit eckigen Klammern definiert (`[element1, element2]`) und kann verschiedene Datentypen enthalten. In anderen Programmiersprachen begegnet Ihnen diese Datenstruktur oft unter der Bezeichnung Array. Eine Möglichkeit wäre, die Liste als `password_characters = ["a", "b", "c"]` und so weiter zu definieren. Bei insgesamt 83 Zeichen würde das aber eine sehr mühsame Zeile.

Die Funktion `list()` konvertiert einen mehrere Zeichen langen String in eine Liste aus Strings mit je einem Zeichen. Damit lässt sich die Zeichenliste wesentlich platzsparender definieren. Zur besseren

Übersicht haben wir die Buchstaben, Zahlen und Sonderzeichen als einzelne Listen definiert. Ähnlich wie Strings können diese Listen mit + aneinandergehängt werden:

```
lower_case_letters = list(
        "abcdefghijklmnopqrstuvwxyz")
upper_case_letters = list(
         "ABCDEFGHJKLMNPQRTUVWXYZ")
numbers = list("0123456789")
special_characters = list(
          '#!"§$%&/()[]{}=-_+*<>;:.')
password_characters = ↲
              ↳lower_case_letters + ↲
              ↳upper_case_letters + ↲
              ↳numbers + ↲
              ↳special_characters
```

Insgesamt enthält die Zeichenliste `password_characters` damit alle 83 Zeichen, die im Passwort verwendet werden können. Um ein Zeichen aus der Liste auszuwählen, schreiben Sie einfach die Nummer des gewünschten Zeichens in eckigen Klammern hinter die Liste. Beispielsweise wählt `password_characters[0]` das a aus und `password_characters[51]` die 2.

`convert_bytes_to_password()` soll das Ergebnis von PBKDF2 zur Auswahl der Zeichen verwenden. Dafür interpretiert das Programm die 64 Bytes als sehr große ganze Zahl.

```
number = int.from_bytes(hashed_bytes,
                        byteorder="big")
```

Bevor Sie gelernt haben, mit Kommazahlen zu rechnen, haben Sie in der Schule bestimmt auch schrift-

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
from hashlib import pbkdf2_hmac

lower_case_letters = list("abcdefghijklmnopqrstuvwxyz")
upper_case_letters = list("ABCDEFGHJKLMNPQRTUVWXYZ")
numbers = list("0123456789")
special_characters = list("#!"§$%&/()[]{}=-_+*<;:.")
password_characters = lower_case_letters + upper_case_letters + numbers + special_characters
salt = "pepper"

def convert_bytes_to_password(hashed_bytes, length):
    number = int.from_bytes(hashed_bytes, byteorder="big")
    password = ""
    while number > 0 and len(password) < length:
        password = password + password_characters[number %
                len(password_characters)]
        number = number // len(password_characters)
    return password

master_password = input("Masterpasswort: ")
domain = input("Domain: ")
while len(domain) < 1:
    print("Bitte gib eine Domain an, für die das Passwort generiert werden soll.")
    domain = input("Domain: ")
hash_string = domain + master_password
hashed_bytes = pbkdf2_hmac("sha512", hash_string.encode("utf-8"), salt.encode("utf-8"), 4096)
print("Passwort: " + convert_bytes_to_password(hashed_bytes, 10))
```

Dies ist der gesamte Programmtext von c't SESAM. Mit nur 22 Zeilen lassen sich in Python sichere, individuelle Passwörter für beliebige Domains berechnen.

lich dividiert. Dabei kam heraus, wie oft der Divisor in die Zahl passt und es blieb ein Rest übrig. c't SESAM nutzt diese Art der Division, um die Zeichen des Passworts auszuwählen: Teilt man `number` durch 83, bleibt ein Rest von 0 bis 82, der als Index für die Liste mit den Passwort-Zeichen dient.

Den Ganzzahlquotienten, also wie oft die Zeichenzahl in `number` passt, erhalten Sie mit dem `//`-Operator. Den sollten Sie nicht mit der normalen Division verwechseln, da `zahl / andere_zahl` in Python immer eine Gleitkommazahl ergibt, also eine Zahl mit Nachkommastellen.

Den Rest der Division berechnet der Modulo-Operator %. Mit `number % len(password_characters)` erhalten Sie also immer eine Zahl zwischen 0 und `len(password_characters)`, der Anzahl der Zeichen in der Liste. In unserem Fall beträgt sie 83. Mit dem Rest können Sie abhängig von `number` ein Zeichen für das Passwort auswählen und an das bisherige Passwort anhängen:

```
password = password + ↵
    ↳password_characters[
    number % len(password_characters)]
```

Damit c't SESAM nicht immer dasselbe Zeichen auswählt, muss das Programm anschließend die Ganzzahldivision ausführen, die `number` einen neuen Wert zuweist:

```
number = number // len( ↵
                ↳password_characters)
```

Im Prinzip können diese Schritte in einer `while`-Schleife so lange wiederholt werden, bis `number` 0 wird. Dadurch entstünde ein enorm langes Passwort. Das nimmt aber kaum eine Webseite entgegen. Deshalb übergeben Sie `convert_bytes_to_password()` die gewünschte Länge des Passworts als Parameter `length`. Um die Länge nicht zu überschreiten, hört c't SESAM auf, weitere Zeichen an das Passwort zu hängen, wenn `len(password) < length` unwahr wird. Die ganze Funktion sieht dann so aus:

```
def convert_bytes_to_password(
                hashed_bytes, length):
  number = int.from_bytes(
        hashed_bytes, byteorder="big")
  password = ""
  while number > 0 and ↵
            ↳len(password) < length):
    password = password + ↵
        ↳password_characters [number %
            len(password_characters)]
    number = number // len(
                password_characters)
  return password
```

Ein logisches Und verknüpft die beiden Bedingungen zu Beginn der `while`-Schleife. In der letzten Zeile gibt die Funktion das generierte Passwort zurück.

Ausgeben und beenden

Um die Funktion zur Passwortgenerierung zu benutzen, muss das Programm sie mit den passenden Parametern aufrufen. Der erste Parameter muss die Bytes enthalten, die PBKDF2 aus der Domain und dem Masterpasswort erzeugt hat. Als zweiter Parameter wird die gewünschte Länge des Passworts übergeben. Für die meisten Webseiten eignen sich zehn Zeichen. Wird ein längeres oder kürzeres Passwort benötigt, kann an dieser Stelle das Programm geändert werden. Wichtig: Diese Änderung verstellt die Erzeugung aller Passwörter. Sollten Sie schon mit c't SESAM erzeugte zehnstellige Passwörter einsetzen, müssen Sie den Wert wieder zurücksetzen, damit das Programm Ihnen diese Kennwörter wieder berechnen kann.

Das Ergebnis von `convert_bytes_to_password()` ist ein String, den Sie zusammen mit dem Hinweis, dass das Passwort folgt, über `print()` ausgeben können:

```
print("Passwort: " +
  convert_bytes_to_password(
                        hashed_bytes, 10))
```

Da nach dieser Zeile kein weiterer Programmtext folgt, beendet sich Python. Das Passwort bleibt auf der Konsole stehen, von wo aus Sie es abtippen oder kopieren können. Wenn Sie das Programm unter Windows mit einem Doppelklick starten möchten, können Sie hinter die letzte Zeile ein `input()` setzen. Dann schließt sich das Fenster erst, nachdem Sie ein weiteres Mal die Eingabe-Taste gedrückt haben.

Umgang mit Fehlern

Wer programmiert, macht Fehler und sollte daher mit Fehlermeldungen umgehen können. Python reagiert bei Fehlern mit „Exceptions“, die eine ganze Menge darüber aussagen, was gerade falsch läuft. Es lohnt sich, absichtlich ein paar Fehler einzu-

bauen, um auszuprobieren, wie die Programmiersprache reagiert. Bei einer Exception gibt Python standardmäßig ein Traceback aus. Darin steht ganz unten die Stelle, an der der Fehler wirklich aufgetreten ist. Darüber stehen alle Aufrufe, die an diesem Code beteiligt waren. Werden einer Funktion Parameter des falschen Typs übergeben, tritt der Fehler zwar innerhalb der Funktion auf. Die Fehlerquelle liegt aber in dem Aufruf, den das Traceback direkt darüber anzeigt. Vertauschen Sie mal beim Aufruf von `convert_bytes_to_password` die Parameter und führen Sie das Programm aus, um die Folgen zu beobachten.

Der Typ einer Exception sagt viel darüber aus, wo der Fehler liegen könnte. „SyntaxError“ deutet auf einen falsch formatierten Quelltext hin oder auf einen Tippfehler. Wenn Sie beispielsweise ein `while` in `wile` ändern, zeigt Python Ihnen sofort, in welcher Zeile der Fehler steht. Bei einem „TypeError“ wurde vermutlich eine Variable mit dem falschen Datentyp übergeben. Diesen Fehler produzieren Sie beispielsweise, wenn Sie die Parameter von `convert_bytes_to_password` vertauschen.

Die Meldung „TypeError: 'int' object is not iterable“ teilt Ihnen mit, dass `int.from_bytes()` einen Datentyp erwartet, der wie eine Liste aus einzelnen Elementen besteht. Im Traceback steht, in welcher Reihenfolge Sie die Parameter an die Funktion über-geben haben. „IndexError“ deutet darauf hin, dass das Programm eine größere Liste erwartet hat. Schreiben Sie mal `password_characters[85]` hinter die Definition von `password_characters` und führen das Programm aus, um diesen Fehler zu provozieren.

Sollten Sie auf unerwartete oder unverständliche Fehler stoßen, hilft das Internet weiter. Wenn Sie den Text einer Fehlermeldung in eine Suchmaschine eingeben, fördert das in vielen Fällen Beiträge von Programmierern zutage, die das gleiche Problem hatten und deren Fragen schon beantwortet wurden. Suchen Sie dabei nach dem allgemeinen Typ des Fehlers, lassen Sie aber Eigenheiten des eigenen Programms wie den Traceback und Variablen- oder Funktionsnamen weg. Beginnt eine Zeile mit einem #, interpretiert Python sie als Kommentar und führt sie nicht aus. Damit können Sie sich Notizen im Quelltext machen oder schnell mal eine Zeile deaktivieren.

c’t SESAM benutzen

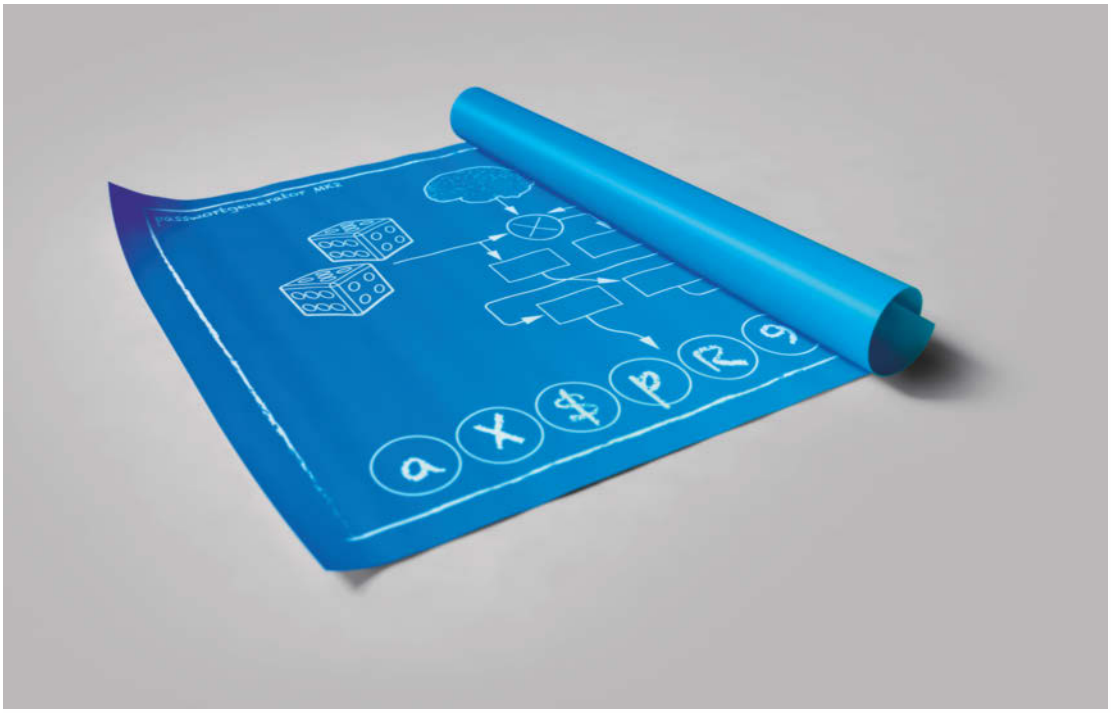
c’t SESAM ist kein reines Anfängerbeispiel: Das Programm generiert sehr sichere Passwörter. Selbst wenn ein Angreifer das Salz kennt, müsste er einen hohen Aufwand treiben, um aus dem Passwort einer Domain das Masterpasswort zu berechnen. Auf einem i5 mit 3 GHz dauert die Berechnung eines einzelnen Passworts gerade mal 0,04 s. Für das Knacken eines 10 Zeichen langen Masterpasswortes hingegen bräuchte der chinesische Supercomputer Tianhe-2 mit einer Brute-Force-Attacke schätzungsweise 5000 Jahre. Ist das Masterpasswort nur halb so lang, wäre er wahrscheinlich in weniger als einer Minute fertig. Denken Sie sich also ein Masterpasswort mit ausreichend vielen Zeichen aus.

Für die meisten Domains wird die Standardeinstellung eines 10 Zeichen langen Passworts mit Sonderzeichen ausreichen. Mit demselben Masterpasswort erzeugt die vorgestellte Version c’t SESAM auf allen Rechnern dieselben Passwörter zu den gleichen Domains. Wenn Sie bei gleichem Domainnamen und Masterpasswort trotzdem andere Passwörter erzeugen wollen, ändern Sie das Salz auf einen anderen Wert. Auf diese Weise erzeugen Sie Ihr ganz persönliches SESAM. Wenn Sie Ihr SESAM auf verschiedenen Geräten einsetzen wollen, müssen Sie auf allen Geräten das gleiche Salz einstellen, damit dieselben Passwörter entstehen.

Für die Domain sollten Sie sich eine Methode überlegen, wie Sie von der URL auf die Domain kommen. Ob Sie „www.paypal.com“, „paypal.com“ oder lieber nur „paypal“ bei der Domain eintragen, hängt von Ihrer Vorliebe ab. Der Passwort-Manager generiert für jede Variante unterschiedliche Passwörter.

Manche Websites akzeptieren keine Sonderzeichen oder begrenzen die Länge des Passworts. In diesem Fall können Sie das Programm kurzzeitig abändern und die Sonderzeichen aus den `password_characters` ausklammern oder die Länge in der letzten Zeile anpassen. Sollten Sie für eine Domain ein neues Passwort brauchen, können Sie die Iterationszahl im Aufruf von `pbkdf2_hmac` um 1 erhöhen. Da alle vorherigen Passwörter mit den alten Einstellungen erzeugt wurden, sollten Sie das Programm nach diesem Sonderfall wieder auf die alten Einstellungen zurückändern. Außerdem sollten Sie sich notieren, welche Passwörter mit anderen Einstellungen erzeugt wurden, damit Sie immer das passende Passwort erzeugen. Da die Einstellungen einem Angreifer nur wenig nützen, können Sie Ihre Notizen auf einem Zettel notieren oder bei einem Cloud-Dienst ablegen. (pmk) c't

Quelltext, iPython-Notebook:
www.ct.de/wrqs



c't SESAM objekt-orientiert erweitern

Für den Python-Crashkurs haben wir einen Passwort-Manager namens c't SESAM programmiert. An einer erweiterten Version lernen Sie jetzt Grundsätze objektorientierter Programmierung, den Import eigener Module, Lesen und Schreiben von Dateien, Verschlüsselung mit AES und einen Hauch Fehlerbehandlung.

Von Pina Merkert

Unser Crashkurs-Beispiel c't SESAM berechnet in nur 22 Zeilen Python-Code aus einem Masterpasswort und einer Domain sichere Passwörter (siehe Seite 8). Für den Alltagseinsatz ist es allerdings niemandem zuzumuten, die Einstellungen für jede Domain im Kopf zu behalten. Die erweiterte Version soll sich deshalb Einstellungen merken und auch Passwörter sicher speichern, die sie nicht aus dem Masterpasswort berechnen kann.

Zusätzlich war es uns wichtig, alle Passwörter – auch das Masterpasswort – ändern zu können. Damit das geht, darf c't SESAM die Passwörter nicht

mehr direkt aus dem Masterpasswort berechnen, da sie sich sonst alle mit jedem neuen Masterpasswort ändern würden. Stattdessen entschlüsselt das Masterpasswort einen sogenannten Schlüsselerzeugungsschlüssel (key generation key, KGK), mit dem c't SESAM die Passwörter erzeugt. Ändert sich das Masterpasswort, bleibt der KGK gleich. Damit blieben auch die Dienstpasswörter unverändert.

Durch diese neue Struktur muss c't SESAM den KGK absolut sicher speichern. Dafür verschlüsselt es ihn mit AES256 und einem Schlüssel, den es per PBKDF2 aus dem Masterpasswort berechnet. Da der KGK aus 64 Bytes kryptografischer Zufallszahlen besteht und bei der Verschlüsselung kein Padding nötig ist, kann ein Angreifer nicht wissen, ob er den KGK korrekt entschlüsselt hat, ohne mit ihm Dienstpasswörter zu erzeugen oder Einstellungen zu entschlüsseln. Der Aufwand für einen solchen Angriff ist auch auf Supercomputern enorm. Die neue Struktur schwächt die Sicherheit von c't SESAM also nicht.

Module bilden

Damit bei diesen Features nicht die Übersicht verloren geht, muss der Quelltext sinnvoll strukturiert sein. Die Berechnung der Passwörter sollte ein anderer Teil der Anwendung übernehmen als die Verwaltung von Einstellungen. Einstellungen für eine Domain gehören wiederum zusammen und das Programm sollte sie immer gemeinsam bearbeiten. Damit ergeben sich drei Module, die das Programm aufteilen: der Generator `CtSesam`, `PasswordSetting`-Einheiten und ein `PasswordSettingsManager`, der die Einstellungen verwaltet und speichert.

Zur sauberen Strukturierung von Programmteilen unterstützt Python objektorientierte Programmierung. Die Objekte in diesem Programmierparadigma speichern Variablen in ihrem eigenen Speicherbereich und bieten Schnittstellen, um mit ihnen zu interagieren. Das Programm beschreibt die Objekte als sogenannte Klassen, die wie Blaupausen wirken: Aus einem Bauplan für Objekte gleicher Art erzeugt die Programmiersprache Instanzen, die jeweils ihre eigenen Variablen haben. Variablen innerhalb eines Objekts bezeichnet man als Eigenschaften, Funktionen innerhalb eines Objekts als Methoden.

SESAM, objekte dich!

Klassen definiert Python mit dem Schlüsselwort `class`, hinter dem der Name der Klasse steht. Der Name ist gleichzeitig der Name des Datentyps der Objekte, die Python mit dieser Klasse erzeugen kann. Ein Doppelpunkt schließt die Zeile ab. Wie bei Python üblich, steht der Programmtext der Klasse in den folgenden eingerückten Zeilen. Klassennamen sollten in Python mit einem Großbuchstaben zu Beginn jedes Worts beginnen, etwa `NameInCamelCase`.

Methoden stehen innerhalb der Klasse und haben einen ersten Parameter namens `self`. Über `self` greift der Code der Methode auf Eigenschaften des Objekts zu. Beim Aufruf der Methode wird der Parameter weggelassen und stattdessen abgetrennt durch einen Punkt vor den Methodennamen geschrieben.

Die Zuweisung `sesam = CtSesam()` erzeugt aus dem Bauplan `CtSesam` ein eigenständiges Objekt oder mit anderen Worten eine Instanz der Klasse:

```
sesam = CtSesam()
sesam.set_salt(b'pepper')
```

Das Beispiel ruft die Methode `set_salt` auf dem Objekt `sesam` auf. Innerhalb der Methode referenziert `self` dann das Objekt, das beim Aufruf in `sesam` steht.

Das `b` vor `'pepper'` gibt an, dass dies kein String ist, sondern der Datentyp `bytes`, der eine Kette von 8-Bit-Zahlen speichert. Innerhalb der Anführungszeichen dürfen entweder ASCII-Zeichen stehen oder hexadezimale Zahlen mit zwei Stellen. Beispielsweise gilt `b'\x63\x74' == b'ct'`.

Die Methode mit dem besonderen Namen `__init__` führt Python automatisch beim Erzeugen jedes Objekts aus. Die objektorientierte Programmierung nennt diese besondere Methode „Konstruktor“. Sie versieht Eigenschaften mit Initialwerten und schafft Strukturen. Beim Passwort-Generator setzt sie die Zeichenmenge und das Salt auf Initialwerte, damit `generate` von Anfang an arbeiten kann. Es gilt als guter Stil, alle Eigenschaften eines Python-Objekts im Konstruktor zu initialisieren.

Einstellungen

Wenn ein `CtSesam`-Objekt Passwörter berechnet, braucht es einen Satz an Einstellungen wie Länge, Iterationsanzahl, Nutzername, Salt und Zeichenauswahl. Jeder Satz an Einstellungen gehört zusammen und bildet daher in der Logik objektorientierter Programmierung ein Objekt. Die zugehörige Klasse ist definiert in der Datei PasswordSetting.py:

```
class PasswordSetting:
  def __init__(self, domain):
    self.domain = domain
    self.url = None
    self.username = None
    self.legacy_password = None
    self.notes = None
    self.iterations = 4096
    self.salt = Crypter.create_salt()
    self.creation_date = datetime.now()
    self.modification_date = ↵
                  ↳self.creation_date
    self.extra_characters = ↵
          ↳DEFAULT_CHARACTER_SET_EXTRA
    self.template = 'x' * 10
    self.calculate_template(
              True, True, True, True,)
    self.synced = False
```

Zusätzliche Parameter in `__init__` müssen beim Erzeugen eines Objekts angegeben werden:

```
ct_setting = PasswordSetting('ct.de')
```

Der Parameter im Konstruktor stellt sicher, dass es nie Einstellungen ohne Domain-Eigenschaft gibt.

Um die einzelnen Einstellungen auszulesen, bieten `PasswordSetting`-Objekte sogenannte „Getter“ an. Die Namen dieser Methoden beginnen üblicherweise mit „get“ und sie liefern die Daten zurück, die ihr Name verspricht. Beispielsweise liefert die Methode `get_domain` den Domain-Namen. Meist liefern Getter einfach Werte von Eigenschaften, sie können ihre Rückgabewerte aber auch berechnen.

Passend dazu sorgen „Setter“ dafür, dass Eigenschaften gespeichert werden. Die Einstellung zur Iterationsanzahl liest und setzt das Programm beispielsweise so:

```
def get_iterations(self):
  return self.iterations
def set_iterations(self, iterations):
  if self.iterations != iterations:
    self.synced = False
  self.iterations = iterations
```

Der Setter speichert nicht nur die neue Iterationsanzahl, sondern vermerkt auch, dass nicht alle Eigenschaften synchronisiert wurden, falls sich die Anzahl wirklich ändert.

Getter und Setter trennen die Daten von ihrer Speicherung, sodass die Objekte die Eigenschaften und Rückgaben auch berechnen oder konvertieren können. Programmierer definieren damit Schnittstellen, die leicht zu nutzen und völlig unabhängig von der internen Implementierung der Klasse sind.

Verwalter

Im Idealfall muss man die interne Struktur eines Objekts nicht mehr verstehen, wenn die Klasse fertig implementiert ist. An dieser Stelle wird klar, dass die Erfinder objektorientierter Programmierung gleich an Teams gedacht haben, bei denen nicht alle Entwickler den Überblick über den gesamten Programmtext behalten können.

Jedes `PasswordSetting`-Objekt speichert nur einen einzelnen Datensatz. Wichtig für den Passwort-Manager sind aber Fragen wie: Gibt es schon eine Einstellung für die Domain ct.de? Solche Fragen können diese Objekte nicht beantworten. Dafür hat das Programm einen Verwalter, der Settings lädt, speichert, bei Bedarf erzeugt und aktualisiert. Das übernimmt die Klasse `PasswordSettingsManager`:

```
class PasswordSettingsManager:
  #...
  def get_setting(self, domain):
    for setting in self.settings:
      if setting.get_domain()==domain:
        return setting
    setting = PasswordSetting(domain)
    self.settings.append(setting)
    return setting
```

Sie hält in der Eigenschaft `settings` eine Liste mit `PasswordSetting`-Objekten. Wird ein `PasswordSettingsManager` mit `get_setting` nach Einstellungen zu einer Domain gefragt, antwortet er entweder mit bereits gespeicherten Einstellungen, falls diese existieren, oder erzeugt ein neues `PasswordSetting`-Objekt.

Er speichert auch Einstellungen in einer Datei, um sie beim Programmstart laden zu können. Zum Laden und Speichern bietet er Methoden an.

Eine Datei funktioniert in Python wie eine Schatztruhe: Wer an ihren Inhalt will, muss die Truhe mit `file = open(dateiname, optionen)` öffnen. Danach erlaubt sie das Lesen (`file.read()`) oder Schreiben (`file.write(daten)`). Wenn man fertig ist, schließt man die Truhe mit `file.close()`. Je nachdem, ob man lesen oder schreiben möchte, gibt man beim Öffnen als Optionen `'r'` oder `'w'` als String an. Liest oder schreibt man statt druckbarer Zeichen lieber Bytes, setzt man einfach noch ein `b` davor (`'br'` oder `'bw'`).

Die Daten soll der Verwalter ins JSON-Format konvertieren, komprimieren und mit dem KGK verschlüsseln. Ein JSON-Konverter ist bei Python dabei und lässt sich mit einer Zeile importieren:

```
import json
```

Der JSON-Konverter benötigt die Daten als Struktur aus Dictionaries und Listen. Dictionaries funktionieren ähnlich wie Listen, erlauben aber statt einer Zahl als Index einen beliebigen Schlüssel. Sie ordnen also jedem Schlüssel einen Wert zu. c't SESAM verwendet für die Schlüssel nur Strings, die Werte sind Strings, Zahlen, Listen oder weitere Dictionaries. Für bessere Übersicht konvertiert der Verwalter die Daten in einer eigenen Methode zu einem Dictionary:

```
def get_settings_as_dict(self):
  settings_list = {'settings': {},
                   'synced': []}
  for setting in self.settings:
    settings_list['settings'][
                  setting.get_domain()
                 ] = setting.to_dict()
    if setting.is_synced():
      settings_list['synced'].append(
                 setting.get_domain())
  return settings_list
```

Der `PasswordSettingsManager` soll die Einstellungen nur verwalten. Komprimierung und Verschlüsselung übernehmen eigene Klassen, die sich durch die Trennung leichter separat testen lassen.

Verpackt und verschlüsselt

Der `Packer` benutzt den Deflate-Algorithmus aus der Bibliothek zlib. Seine Methoden `compress` und `decompress` müssen keine inneren Zustände ändern, sondern nur Daten verarbeiten, die sie beim Aufruf erhalten. Solche Methoden kennzeichnet man in Python mit `@staticmethod`. Die Funktion `decompress` sieht damit beispielsweise so aus:

```
@staticmethod
def decompress(compressed_data):
  return zlib.decompress(
                  compressed_data[4:])
```

Statische Methoden haben keinen ersten Parameter `self`, weil sie nicht zu einer Instanz eines Objekts gehören, sondern zur Klasse. Sie lassen sich aufrufen, ohne eine Instanz der Klasse erzeugen zu müssen.

Die vier Bytes, die `decompress` nicht entpackt, enthalten die Länge des unkomprimierten Datensatzes als 32-Bit-Integer.

Der `Crypter` ver- und entschlüsselt übergebene Daten mit AES im CBC-Modus (Cipher Block Chaining). Da AES einen Schlüssel mit 256 Bit und einen iv mit 128 Bit benötigt, muss der Konstruktor zunächst beide Werte aus der übergebenen Bytefolge extrahieren:

```
def __init__(self, key_iv):
  if len (key_iv) == 48:
    self.key = key_iv [:32]
```

```
    self.iv = key_iv [32:]
  else:
    raise Value Error (
                "Wrong key_iv length")
```

Im CBC-Modus lässt AES in den ersten Block einen Initialisierungsvektor `iv` einfließen, der auf beliebige 16 Bytes gesetzt sein darf. Zum Entschlüsseln muss man AES mit denselben 16 Bytes initialisieren. c't SESAM erzeugt Schlüssel und iv mit einem einzelnen Aufruf von PBKDF2:

```
def create IvKey (password, salt,
                  iterations = 32768)
  return pbkdf2_hmac('sha384',
          password, salt, Iterations)
```

Da AES immer komplette Blöcke von je 16 Bytes verschlüsselt, muss der letzte Block bei den Einstellungen mit einem Padding um einige Bytes verlängert werden. Der `Crypter` verwendet das PKCS7-Padding, bei dem jedes angehängte Byte die Anzahl der angehängten Bytes als 8-Bit-Wert enthält. Die vier Methoden dafür sehen so aus:

```
@staticmethod
def add_pkcs7_padding(data):
  length = 16 - (len(data) % 16)
  data += bytes([length]) * length
  return data
def encrypt(self, data):
  aes_object = AES.new(self.key,
                AES.MODE_CBC, self.iv)
  return aes_object.encrypt(
          self.add_pkcs7_padding(data))
@staticmethod
def remove_pkcs7_padding(data)
  return data[:-data[-1]]
def decrypt(self, encrypted_data):
  aes_object = AES.new(self.key,
                AES.MODE_CBC, self.iv)
  return self.remove_pkcs7_padding(
   aes_object.decrypt(encrypted_data))
```

Beim Entfernen des Paddings schneidet der Code mit einer sehr effizienten Syntax einen Teil einer Liste ab: `l[:5]` gibt eine Kopie der Liste `l` mit den Elementen 0 bis 4 zurück. `l[5:]` liefert eine Liste mit allen Elementen ab dem sechsten. `l[5]` ist das sechste Element der Liste, da diese ja mit `l[0]` beginnt. Praktischerweise gilt `l[:x] + l[x:] == l` für jedes x im Bereich der Liste. `l[:-3]` erzeugt eine Liste ohne die letzten drei Elemente.

Um den verschlüsselten KGK zu verwalten, gibt es eine eigene Klasse `KgkManager`. Sie verwendet ein `Crypter`-Objekt, das den verschlüsselten KGK entschlüsseln kann:

```
self.kgk_crypter = Crypter(
    Crypter.createIvKey(
        password=password, salt=salt))
```

Die Methode `Crypter.createIvKey()` benutzt PBKDF2 mit 32768 Iterationen. Da das auf älteren Rechnern mehr als eine Sekunde braucht, speichert der `KgkManager` das `Crypter`-Objekt und den entschlüssel-

Dictionaries

Listen erlauben den Zugriff auf Elemente über ihre Nummer. Wenn Sie aber einen Wert lieber durch einen beliebigen Schlüssel wie einen String ansprechen wollen, brauchen Sie dafür Dictionaries. Dictionaries stehen in geschweiften statt eckigen Klammern, ihre Elemente bestehen aus Paaren von key: value:

```
hugo = {'Name': 'Hugo', 'Alter': 42}
```

Sie greifen wie bei Listen mit eckigen Klammern auf Elemente zu. Im Beispiel gibt hugo['Alter'] also 42 zurück. Auf die gleiche Art können Sie auch Eigenschaften setzen:

```
hugo['Alter'] = 43
```

Wenn Sie dabei einen Schlüssel benutzen, der bisher nicht im Dictionary vorkam, werden Schlüssel und Wert als neue Einträge gespeichert.

Es gibt 10 Arten von Menschen.

iX-Leser und die anderen.

3 × als Heft

Jetzt Mini-Abo testen:
3 Hefte + Bluetooth-Tastatur
nur 16,50 €

www.iX.de/testen

```
jme@jme: ~/Programmierung/ctSESAM-python-memorizing
jme@jme:~/Programmierung/ctSESAM-python-memorizing$ ./ctSESAM.py
Masterpasswort:
Domain: c
Für die Domain 'ct.de' wurden Einstellungen gefunden.
Sollen sie geladen werden [J/n]? J
Benutzername: jme
Passwort: *.KG(NbM}Ze={C
```

Fängt die angegebene Domain wie ein gespeicherter Datensatz an, fragt c't SESAM, ob es stattdessen diesen Datensatz laden soll.

ten KGK. Zum Speichern der Einstellungen muss PBKDF2 daher nicht nochmal bemüht werden.

Mit dem KGK kann der `PasswortSettingsManager` einen zweiten `Crypter` erzeugen, der die Domaineinstellungen entschlüsselt:

```
settings_crypter = Crypter(
    Crypter.create_key(
        kgk_manager.get_kgk(),
        kgk_manager.get_salt2()) +
    kgk_manager.get_iv2())
```

c't SESAM speichert zusammen mit dem KGK auch ein Salz und einen Initialisierungsvektor für die Verschlüsselung der Einstellungen. Die Methode `Crypter.createKey()` benutzt nur 1024 Iterationen, sodass dieser Befehl deutlich schneller geht. Der `settings_crypter` entschlüsselt dann die Einstellungen:

```
decrypted_settings = ↵
           ↳settings_crypter.decrypt(
                  encrypted_settings)
```

Da c't SESAM auch die Einstellungen für die Synchronisation verschlüsselt speichert, verwirft der `PasswortSettingsManager` diesen Teil der Daten und entpackt den Rest:

```
sync_settings_len = struct.unpack(
     '!I', decrypted_settings[0:4])[0]
decompressed_settings = ↵
           ↳Packer.decompress(
             decrypted_settings[
              4 + sync_settings_len:])
saved_settings = json.loads(str(
               decompressed_settings,
               encoding='utf-8'))
```

`saved_settings` enthält danach die vorher erwähnte Datenstruktur aus Dictionaries und Listen. Aus ihr erzeugt der `PasswortSettingsManager` eine Liste mit `PasswordSetting`-Objekten. Deren Methode `load_from_dict()` kümmert sich dabei um die eigentliche Arbeit.

Beim Speichern läuft der ganze Prozess rückwärts ab: Die Datenstruktur für die JSON-Daten erzeugen, JSON-Daten komprimieren, verschlüsseln, mit dem verschlüsselten KGK kombinieren und die verschlüsselten Daten in die Datei schreiben.

Startpunkt

Neben den wichtigen Klassen muss das Programm noch eine Datei besitzen (`ctSESAM.py`), die beim Start ausgeführt wird. Bevor diese die definierten Klassen nutzen kann, muss sie sie importieren:

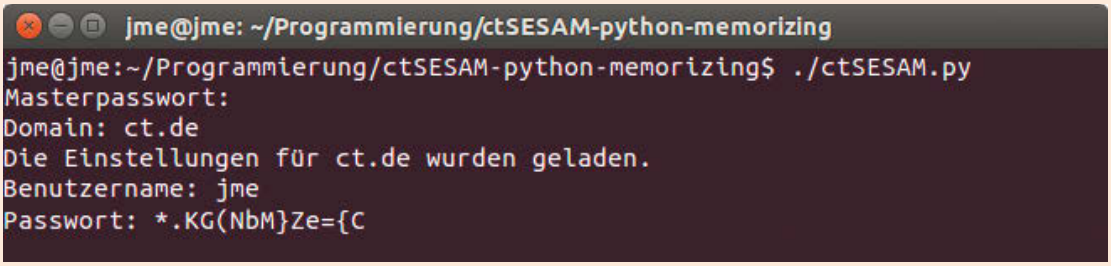
```
from password_generator import CtSesam
from preference_manager import ↵
                    ↳PreferenceManager
from kgk_manager import KgkManager
from password_settings_manager import ↵
                ↳PasswordSettingsManager
from base64 import b64decode
import argparse import getpass
```

Der letzte Import lädt ein externes Modul. Es dient dazu, Passwörter so abzufragen, dass sie nicht auf der Konsole ausgegeben werden:

```
master_password = getpass.getpass(
           prompt='Masterpasswort: ')
```

Anschließend erzeugt `ctSESAM.py` einen `PasswordSettingsManager` und weist ihn an, seine Einstellungen zu laden. Das geht schief, wenn das Masterpass-

Findet das Programm die Domain direkt in den gespeicherten Daten, lädt es die Einstellungen ohne Nachfrage.



wort nicht stimmt, was einen `ValueError` erzeugt. Dass ist eine `Exception`, also eine Ausnahme, die der Code in einem `try/except`-Block abfängt:

```
try:
  settings_manager.load_settings(
         kgk_manager, master_password,
         args.no_sync)
  # ...
except ValueError:
  print("Falsches Masterpasswort. " +
   "Es wurden keine Einstellungen " +
   "geladen.")
```

Schlägt `load_settings` fehl, merkt sich das der `PasswordSettingsManager` und wird daher später die Einstellungen nicht überschreiben.

Nach Eingabe der Domain prüft die Software, ob im `PasswordSettingsManager` eine Einstellung gespeichert ist, die mit den eingegebenen Zeichen anfängt. Bei kompletter Übereinstimmung lädt sie direkt die Einstellung. Stimmen die eingegebenen Zeichen mit dem Beginn eines Domain-Namens überein, fragt sie, ob die Einstellung geladen werden soll. Bei mehreren Kandidaten wiederholt die Software die Frage für jeden Kandidaten, solange der Nutzer mit „Nein“ antwortet.

Findet das Programm keine Einstellung, legt es eine neue an. PasswordSetting-Objekte haben eine Methode, um in diesem Fall nach Zeichenauswahl template und Iterationszahl zu fragen. Gibt der Benutzer nichts an, werden jeweils die Standardeinstellungen konfiguriert. Falls das Masterpasswort gestimmt hat, werden die neuen Einstellungen anschließend verschlüsselt auf der Festplatte gespeichert. Danach berechnet c’t SESAM das Passwort anhand der Einstellungen und gibt es aus. Dank der objektorientierten Struktur ist dieser Schritt übersichtlich:

```
sesam = CtSesam(
    password_setting.get_domain(),
    password_setting.get_username(),
    kgk,
    password_setting.get_salt(),
    password_setting.get_iterations())
password = sesam.generate(
                    password_setting)
if quiet:
  print(password)
else:
  print("Passwort: "+ password)
```

SESAM: „Nutze mich!“

Nach dem Prinzip eines klassischen Passwort-Safes verwaltet c’t SESAM mit dieser Version die Einstellungen. Da diese auch ein Feld für klassische Passwörter vorsehen, verwaltet c’t SESAM jetzt auch die Passwörter, die Sie nicht ändern können.

Den vollständigen Quelltext finden Sie im GitHub-Repository des Projekts. Die objektorientierte Struktur und dieser Artikel sollten es Ihnen leicht machen, den Quelltext zu verstehen. Es gibt aber auch eine englische Dokumentation zum Quelltext.

Das Programm im Repository enthält zusätzlich Klassen für die Synchronisation, die kompatibel mit QtSESAM ist.

Wenn Ihnen Fehler auffallen oder Sie sich zusätzliche Funktionen wünschen, möchten wir Sie animieren, sich an dem Projekt zu beteiligen. c’t SESAM erscheint als freie Software unter der GPL. Wir freuen uns auf Ihre Pull-Requests. (pmk) c't

Quelltexte und Dokumentation:
www.ct.de/w6au

Bild: Sophia Sanner

Spiele mit Python und Pygame, Teil 1

Computerspiele selbst zu programmieren klingt kompliziert, muss es aber nicht sein: Wir zeigen, wie mit Python und dem Spiele-Baukasten Pygame ein erstes Projekt überraschend leicht gelingt.

Von Pit Noack

Die Skriptsprache Python eignet sich prima für den Einstieg in die Programmierung: Die Grundlagen lassen sich leicht lernen und der Code läuft stabil auf allen Betriebssystemen. Zusätzlich gibt es nützliche Erweiterungen für fast jeden Zweck, sogenannte Module, zum Beispiel Pygame. Mit diesem Spiele-Framework entwickeln Sie innerhalb kürzester Zeit Ihr eigenes kleines Pausenfüller-Spiel.

Der Artikel-Zweiteiler zeigt die Pygame-Grundlagen anhand eines kleinen Beispielprojektes. Im Spiel „Knäueljagd“ steuern Sie ein putziges Kätzchen über den Bildschirm. Schnappen Sie sich Wollknäuel, Schmetterlinge und Mäuse, um das Punktekonto zu füllen. Aber weichen Sie Hunden, Wassertropfen und Megafonen aus: Denn nach sieben Zusammenstößen mit diesen für Katzen widerwärtigen Dingen ist das Spiel vorbei. Dieser erste Teil zeigt, wie Sie das Kätzchen als Spielfigur auf den Bildschirm bringen und mit den vier Pfeiltasten senkrecht und waagerecht über den Schirm flitzen lassen. Der zweite Teil komplettiert das Spiel

mit sich frei bewegenden Objekten, Kollisionserkennung und Punktezähler.

Python setzt wie jede Programmiersprache ein gewisses Abstraktionsvermögen voraus, daher eignet sich unser Projekt eher für ältere Kinder beziehungsweise Jugendliche.

Eigene Zeichnungen

Das Netz ist voll von frei verwendbaren Grafikelementen für Spiele. Mit eigenen Zeichnungen verleihen Sie Ihrem Spiel aber individuellen Charme. Greifen Sie und Ihre Kinder also gern selbst zum Zeichenstift, fertigen Sie eigene Spielsymbole an und bauen Sie das Spiel nach Lust und Laune um. Sie benötigen nur Stift, Papier, Scanner oder Scanner-App des Smartphones und ein kostenloses Bildbearbeitungsprogramm wie Gimp, mit dem Sie Bilder freistellen und im PNG-Format speichern. Wie wäre es mit einem Pizzabäcker, der seinen Zutaten hinterherjagt und vor Restaurantinspektoren flüchten muss? Sobald Sie mit Ihrem Kind gemeinsam über Varianten und Erweiterungen der Knäueljagd nachdenken, merken Sie: Beim Programmieren ist Fantasie gefragt.

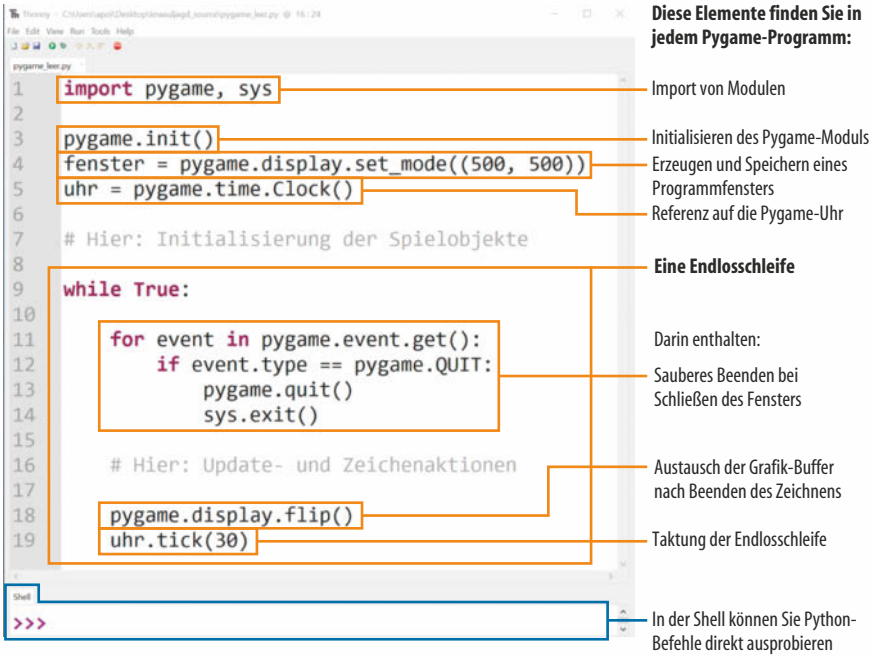
Umgebung einrichten

Bevor Sie mit dem Programmieren beginnen, sollten Sie eine Entwicklungsumgebung installieren. Für unsere Zwecke eignet sich das kostenlose Thonny. Alle im Artikel verwendeten Tools, Skripte und Bilder finden Sie über ct.de/wd6n.

Das Pygame-Modul gehört nicht zur Python-Grundausstattung, lässt sich jedoch leicht nachinstallieren: Starten Sie Thonny und wählen Sie Tools/Terminal. Im Terminal-Fenster tippen Sie den folgenden Befehl ein: `pip3 install pygame`. Damit das funktioniert, muss Ihr Rechner mit dem Internet verbunden sein. Übrigens: Das Raspberry-Pi-System „Raspbian“ hat Thonny und Pygame bereits an Bord.

Thonny und ein leeres Pygame-Programm

Die Thonny-Bedienoberfläche: Im Textbereich tippen Sie den Code ein. Die Shell direkt darunter zeigt zum Beispiel Fehlermeldungen an.



Unser Programm „Katzensolo“ besteht aus zwei Dateien: main.py und katzensolo.py. Wenn die Datei main.py in Thonny geöffnet ist und Sie den Start-Knopf oder F5 drücken, startet das Spiel.

Nehmen Sie jetzt den Code im Listing main.py (auf der nächsten Seite) unter die Lupe: Die erste Zeile lädt per `import` die drei Module `pygame`, `sys` und `katzensolo`. Das Modul `sys` gehört zur Python-Grundausstattung und wird benötigt, um das Spiel sauber zu beenden. Das Modul `katzensolo` ist selbst programmiert, dazu später mehr.

Die Variablen `F_BREITE` und `F_HOEHE` speichern die Ausdehnung des Programmfensters. Es gilt die Regel: Namen von Variablen, die während eines Programms gleich bleiben, schreibt man in Großbuchstaben. Der Befehl `pygame.init()` macht sämtliche Pygame-Module startklar. `fenster = pygame.display.set_mode((F_BREITE, F_HOEHE))` öffnet ein Programmfenster mit den festgelegten Abmessungen und speichert dieses unter dem Namen `fenster`.

Keine Limonade

Zeile 7 legt einen Behälter für Sprites an. Sprite meint hier keine Zitronenbrause, sondern kurz gesagt bewegliche Grafikobjekte wie das Kätzchen.

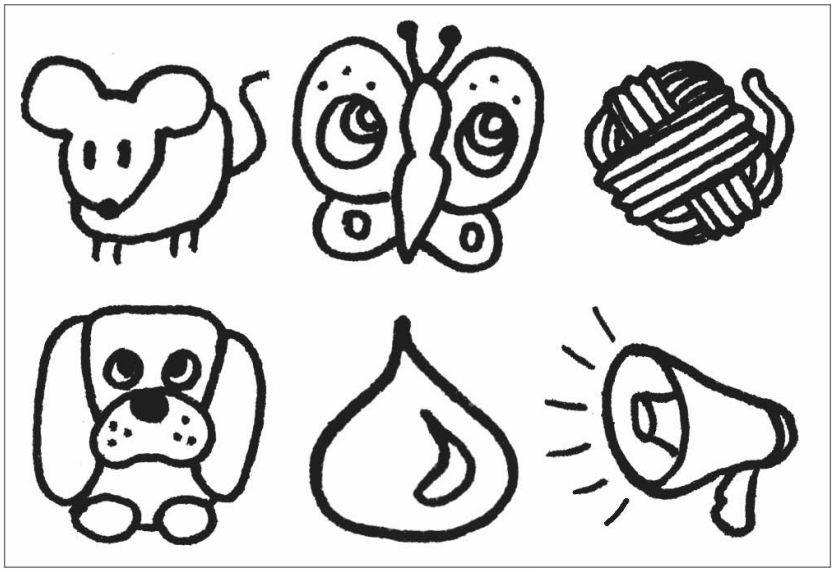
Ein zweidimensionales Computerspiel ist eine Ansammlung solcher Sprites, die miteinander mehr oder weniger friedlich wechselwirken: Raumschiffe, Raketen, Klempner, Mauersteine, Münzen oder eben Katzen und Wollknäuel. Egal, worum es in einem Spiel geht: Immer wird ein Mechanismus benötigt, der Kollisionen dieser Spielobjekte erkennt und entsprechende Aktionen wie Löschen, Einsammeln, Verschieben oder Abprallen auslöst.

Die Geburt der Katze

In Zeile 8 erblickt das Kätzchen das Licht der Welt. Die Zeile sieht etwas seltsam aus, lässt sich jedoch schnell erklären: `Katze()` ist eine Funktion, die Katzen-Objekte erzeugt. Objekte sind Bausteine von Python-Programmen. Die Beschreibung der Katze ist in einem eigenen Modul `katzensolo` definiert.

Die Funktion `Katze()` erhält die Parameter `F_BREITE` und `F_HOEHE` – da die Katze die Abmessungen des Programmfensters kennen muss, damit sie nicht versehentlich ausbüxt. Schließlich speichert Zeile 8 die neugeborene Katze für späteren Zugriff unter dem Variablennamen `katze`.

Zeile 9 fügt der Sprite-Gruppe die Katze hinzu. Auch wenn es vorerst nur ein Sprite gibt: Jedes „lebendige“ Sprite muss in mindestens einer Gruppe sein. Zur späteren Taktung des Spiels speichert Zeile 10 einen Verweis auf die Pygame-Uhr.



Im zweiten Teil kommt Bewegung ins Spiel: Ein gefangenes Knäuel bringt Punkte, während die Begegnung mit einem Hund ein Leben kostet.

Endlosschleife und Ende

Die verbleibenden Zeilen 12 bis 24 bilden eine Endlosschleife. Eine solche Schleife gehört in jedes Pygame-Programm. Pro Schleifendurchgang werden die Zustände aller Spielobjekte wie zum Beispiel deren Position auf dem Bildschirm aktualisiert und anschließend alle Sprites gezeichnet. Die Schleife wiederholt sich, bis der Nutzer das Programmfenster schließt.

Den Beginn einer Code-Schleife markiert das Wörtchen `while`, gefolgt von einer Bedingung und einem Doppelpunkt. Eine Bedingung ist ein Ausdruck, der `True` (wahr) oder `False` (unwahr) ergibt. Die Schleife läuft, solange dieser Ausdruck wahr ist. Da dies bei `True` immer der Fall ist, läuft die Schleife also endlos.

Zusammengehöriger Code – beispielsweise zur Wiederholung in einer `while`-Schleife – wird in Python immer durch eine gemeinsame Einrückungstiefe markiert und als Block oder Suite bezeichnet. Wenn Sie `while True:` eintippen, stellen Sie fest,

dass Thonny die folgenden Zeilen automatisch um vier Leerzeichen einrückt.

Sinn und Zweck der Zeilen 14 bis 17 ist es, das Programm sauber zu beenden, falls der Benutzer das Fenster per Mausklick schließt. Wann immer der Nutzer eine Taste drückt, die Maus bewegt oder das Fenster zu schließen versucht, erzeugt Pygame ein Event-Objekt. Pygame speichert diese Events im Hintergrund automatisch. Der Befehl `pygame.event.get()` liefert eine Liste dieser Events. Diese untersucht das Programm anschließend daraufhin, ob sie das gesuchte `QUIT`-Event enthält. Ist das der Fall, führen `pygame.quit()` und `sys.exit()` zu einem korrekten Abschluss des Programms. Diesen Abschnitt finden Sie so oder ähnlich in jedem Pygame-Programm.

Doppelte Bufferhaltung

Bisher ging es um das technische Drumherum. In den Zeilen 19 bis 21 lassen Sie zeichnen: `fenster.fill((255,255,255))` füllt den Hintergrund weiß. Die drei Zahlen stehen für Rot-, Grün- und Blauanteil, die jeweils von 0 bis 255 gehen. Die Mischung macht die Farbe – experimentieren Sie mit anderen Zahlen für einen farbigen Hintergrund.

`sprites.update()` ruft für alle Sprites in der Gruppe – im Moment nur das Kätzchen – die Funktion `update()` auf. Alle Sprite-Objekte müssen diese Funktion besitzen. Bei der Katze ist sie für die Aktualisierung der Position auf dem Bildschirm zuständig. Zeile 21 malt mit `sprites.draw(fenster)` alle Sprites in das Fenster.

Sie haben soeben einen weißen – oder andersfarbigen – Hintergrund und das Zeichnen des Kätzchens per Code veranlasst. Doch ohne den folgenden Schritt würden Sie nichts als ein schwarzes Fenster sehen. Auch wenn das Programm nur ein Fenster hat, den Bildspeicher (Framebuffer) gibt es zweimal: Während Speicher A zu sehen ist, wird in Speicher B gezeichnet und umgekehrt. Wenn das Zeichnen im gerade unsichtbaren Buffer abge-

```
01  import pygame, sys, katzensolo
02
03  F_BREITE, F_HOEHE = 1000, 600
04
05  pygame.init()
06  fenster = pygame.display.set_mode((F_BREITE, F_HOEHE))
07  sprites = pygame.sprite.Group()
08  katze = katzensolo.Katze(F_BREITE, F_HOEHE)
09  sprites.add(katze)
10  uhr = pygame.time.Clock()
11
12  while True:
13
14      for event in pygame.event.get():
15          if event.type == pygame.QUIT:
16              pygame.quit()
17              sys.exit()
18
19      fenster.fill((255, 255, 255))
20      sprites.update()
21      sprites.draw(fenster)
22
23      pygame.display.flip()
24      uhr.tick(30)
```

Die Datei main.py ist die Steuerzentrale des kleinen Katzenprogramms.

```
01  class Katze(pygame.sprite.Sprite):
02
03      def __init__(self, F_BREITE, F_HOEHE):
04          super().__init__()
05          self.F_BREITE = F_BREITE
06          self.F_HOEHE = F_HOEHE
07          self.image = pygame.image.load("katze.png")
08          self.rect = self.image.get_rect()
09          self.rect.center = (self.F_BREITE / 2, self.F_HOEHE / 2)
10
11      def update(self):
12          gedrueckt = pygame.key.get_pressed()
13          if gedrueckt[pygame.K_UP]: self.rect.y -= 8
14          if gedrueckt[pygame.K_DOWN]: self.rect.y += 8
15          if gedrueckt[pygame.K_LEFT]: self.rect.x -= 8
16          if gedrueckt[pygame.K_RIGHT]: self.rect.x += 8
17          self.rect.clamp_ip(pygame.Rect(0, 0, self.F_BREITE, self.F_HOEHE))
```

Die Datei katzensolo.py ist ein selbstprogrammiertes Modul und definiert die Klasse Katze.

schlossen ist, vertauscht `pygame.display.flip()` in Zeile 23 A und B, und das Gezeichnete erscheint auf dem Bildschirm.

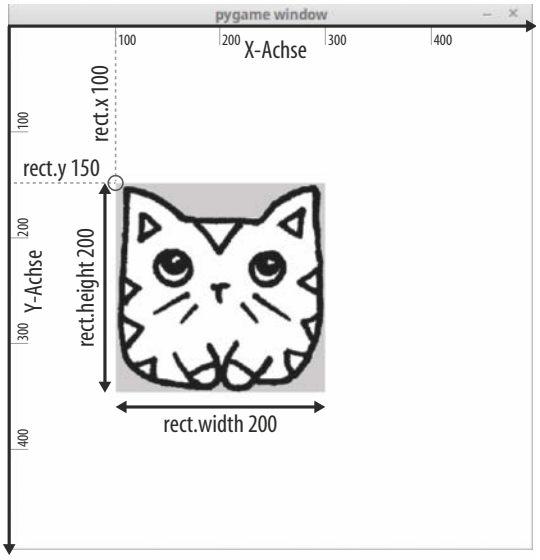
Zu guter Letzt ist die Zeile 24 für die Taktung des Programms zuständig. Die Endlosschleife frischt bei jedem Durchlaufen das Programmfenster auf. Eine Rate von 30 Bildern pro Sekunde lässt die Katze sich flüssig bewegen. Diese Bildwiederholrate übergibt man mit der Funktion `tick()` an ein weiteres Objekt: `pygame.time.Clock`. Sie verzögert die Ausführung so lange, bis exakt eine dreißigstel Sekunde seit dem letzten Aufruf verstrichen ist.

Die Katzen-Klasse

Damit ist der Rundgang durch das Listing main.py abgeschlossen. Die Katze haben Sie dort verwendet wie jedes andere Python-Objekt auch. Doch wo kommt die Katze eigentlich her? Was passiert, wenn Sie wie in Zeile 8 von main.py mit `Katze()` eine neue Katze erzeugen?

Die Datei katzensolo.py definiert die Klasse `Katze`. Eine Klasse ist ein Bauplan für eine bestimmte Sorte von Objekten. Das Wichtigste in Kürze: `class Katze(pygame.sprite.Sprite):` in Zeile 1 zeigt an, dass die Definition einer Klasse mit Namen `Katze` folgt. Sie ist eine Erweiterung der Klasse `Sprite`, das heißt, sie besitzt alle Variablen und Funktionen von `Sprite`-Objekten.

Die Klasse `Sprite` liefert viele nützliche Funktionen, etwa um Kollisionen mit anderen Sprites festzustellen. Sie erledigt vieles automatisch im Hintergrund, um das Sie sich nicht mehr zu kümmern brauchen. `Sprite`-Objekte müssen Variablen für ein Bild (`image`) und ein Rechteck (`rect`) besitzen. Das



Das graue Rechteck bestimmt Ausdehnung und Größe der Katze.

Bild bestimmt das Aussehen des Spielobjektes. Das Rechteck ist nötig, um Ausdehnung und Position zu beschreiben und um festzustellen, ob es zu einer Kollision mit anderen Sprites kommt.

Katzengeburt zum Zweiten

In Zeile 3 bis 9 steht die besondere Funktion `__init__()`, die beim Erzeugen einer Katze automatisch aufgerufen wird. Zeile 4 initialisiert zunächst das übergeordnete `Sprite`-Objekt.

Wann immer in der Klassendefinition der Begriff `self` auftaucht, ist damit das Objekt selbst (also hier die Katze) gemeint. Mit `self.F_BREITE = F_BREITE` merkt sich die Katze also die Breite des Programmfensters. In Zeile 7 lädt `self.image = pygame.image.load("katze.png")` das Katzenbild aus dem Programmordner. `self.rect = self.image.get_rect()` in Zeile 8 ist dafür zuständig, dass das Rechteck und das Bild dieselbe Ausdehnung haben. `self.rect.center = …` positioniert das Rechteck (und damit das Sprite) genau in der Mitte des Fensters.

Grenzen der Bewegung

Die Zeilen 11 bis 17 definieren die Funktion `update()`. Diese Funktion wird bei jedem Durchgang der Endlosschleife aufgerufen und hat zwei Aufgaben: die Position der Katze bei gedrückten Pfeiltasten verändern und die Katze innerhalb der Grenzen des Programmfensters halten.

Damit sich die Katze in die gewünschte Richtung bewegt, benötigt das Programm Informationen darüber, welche Taste gedrückt ist und was in diesem Fall passieren soll. Diese liefert `pygame.key.get_pressed()` in Form eines Dictionarys: Das sind spezielle Objekte, die wie ein Nachschlagewerk funktionieren. Im Falle von `gedruecktpygame.K_UP` ist `K_UP` sozusagen das Schlagwort und liefert `True`, wenn die Pfeiltaste „hoch“ gedrückt ist, sonst `False`. Im ersten Fall verringert sich die y-Koordinate des zur Katze gehörenden Rechtecks, sprich: Die Katze wandert um 8 Pixel nach oben. Die Zeilen 14 bis 16 arbeiten analog. Was passiert, wenn Sie versuchen, das Kätzchen über den Rand zu steuern? `self.rect.clamp_ip()` in Zeile 17 hält das zur Katze gehörende Rechteck innerhalb des Programmfensters fest.

Wie gehts weiter

Sie haben nun einige Grundlagen der Pygame-Programmierung kennengelernt. Wir hoffen, Sie und Ihr Kind haben Freude am Programmieren gefunden. Im folgenden Artikel geht es weiter – dann wird das Spiel fertig programmiert. Bis dahin können Sie ausprobieren, welche Hintergrundfarbe Ihnen am besten gefällt. Und wer mag, zeichnet bis dahin eigene Grafiken oder scannt Fotos ein: Sie könnten zum Beispiel der buckligen Verwandtschaft ausweichen und Freunde einsammeln.

(apoi) c't

Python, Beispielcode, Bilder:
www.ct.de/wd6n

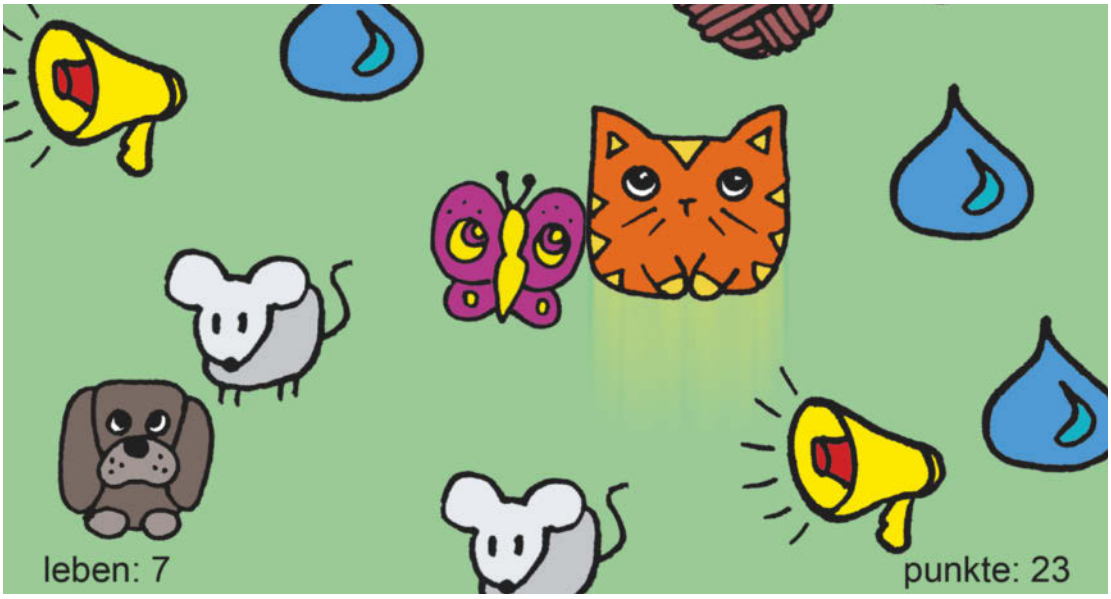


Bild: Sophia Sanner

Spiele mit Python und Pygame, Teil 2

Im ersten Teil der Mini-Serie zur Spieleprogrammierung mit Python haben Sie gelernt, per Pfeiltasten ein niedliches Kätzchen über den Bildschirm wetzen zu lassen. Jetzt bekommt der Stubentiger Gesellschaft und eine Mission.

Von Pit Noack

Dieser Artikel erläutert den erweiterten Code des Programms „Knäueljagd“ (siehe Seite 28). In unserem Spiel schlüpfen Sie in die Rolle einer Katze und müssen möglichst viele Mäuse, Schmetterlinge und Wollknäuel ergattern, aber geschickt Wassertropfen, Hunden und lärmenden Megafonen ausweichen.

Das Programm umfasst rund 120 Codezeilen, die auf zwei Dateien verteilt sind, außerdem sieben Bilddateien. Dieser Artikel nimmt zwei besonders wichtige Codeabschnitte unter die Lupe. Den kompletten Code, die Bilder und eine kurze Installationsanleitung finden Sie über ct.de/wccc.

Aus der Vogelperspektive

Zu den wesentlichen Elementen zweidimensionaler Spiele gehören Sprites. Das sind Grafikobjekte wie Aliens und Klempner oder eben Katzen und Hunde, die sich vor dem Hintergrund bewegen. Der Code eines Spiels beschreibt, nach welchen Regeln sich die Sprites auf dem Bildschirm bewegen und

was bei Kollisionen zwischen den Sprites passieren soll.

In Teil eins haben Sie bereits das Katzen-Sprite im Spiel Katzensolo kennengelernt, das der Spieler mit den Pfeiltasten der Tastatur steuert. Weil einsames Herumflitzen über den Bildschirm auf die Dauer langweilt, kommt in diesem Artikel eine zweite Sorte Sprites hinzu, die wir „ZufallsObjekte“ getauft haben. Wenn das Objekt entsteht, entscheidet der Zufall, ob es gut oder schlecht für die Katze ist.

Sobald die Katze ein `ZufallsObjekt` berührt, verschwindet es vom Bildschirm. Bei Zusammenstößen mit guten Objekten – also Mäusen, Schmetterlingen und Wollknäueln – bekommt die Katze jeweils einen Punkt spendiert. Bei unerwünschten Begegnungen mit schlechten Objekten – also Hunden, Wassertropfen und Megafonen – büßt die Katze ein Leben ein. Hat sie ihre sieben Leben verbraucht, ist das Spiel vorbei.

Wie läuft das im Code? Schauen Sie sich dazu den aufs Wesentlichste gekürzten Codeausschnitt im Kasten unten an. Der Code-Ausschnitt steht in der Datei main.py innerhalb der Endlosschleife. Diese Schleife ist Dreh- und Angelpunkt unseres Spiels und wiederholt sich etwa dreißigmal pro Sekunde. Bei jedem Durchgang der Endlosschleife muss überprüft werden, ob eines der Zufallsobjekte mit der Katze kollidiert ist.

Zusammenstöße behandeln

`sprites` ist ein Behälter für alle Spielobjekte, also die ZufallsObjekte und die Katze. Die `for`-Schleife in Zeile 1 veranlasst, dass jedes Sprite aus diesem Behälter im folgenden Block (Zeile 2 bis 10) mal unter dem Namen `sprite` zur Verfügung steht. Dort überprüft Zeile 2 mit `collide_rect(katze, sprite)`, ob eine Kollision stattgefunden hat. Zusammenstöße der Katze mit sich selbst gelten natürlich nicht: `if sprite != katze` heißt auf Deutsch „Wenn das Sprite nicht die Katze ist“.

Ist das Kätzchen tatsächlich in etwas hineingekracht, wird das Objekt in jedem Fall gelöscht. Das passiert in Zeile 10 durch den `kill()`-Befehl. Dieser Befehl nimmt das Sprite aus dem Behälter `sprites`. Im nächsten Schleifendurchgang existiert das Objekt nicht mehr. Vorher prüft Zeile 3, ob das Objekt gut ist. In dem Fall erhöht `katze.punkte += 1` in Zeile 4 den Punktestand der Katze um 1. Andernfalls geht es in Zeile 6 weiter: Die Katze büßt ein Leben ein. Anschließend prüft Zeile 7, ob die Katze kein Leben mehr hat. Wenn das der Fall ist, beenden Zeilen 8 und 9 den Spielspaß.

In der Datei knaeuljagd.py steht die Definition der Klassen `Katze` und `ZufallsObjekt` sowie eine Funktion zur Ausgabe von Texten auf dem Bildschirm, beispielsweise „GAME OVER“ und Punktestand. Die Katzen-Definition kennen Sie bereits aus Teil 1. Neu dazugekommen sind die Eigenschaften `punkte` und `leben`.

ZufallsObjekte (siehe Listing auf Seite 37) haben viele Gemeinsamkeiten mit Katzen: Beides sind Erweiterungen der Pygame-Klasse `Sprite`, die viele nützliche Funktionen für grafische Spielobjekte mitbringt. Hierfür benötigen Sprites ein Bild (`self.image`) sowie eine Position und eine Ausdeh-

```
01  for sprite in sprites:
02    if sprite != katze and pygame.sprite.collide_rect(katze, sprite):
03      if sprite.gut:
04        katze.punkte += 1
05      else:
06        katze.leben -= 1
07        if katze.leben <= 0:
08          pygame.quit()
09          sys.exit()
10      sprite.kill()
```

Dieser Code stellt fest, ob die Katze in etwas hineingekracht ist. Bei Kollisionen mit genehmen Objekten gibt’s einen Punkt, bei unangenehmen Objekten verliert die Katze eines ihrer sieben Leben.

nung (`self.rect`). auf dem Bildschirm. `update()` aktualisiert den Zustand von Sprites. Genau wie bei jeder anderen Python-Klasse initialisiert die Funktion `__init__()` neue Objekte.

Damit hören die Gemeinsamkeiten auf. Folgende wesentliche Eigenschaften und Verhaltensweisen sollen ZufallsObjekte haben: Sie können gut oder schlecht sein, und abhängig davon wird bei erzeugten Objekten zufällig eines von drei „guten" oder „schlechten" Bildern ausgewählt.

Neu erzeugte ZufallsObjekte haben eine zufällig gewählte Position oberhalb des Programmfensters und bewegen sich von dort selbstständig mit unterschiedlichen Geschwindigkeiten von oben nach unten und zufälligen, veränderlichen Seitwärtsbewegungen über den Bildschirm. Wenn ein Objekt das Programmfenster komplett von oben nach unten durchquert hat, wird es gelöscht.

Die Zeilen 3 bis 9 laden jeweils drei „gute" und „schlechte" Bilder in Listen mit den Namen `bilder_gut` und `bilder_schlecht`. Da dieser Code außerhalb von `__init__()` steht, handelt es sich um Klassenvariablen. Das heißt, diese Variablen sind gemeinsames Eigentum aller Objekte der Klasse `ZufallsObjekt`. Vorteil: Nicht jedes Objekt muss die Bilder einzeln laden, das spart Speicher und Rechenzeit.

ZufallsObjekte erzeugen

In Zeile 16 fällt die schicksalhafte Entscheidung, ob das neue Objekt gut oder schlecht ist. `random.choice((True, False))` wählt hier zufällig zwischen True oder False. Auf Grundlage dieser Entscheidung bewirken die Zeilen 18 bis 21, dass `self.image` auf ein passendes Bild verweist. Auch hier kommt `random.choice()` zum Zuge. An dieser Stelle steht `ZufallsObjekt.bilder_gut` und `ZufallsObjekt.bilder_schlecht`, denn es handelt sich hierbei um Klassenvariablen.

Wie in der Katzenklasse setzt Zeile 23 das zugehörige Rechteck `rect` auf die Ausdehnung des gewählten Bildes. Zur Erinnerung: das Bild ist zuständig für das Aussehen des Sprites, das Rechteck dient der Beschreibung von Ausdehnung und Position und der Feststellung von Kollisionen.

Zu guter Letzt müssen neue Objekte Position und Geschwindigkeit erhalten. Die Zeilen 25 und 26 positionieren das jeweilige Objekt zunächst an einem zufälligen Ort oberhalb des Programmfensters, also außerhalb des sichtbaren Bereiches. Die hierfür benötigten zufälligen ganzen Zahlen liefert

Kostenlose Sprites für Spieleprogrammierung gibt es zuhauf im Netz, zum Beispiel auf kenney.nl.

die Funktion `random.randint()`. Das `int` steht übrigens für „Integer", zu Deutsch: eine Ganzzahl. Die Zeilen 28 und 29 legen die Geschwindigkeit des Objektes fest. Eine Bewegung auf einer Fläche lässt sich in einen horizontalen (x) und vertikalen (y) Anteil zerlegen. `x_speed` ist der Wert, um den sich die horizontale Position des Gegenstandes pro Schleifendurchgang verändern soll, `y_speed` bestimmt die Veränderung der vertikalen Position. `random.randint(-2,2)` liefert eine zufällige Zahl zwischen -2 und 2 und sorgt also für eine zufällige horizontale Bewegung. Da die Gegenstände immer von oben nach unten laufen sollen, bekommt `y_speed` einen zufälligen positiven Wert zwischen 1 und 5.

Updates und Herausforderungen

Damit ist die Funktion `__init__()` komplett. Es bleibt die Funktion `update()` in den Zeilen 31 bis 38, die für die Aktualisierung des Gegenstandes zuständig ist und bei jedem Durchgang der Endlosschleife aufgerufen wird. Zeile 32 prüft, ob der Gegenstand überhaupt noch im Bild ist. `if self.rect.top > self.F_HOEHE` heißt frei übersetzt: „Wenn die Oberkante des Gegenstandes unterhalb der Unterkante des Programmfensters ist." Ist dies der Fall, beseitigt `self.kill()` den Gegenstand.

Andernfalls geht es in den Zeilen 35 und 36 weiter, die Bewegung ins Spiel bringen: `self.rect.x += self.x_speed` addiert die entsprechende Geschwindigkeit auf die x-Position, Gleiches läuft für y. Um das Bewegungsmuster der Gegenstände interessanter zu machen, weisen die Zeilen 37 und 38 der horizontalen Geschwindigkeit einen neuen Wert zu – und zwar im Schnitt alle 60 Schleifendurchgänge (also alle 2 Sekunden). Im Ergebnis segeln die Gegenstände ein wenig anarchisch durchs Bild. Deaktivieren Sie probehalber diese beiden Zeilen mit einem vorgestellten #, um den Unterschied zu erkennen.

Sie haben nun einige sehr wesentliche Elemente des Knäueljagd-Programms genauer kennengelernt. Aber das ist noch nicht alles. Entscheidend

```
01 class ZufallsObjekt(pygame.sprite.Sprite):
02
03   bilder_gut = [pygame.image.load("maus.png"),
04                 pygame.image.load("falter.png"),
05                 pygame.image.load("knaeul.png")]
06
07   bilder_schlecht = [pygame.image.load("wasser.png"),
08                 pygame.image.load("laerm.png"),
09                 pygame.image.load("hund.png")]
10
11   def __init__(self, F_BREITE, F_HOEHE):
12     super().__init__()
13     self.F_BREITE = F_BREITE
14     self.F_HOEHE = F_HOEHE
15
16     self.gut = random.choice((True, False))
17
18     if self.gut:
19       self.image = random.choice(ZufallsObjekt.bilder_gut)
20     else:
21       self.image = random.choice(ZufallsObjekt.bilder_schlecht)
22
23     self.rect = self.image.get_rect()
24
25     self.rect.center = (random.randint(0, self.F_BREITE),
26                         random.randint(-self.F_HOEHE, -self.rect.height))
27
28     self.x_speed = random.randint(-2, 2)
29     self.y_speed = random.randint(1, 5)
30
31   def update(self):
32     if self.rect.top > self.F_HOEHE:
33       self.kill()
34     else:
35       self.rect.x += self.x_speed
36       self.rect.y += self.y_speed
37       if random.randint(0, 120) == 0:
38         self.x_speed = random.randint(-2, 2)
```

Im erweiterten Spiel gesellen sich sechs ZufallsObjekte zur Katze hinzu. Drei von ihnen findet die Katze super, den anderen geht sie besser aus dem Weg.

für den Spielspaß ist eine kleine Gemeinheit: Die Anzahl der im Spiel befindlichen Gegenstände nimmt mit steigender Punktezahl zu. Wäre die Anzahl konstant, würde das Spiel sehr schnell langweilig. Auf diese Weise bleibt es spannend, und es ist nur mit viel Übung möglich, Ergebnisse von mehr als 100 Punkten zu erreichen. Zusätzlich begrenzt dieser Trick die Dauer eines einzelnen Spiels – man soll schließlich nicht zu viel vorm Computer sitzen, außerdem wollen die Mitspieler auch mal ran. Den entsprechenden Code zur Kontrolle der Anzahl von im Spiel befindlichen Objekten finden Sie in der Datei main.py in Zeile 3 und den Zeilen 23 bis 26.

Allerlei Anzeigen

Rückmeldungen machen es dem Spieler einfacher zu erkennen, ob ein guter oder ein schlechter Zusammenstoß passiert ist. Aus Rücksicht auf die Hörorgane Ihrer Mitmenschen haben wir hier keine akustische, sondern eine optische Variante gewählt: Bei Zusammenstoß mit einem guten Objekt flackert der Hintergrund kurz grün auf, sonst rot.

Technisch funktioniert das so, dass die Variable `t_kollision_gut` in Zeile 33 (main.py) den Zeitpunkt der jeweils letzten Kollision speichert, die das Punktekonto des Spielers füllt: `t_kollision_gut = pygame.time.get_ticks()`. Entsprechendes passiert für `t_kollision_schlecht`-Variable in Zeile 36. Bei jedem Schleifendurchgang wird dann die aktuelle Zeit mit diesen gespeicherten Werten verglichen. Sind weniger als 100 Millisekunden vergangen, wird die Füllfarbe entsprechend auf Rot oder Grün gesetzt. Ist die letzte Kollision länger her, ist die Füllfarbe Weiß (main.py Zeilen 47–52):

```
if pygame.time.get_ticks() ↵
    ↳- t_kollision_gut < 100:
  fenster.fill((0, 255, 0))
...
```

Der Spieler möchte gern wissen, wie viele Punkte er gewonnen und wie viele Katzenleben er noch zur Verfügung hat. Die Anzeige erledigt eine selbstgeschriebene Funktion `text()` (knaeuljagd.py), die schlichtweg Textnachrichten in das Programmfenster zaubert:

```
def text(text, fenster,
    position, groesse):
  font = pygame.font.SysFont(
    'arial', groesse)
  text = font.render(text,
    False, (0, 0, 0))
```

Unser fertiges Spiel ist ganz bewusst nicht koloriert. Probieren Sie farbige Hintergründe und bunt eingefärbte Spielobjekte – lassen Sie Ihrer Fantasie freien Lauf.

```
  F_BREITE = text.get_rect().width
  fenster.blit(text, (position[0] - (
    F_BREITE / 2), position[1]))
```

Die Funktion erwartet vier Parameter: Textnachricht, Ausgabefenster, die Position des Textes und dessen Größe. Diese Funktion nutzen wir auch, um bei Spielende „GAME OVER“ und Punktestand anzeigen zu lassen (Zeilen 39, 40 main.py):

```
knaeuljagd.text("GAME OVER", fenster,
  (F_BREITE / 2, F_HOEHE / 2), 50)
knaeuljagd.text(str(katze.punkte) +
  " punkte", fenster, (F_BREITE / 2,
  F_HOEHE / 2 + 60), 30)
```

Auch während das Spiel noch läuft, will der Spieler jederzeit wissen, wie viele Punkte er bereits gesammelt hat und wie viele Leben seine Katze noch besitzt. Die `text()`-Funktion kommt daher auch in Zeile 57 und 58 zum Einsatz und zeigt Punktestand sowie Katzenleben an:

```
knaeuljagd.text("punkte: " + str(
  katze.punkte), fenster, (
  F_BREITE - 100, F_HOEHE - 50), 30)
knaeuljagd.text("leben: " + str(
  katze.leben), fenster, (
  80, F_HOEHE - 50), 30)
```

Weiterbasteln

Wir hoffen, Ihnen macht das Knäueljagd-Programm und das Experimentieren mit dem Code genauso viel Spaß wie uns. Wenn Sie auf den Geschmack gekommen sind, können Sie das Spiel modifizieren oder weiterentwickeln. Wie wäre es mit unterschiedlichen Sorten guter Objekte, die sich in Fortbewegungsart und Punktzahl unterscheiden? Klasse wären die aus anderen Spielen bekannten Vitaminpillen, welche die Katze für eine bestimmte Zeit unverwundbar machen. Wie könnte ein Mechanismus aussehen, der die Überschneidung von Zufallsobjekten auf dem Bildschirm verhindert? Wie müsste man Hindernisse programmieren, die den Bewegungsspielraum von Katze & Co. einschränken? Wie könnte eine Levelstruktur aufgebaut sein und wie könnte ein mächtiger Super-Hund aussehen, der Wassertropfen, Megafone und Blitze umherschleudert?

Individueller gestalten Sie das Spiel, wenn Sie eigene Fotos verwenden. Lassen Sie zum Beispiel Ihre Hauskatze dem Hofhund des Nachbarn aus dem Weg gehen. Falls Sie keine Lust haben, Ihre Fotos freizustellen, können Sie auf vorgefertigte Icon-Sets zurückgreifen. Im Internet gibt es zahlreiche Websites mit kostenlosen und kostenpflichtigen Sammlungen. Die Bilddateien müssen im Programmordner liegen, sollten das PNG-Format haben und freigestellt sein, also einen transparenten Hintergrund (Alpha-Kanal) haben. Achten Sie darauf, dass die Bilder etwa 80 mal 80 Pixel groß sind, da die aktuelle Version der Knäueljagd Sprites nicht automatisch skaliert.

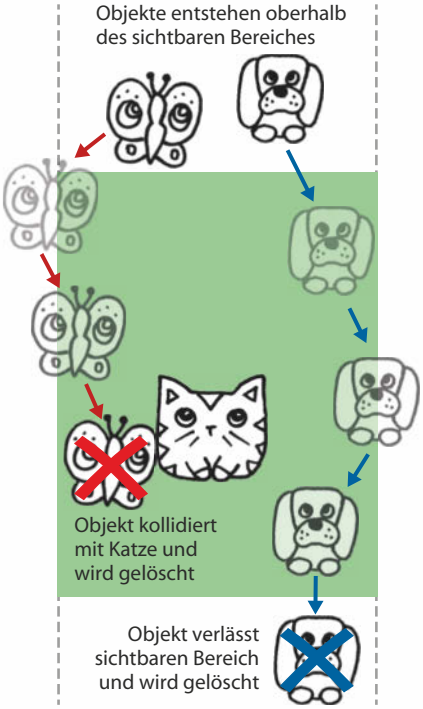
Die Möglichkeiten sind buchstäblich unbegrenzt. Fantasie, Geduld und Übung sind alles, was man braucht, um tolle Spiele zu entwickeln. (apoi) c't

Beispielcode, Bilder:

www.ct.de/wccc

Objektleben

Zwei mögliche Lebenswege von Spielobjekten durch das Programmfenster



Zwei Faktoren mit OAuth2

Wer auf APIs bei Google, Facebook oder GitHub zugreifen möchte, kommt um OAuth2 nicht herum. Das Login-Protokoll authentifiziert Anwendungen, indem es den Nutzer im Browser um Erlaubnis fragt. Damit geht sogar Zwei-Faktor-Authentifizierung.

Von Pina Merkert

Um auf Daten im Google-Account zuzugreifen, müssen sich Python-Programme per OAuth2 authentifizieren und vom Benutzer die Erlaubnis für den Zugriff einholen. Der Workflow dafür ist durchaus komplex [1], doch glücklicherweise hilft die `requests_oauthlib` kräftig mit, sodass das Login mit wenigen Zeilen Python-Code gelingt. In der Dokumentation (siehe ct.de/w66e) steht auch Beispielcode für die Anmeldung bei Facebook, Fitbit, GitHub und zahlreiche andere Dienste, die OAuth2 unterstützen.

Die Bibliothek startet den Anmeldevorgang und liefert für Konsolenanwendungen die URL, mit der man anschließend auf einem beliebigen Rechner mit grafischem Browser die Anmeldung durchspielen kann. An deren Ende steht ein Code, den man einfach wieder in die Konsole kopiert. `requests_oauthlib` ein speichert Zugangstoken intern und hält es auf Wunsch auch automatisch aktuell. Wer es in eine Datei schreibt, spart sich auch am nächsten Tag das Browserfenster.

OAuth2Session

Die Authentifizierung mit `requests_oauthlib` läuft komplett über ein `OAuth2Session`-Objekt. Nachdem man die Bibliothek mit `pip install requests_oauthlib` installiert hat, importiert man die Klasse mit:

```
from requests_oauthlib import OAuth2Session
```

Um eine `OAuth2Session` zu starten, möchte die aber drei Dinge wissen: Eine Client-ID der Anwendung, die API-Scopes, die definieren, auf welche Daten die Anwendung zugreifen möchte, und eine URL, zu der Google weiterleitet, sobald die Anmeldung abgeschlossen ist. Die Client-ID und die URL stehen in `client_id.json`, einer Datei, die man bei der Anmeldung der Anwendung in der Cloud-Developer-Console herunterladen kann (siehe Kasten auf Seite 42). Die API-Scopes können Sie beispielsweise aus unserem GitHub-Repository aus der Datei `google_fit_api_scopes.json` kopieren (siehe ct.de/w66e). Wir haben für unsere Anwendung alle verfügbaren Zugriffsrechte für die Fit-API angefragt. Sie können die Liste für die eigene Anwendung also einfach auf die Rechte zusammenkürzen, die Sie wirklich brauchen. Die Definitionen für andere APIs erhalten Sie über deren Dokumentation.

Als URL für die Weiterleitung nutzt unsere Python-Anwendung den String `"urn:ietf:wg:oauth:2.0:oob"`. Diese Pseudo-URL veranlasst Google, nach der Anmeldung das Token zum Herauskopieren anzuzeigen statt weiterzuleiten. Zusammen mit dem Laden der JSON-Daten aus den beiden Dateien sieht das Erzeugen der `OAuth2Session` dann so aus:

```
with open("client_id.json") as f:
    client_data = json.load(f)
```

Ist der Google-Account per Zwei-Faktor-Authentifizierung gesichert, fragt die Anmeldung auch den zweiten Faktor ab.

```
sc_file = "google_fit_api_scopes.json"
with open(sc_file) as f:
   scopes = json.load(f)
cdi = client_data['installed']
google_fit = OAuth2Session(
   cdi['client_id'], scope=scopes,
   redirect_uri=cdi['redirect_uris'][0])
```

Die so erzeugte Instanz der Session (`google_fit`) übernimmt fortan alle Netzwerkanfragen an das API. Die erste Anfrage muss darin bestehen, die URL zum Authentifizieren aufzurufen. Das macht die Methode `authorization_url()`:

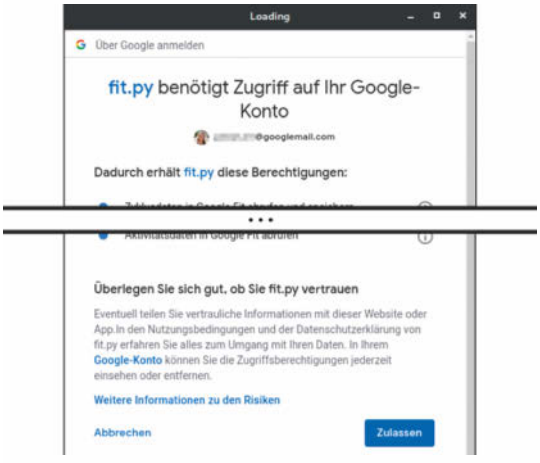
```
url, _ = google_fit.authorization_url(
 client_data['installed']['auth_uri'],
 access_type="offline",
 prompt="select_account")
authorization_url = url
```

Die Anwendung zeigt nun die URL an und fordert den Nutzer auf, sie in einem Browser zu öffnen.

Der Browser zeigt nun mehrere Seiten an: Existieren noch keine Google-Cookies, fragt er erst nach dem Nutzernamen (Gmail-Adresse) und auf einer neuen Seite nach dem Passwort. Hat man Zwei-Faktor-Authentifizierung aktiviert, gelangt man danach zu einer Seite dafür. Ist man dagegen im aktiven Browser bereits bei Google angemeldet, fragt der lediglich, ob man sich mit dem bereits angemeldeten Account anmelden möchte.

In beiden Fällen zeigt Google danach an, welche Datenzugriffe die Anwendung verlangt. Dem muss man zustimmen, um die Anmeldung abzuschließen.

Anschließend zeigt Google eine Seite mit dem Zugriffscode für den API-Zugriff. Den Code kopiert man nun vom Browserfenster ins Konsolenfenster. Der `approval_code` ist noch nicht das Token, mit dem man auf die API zugreift. Mit ihm bezieht man aber ein solches Token:



Um auf Daten aus Google Fit zuzugreifen, muss der Nutzer die Berechtigung dazu erteilen.

```
google_fit.fetch_token(client_data['installed'] ↵
  ↳['token_uri'],
   client_secret=client_data['installed'] ↵
   ↳['client_secret'],
   code=approval_code)
```

Die Funktion gibt ein Dictionary mit allen Informationen zurück, das eigentliche Token behält das `OAuth2Session`-Objekt `google_fit` auch intern.

Tokens erneuern

OAuth2-Tokens laufen standardmäßig nach einer Stunde ab. Wie lang sie genau gültig bleiben, steht als Zahl der Sekunden in `expires_in`. Der Timestamp, wann das Token abläuft, steht in `expires_at`. Man kann die Tokens ganz leicht erneuern:

```
google_fit.refresh_token(client_data ↵
                    ↳['installed']['token_uri'],
   client_id=client_data['installed'] ↵
                                  ↳['client_id'],
   client_secret=client_data['installed'] ↵
                                ↳['client_secret'])
```

Anders als viele andere OAuth-Provider verwendet Google als URL zum Erneuern die gleiche wie beim Abruf.

Wer nicht selbst erneuern will, erstellt direkt eine `OAuth2Session` mit automatischem Update:

```
google_fit = OAuth2Session(client_data ↵
                    ↳['installed']['client_id'],
  token=google_fit.token, auto_refresh_kwargs={
    'client_id': client_data['installed'] ↵
                                  ↳['client_id'],
    'client_secret': client_data['installed'] ↵
                                ↳['client_secret']},
   auto_refresh_url=client_data['installed'] ↵
                                  ↳['token_uri'],
   token_updater=token_update_callback)
```

Statt wie im Beispiel das Token aus einer bestehenden `OAuth2Session` zu holen, kann man es aus einer Datei laden. Falls es sich noch erneuern lässt, spart man sich dann den Anmeldevorgang im Browser. Ob das Token unbrauchbar ist, merkt man daran, dass ein API-Request einen `ValueError` wirft. In dem Fall muss man sich dann authentifizieren, als ob man noch kein Token hätte. (pmk) c't

Literatur

[1] Oliver Lau, **Gottvertrauen,** Benutzer von Web-Anwendungen mit Hilfe von OAuth 2.0 authentifizieren, c't 07/2014, S. 190

Beispielcode bei GitHub, Dokumentation:

www.ct.de/w66e

Bild: Albert Hulm

Auf das Google-Fit-API zugreifen

Google Fit ist nicht nur eine App, die automatisch sportliche Aktivitäten erkennt. Unter gleichem Namen bietet Google auch ein API zum Zugriff auf den dahinterliegenden Cloud-Service an. Mit dem API lassen sich zahlreiche Daten auslesen und schreiben, die die Google-App nicht anzeigt. Wir erklären, wie Sie mit Python an alle Daten kommen.

Von Pina Merkert

Google Fit speichert Workouts, Schritte, Gewicht, Herzfrequenz und vieles mehr im „Google Fitness Store“. Auf den können auch andere Apps, Webanwendungen und Programme über das Google-Fit-API zugreifen. Für Android hat Google den API-Zugriff über die Play-Services implementiert. Für alle anderen Plattformen bietet der Konzern ein REST-API an. Das API nutzen bereits viele Anwendungen und speichern dort auch Daten, die Googles Fitness-App gar nicht anzeigen kann, beispielsweise Ernährungsdaten. Wir wollten natürlich Zugriff auf alles und auch selbst Daten in Googles Fitness-Cloud ablegen.

Dafür muss man sich zunächst per OAuth2 bei Google anmelden (siehe S. 40). Google erlaubt das nur Apps, die Google kennt, sodass man die eigene Anwendung zuvor in der Cloud-Developer-Console anlegen muss (siehe Kasten). Bei der Anmeldung legt man auch fest, auf welche Daten die eigene Anwendung zugreift, und Google lässt diesen Datenzugriff auch vom Nutzer abnicken, bevor das API Daten liefert.

Um die Daten anzuzeigen, haben wir ein grafisches Programm mit PyQt5 geschrieben. Für die Anmeldung bei Google zeigt die eine URL an, die man in einem aktuellen Browser wie Firefox oder Chrome öffnet und dort die Anmeldung durchspielt. Am Ende des Prozesses steht dort ein Code, den man wieder in die Anwendung kopiert. Das funktioniert auch in der Konsole, weshalb die hier

gezeigten API-Zugriffe auch ohne Qt und GUI funktionieren.

Datenquellen

Eine Grundidee des Google-Fit-API besteht darin, dass alle Daten eine Quelle und einen Typ haben. Stammen Datenpunkte beispielsweise von einem Bluetooth-Sensor, der mit einer App auf dem Smartphone verbunden ist, legt die App eine Datenquelle für den Sensor in Verbindung mit dieser App an. Auf diese Weise registrieren Apps meist einige verschiedene Datenquellen bei Google Fit. Für jeden unterschiedlichen Sensor und Typ gibt es eine eigene Quelle. Auch aggregierte Daten wie die Summe der gelaufenen Schritte über einen Tag bekommen eine eigene Datenquelle.

Ohne besondere Freischaltung durch Google können Apps die Daten anderer Apps nicht löschen oder verändern, die meisten Daten aber lesen. Will eine Anwendung wie unser Python-Programm Daten editieren, muss es daher eigene Datenquellen registrieren und die veränderten Daten aus diesen Quellen höher priorisieren als die aus anderen Apps.

Außerdem gibt es unterschiedliche Typen von Daten. Google liefert einige Standardtypen wie Aktivitäten, Herzfrequenzen, Geschwindigkeiten, verbrauchte Kalorien, Ernährungsdaten, Größe, Gewicht oder Blutdruck. Zusätzlich können Anwendungen in ihren Datenquellen eigene Typen definieren. Diese sind aber zunächst privat, sodass nur diese Anwendung die Daten lesen kann. Sollen die Daten auch anderen Apps zur Verfügung stehen, können sich Entwickler bei Google melden und dort beantragen, dass ihre Datenquellen „shared“ werden und damit von allen Apps gelesen werden dürfen.

Will eine Anwendung wie unsere nicht nur die selbst gespeicherten Daten auslesen, muss sie zunächst die Liste aller Datenquellen abrufen. Das geht mit einem GET-Request an `https://www.googleapis.com/fitness/v1/users/me/dataSources`. Die URL für dieses API setzt sich aus der Adresse des API (`https://www.googleapis.com/fitness/`), einer Versionsnummer (`v1/`), der Angabe des Nutzers (momentan unterstützt Google ausschließlich `users/me/`) und dem Endpunkt (`dataSources`) zusammen. Die Anfragen an das REST-API nutzen alle dieses Schema und hängen lediglich noch weitere Informationen an die URL an.

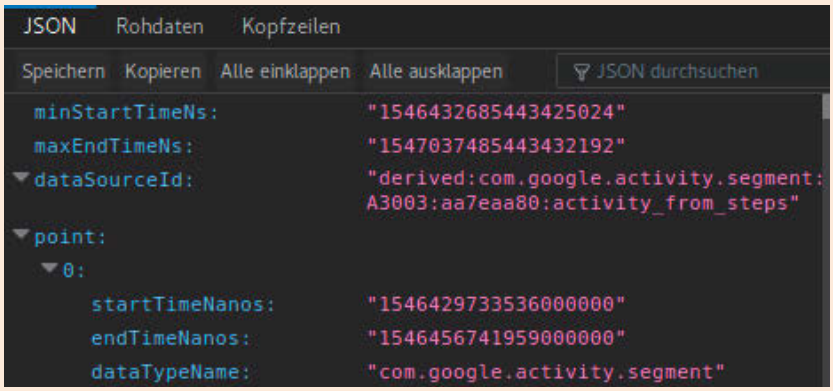
Das Google-Fit-API antwortet grundsätzlich mit Daten im JSON-Format. Für die Nutzung mit `requests_oauth` ist das praktisch, da die Bibliothek mit der `.json()`-Methode gleich den Inhalt parsen und als Python-`dict` oder -`list` zurückgeben kann. Das Abfragen aller Datenquellen geht daher in zwei Zeilen:

```
response = google_fit.get(
   "https://www.googleapis.com" +
   "/fitness/v1/users/me/dataSources")
data_sources = response.json()[
                        'dataSource']
```

In der Antwort steht bei jeder Datenquelle eine `"dataStreamId"`, die Sie verwirrenderweise beim Abfragen der Datensätze als `"dataSourceId"` angeben müssen.

Workouts laden

Mit der Liste der Datenquellen kann man nun alle Daten eines bestimmten Typs laden, beispielsweise `"com.google.activity.segment"`. Fitnessaktivitäten haben diesen Typ. Wie alle Daten in Google Fit ha-



Ausschnitt aus der Antwort des API, angezeigt in Firefox: In ‚points‘ stehen die Workouts mit Zeitraum und der Nummer der Aktivität in ‚value‘.

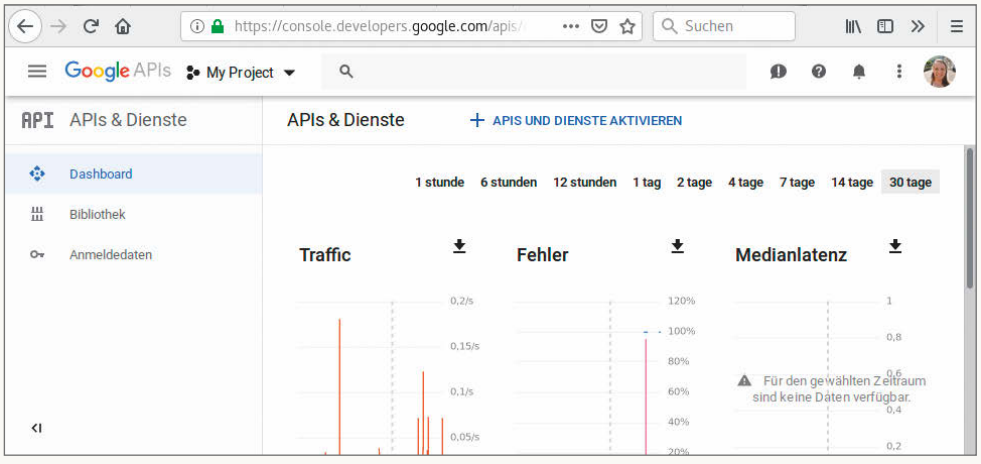
Registrieren eigener Anwendungen in der Cloud-Developer-Console

Damit Ihre eigene Anwendung auf Google-APIs zugreifen darf, müssen Sie für sie ein Projekt in Googles Cloud-Developer-Console anlegen. Die erreichen Sie über https://console.developers.google.com. Legen Sie zunächst ein neues Projekt für die Anwendung an, indem Sie links oben auf das Symbol mit den drei Sechsecken klicken und im erscheinenden Pop-up rechts oben „neues Projekt“ auswählen. Die Erstellung dauert einige Sekunden, danach können Sie das neue Projekt bei den drei Sechsecken auswählen.

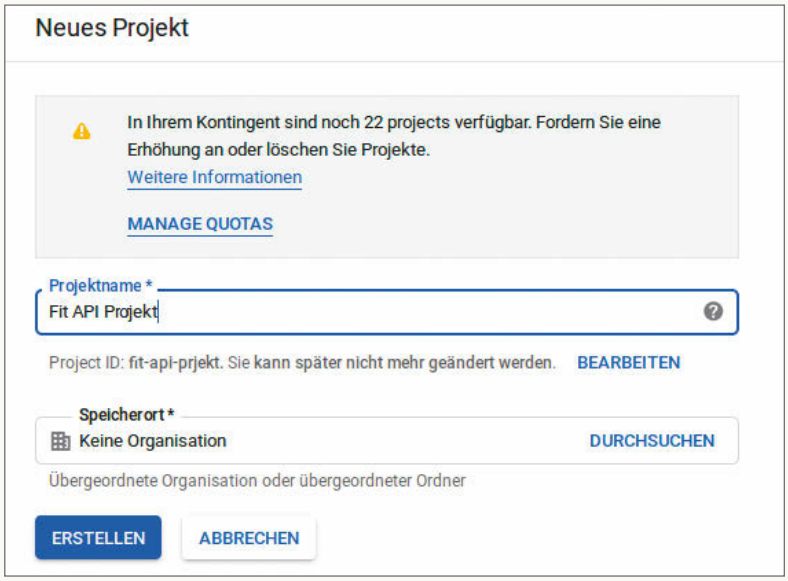
Danach landen Sie automatisch im Menüpunkt „APIs & Dienste“, wo Sie oben mittig „APIs und Dienste aktivieren“ auswählen. Es öffnet sich eine Übersichtsseite mit allen Google-APIs. Das Fit-API finden Sie am schnellsten, wenn Sie oben ins Suchfeld „Fit“ eintippen, dann nämlich erscheint sie als erstes Ergebnis. Nach einem Klick auf das Suchergebnis erscheint eine Beschreibungsseite mit einem „Aktivieren“-Button. Das Aktivieren dauert wieder ein paar Sekunden, woraufhin Google zu einer Übersichtsseite mit Statistiken weiterleitet.

Ein Kasten oben auf der Seite informiert, dass man vermutlich Anmeldedaten benötigt, bietet aber auch gleich einen Button, um sie zu erstellen. Von dort gelangen Sie in einen Wizard, wo Sie zunächst das „Fitness API“ auswählen. Danach geben Sie an, welche Art von Programm auf die Daten zugreift. In unserem Fall passt „Andere Benutzeroberfläche (z.B. Windows, CLI-Tool)“ am besten. Was Sie hier auswählen, bestimmt, welche URL für Redirects Google in die Daten schreibt. Unsere Auswahl sorgt für einen Anmeldeflow, der zuletzt einen Code anzeigt, statt auf eine tatsächliche URL weiterzuleiten. Zuletzt wählen Sie noch aus, dass Sie auch auf bestehende Nutzerdaten und nicht nur die Ihrer eigenen Anwendung zugreifen möchten. Ein Klick auf den Button „Welche Anmeldedaten brauche ich?“ führt zur nächsten Seite.

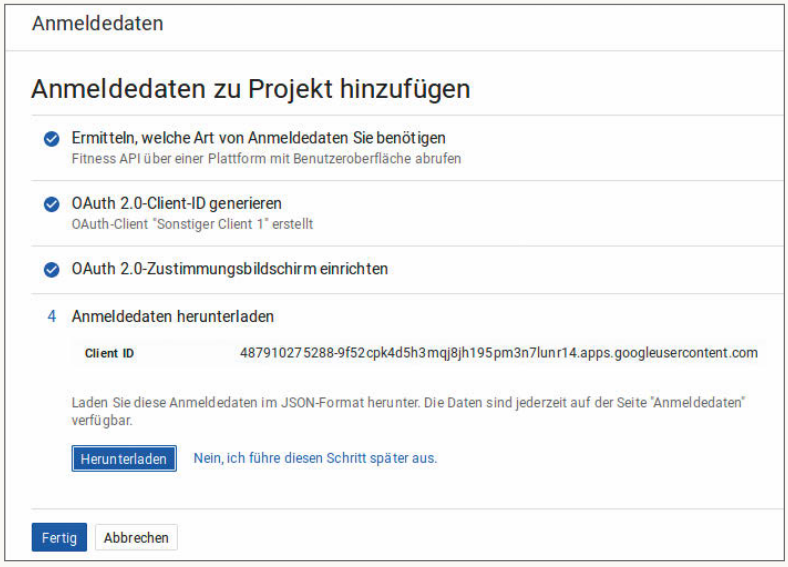
Im Schritt 2 legen Sie den Projektnamen fest und beim dritten Schritt geben Sie eine E-Mail-Adresse an und legen alle Infos fest, die der Nutzer bei der Anmeldung angezeigt bekommt. Beim Schritt 4 zeigt Google bereits die Client-ID. Statt die zu kopieren, empfiehlt es sich aber, einfach auf den „Herunterladen“-Button zu drücken und die von Google erzeugte JSON-Datei im Projektordner zu platzieren. Im Menüpunkt „Anmeldedaten“ lassen sich aber auch später alle Daten einsehen und abspeichern.



Die Cloud-Developer-Console begrüßt Sie zunächst mit einem Dashboard mit Statistiken. Den Button zum Anlegen neuer Projekte finden Sie oben links.



Für Ihre Anwendung sollten Sie zunächst ein neues Projekt anlegen.



Sobald die Anmeldedaten erstellt sind, können Sie diese als JSON-Datei herunterladen.

ben die Datensätze eine Startzeit und eine Endzeit (Timestamps in Nanosekunden) und ein `value`. Hinter dem JSON-Schlüssel `value` können sich mehrere Werte verbergen, die Datensätze geben aber nur die Art der Aktivität als Integer an. Welcher Name zu dieser Nummer gehört, listet die API-Doku auf (siehe ct.de/wmeh). Wir haben die Liste zur Verwendung im Programm als `dict` in die Datei `google_fit_activity_types.py` übertragen.

Jede Datenquelle nennt nicht nur den Typ der enthaltenen Daten, den wir für die Suche nach geeigneten Quellen verwendet haben, sondern auch stets eine `dataSourceId`. Die braucht man, um die Daten aus der Datenquelle abzufragen. Damit man dabei nicht immer die ganze Datenbank herunterladen muss, gibt man zusätzlich Startzeit und Endzeit in der URL an:

```
for source in self.data_sources:
  if (source['dataType']['name'] ==
      "com.google.activity.segment"):
    response = self.google_fit.get(
      "https://www.googleapis.com" +
      "/fitness/v1/users/me" +
      "/dataSources/" +
      source['dataStreamId'] +
      "/datasets/" +
      str(int((datetime.now(
         ) - timedelta(days=7)
        ).timestamp() * 1000000000))
      + "-" +
      str(int(datetime.now(
        ).timestamp() * 1000000000)))
```

Die Anfrage beginnt wie die zum Laden der Datenquellen, hängt aber die `"dataStreamId"` an. Danach legt man mit `datasets/` fest, dass man die Inhalte der Quelle laden möchte. Zuletzt kommen Start- und Endzeit, getrennt durch ein Minus. Die Zeitstempel haben wir ausgehend von Python-`datetime`-Objekten berechnet. Die geben mit der Methode `timestamp()` zwar einen Zeitstempel zurück, allerdings in Sekunden. Um daraus Nanosekunden zu machen, muss man die noch mit einer Milliarde multiplizieren.

Die Antwort besteht aus einer Angabe zur Datenquelle, dem Zeitbereich, in dem alle geladenen Aktivitäten liegen, und in `"point"`, einer Liste der einzelnen Aktivitäten. Interessant sind bei denen die Zeitstempel `"startTimeNanos"` und `"endTimeNanos"` und der bereits erwähnte Integer. Die Zeitstempel konvertiert man in `datetime`, indem man sie durch eine Milliarde teilt:

```
datetime.fromtimestamp(int(activity[
      'startTimeNanos']) / 1000000000
```

Andere Daten wie das Gewicht lädt man nach dem gleichen Schema und passt lediglich den Datentyp an:

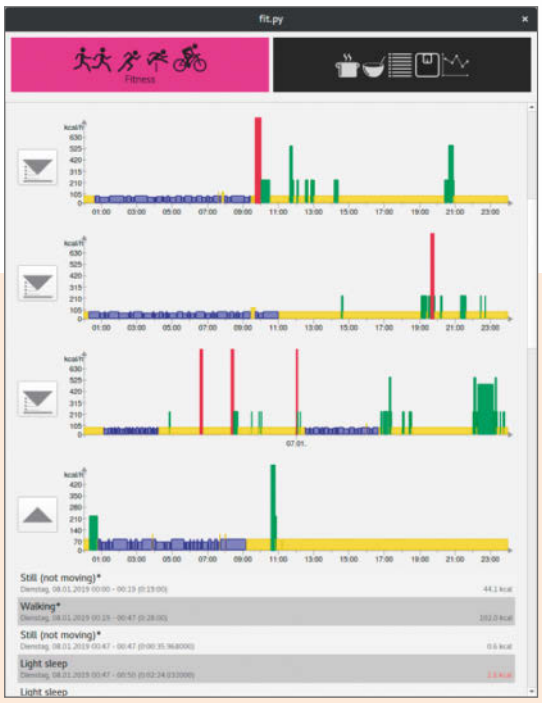
```
weights = []
for source in data_sources:
  if (source['dataType']['name'] ==
      "com.google.weight"):
    response = self.google_fit.get(
      "https://www.googleapis.com" +
      "/fitness/v1/users/me" +
      "/dataSources/" +
      source['dataStreamId'] +
      "/datasets/" +
      str(int((datetime.now() -
               self.time_window
        ).timestamp() * 1000000000))
      + "-" +
      str(int(datetime.now(
        ).timestamp() * 1000000000)))
    for activity in response.json()[
                              'point']:
      weights.append({
        "time":
          datetime.fromtimestamp(int(
            activity['startTimeNanos'])
            / 1000000000),
        "weight":
          activity['value'][0]['fpVal']
      })
```

Da das Gewicht als Gleitkommazahl vorliegt, muss man hier nur `fpVal` statt `intVal` aus dem ersten Eintrag in `value` auslesen.

Datenquellen anlegen

Wir wollten aber nicht nur Daten aus Google Fit auslesen, sondern auch eigene Daten hinzufügen. Wie erwähnt erlaubt Google unserer Anwendung nicht, die Daten anderer Apps zu verändern. Wir können aber eigene Datenquellen anlegen und in diese nach Herzenslust Daten schreiben. Die Nutzungsbedingungen verbieten, etwas anderes als Fitnessdaten zu speichern (was Google vermutlich nicht überprüft). Das Geburtsdatum wäre regelkonform. Es gibt bereits fertige Datentypen für Größe und Gewicht, nicht jedoch für das Alter. Da wir nach der Formel von Mifflin-St.Jeor (siehe ct.de/wmeh) den Grundumsatz berechnen wollten, brauchten wir auch das Alter. In die Formel fließen nämlich das Alter, Gewicht, Geschlecht und die Größe ein.

Das Anlegen von Datenquellen läuft über den gleichen API-Endpunkt wie das Auslesen von Datenquellen, jedoch als POST- statt als GET-Request. Die Anfrage muss eine Definition der Datenquelle im JSON-Format enthalten. Die enthält einen frei wählbaren Namen, einen Typ (`"raw"` oder `"derived"`), Angaben zur Anwendung, die ihn anlegt, eine ID und in `"dataType"` die Definition des neuen Daten-



Unser Qt-Programm ruft Fitnessdaten von Google Fit ab (hier Workouts) und bereitet sie grafisch ansprechend auf – beispielsweise als Balkendiagramm, bei dem höhere Balken für mehr verbrauchte Kalorien stehen.

typs. Der sollte einen verständlichen Namen haben und in "field" Name und Format aller enthaltenen Daten festlegen. Für das Geburtsdatum reicht ein einzelner Integer, da es die Beispielanwendung als Zeitstempel speichert. Die JSON-Struktur sieht als Ganzes dann so aus:

```
{
 "name": "fit.py-birthdate",
 "type": "raw",
 "dataType": {
   "name": "net.pinae.fit.birthdate",
   "field": [
     {"name": "birthdate",
      "format": "integer"}]},
  "application": {
    "name": "fit.py",
    "version": "1.0"},
  "dataStreamId":
  ":raw:fit.py-birthdate:987628404510"
}
```

Die "dataStreamId" setzt sich zusammen aus dem Typ, dem Namen und der Projektnummer aus der Developer-Console. Die steht vor dem Minus ganz am Anfang der Client-ID, die schon bei der Anmeldung zum Einsatz kam.

Daten schreiben

Daten schreibt man mit einem PATCH- statt einem GET-Request. In der URL gibt man vor /datasets/ die "dataStreamId" der gerade erstellten Datenquelle an. Auch an diese URL hängt man einen Zeitbereich an: Daten außerhalb dieses Bereichs tastet die API dann nicht an.

Die JSON-Daten dieses Requests muss man im richtigen Format zusammenbauen. Auf oberster Ebene gibt man zunächst mit "minStartTimeNs" und "maxEndTimeNs" noch mal den eben erwähnten Zeitbereich und mit "dataSourceId" die Datenquelle an. Die eigentlichen Daten gehören in eine Liste unter dem Schlüssel "point". In der stehen Datensätze, die mindestens eine "startTimeNanos", eine "endTimeNanos", einen "dataTypeName" und in einer Liste unter "value" die eigentlichen Daten angeben. Die eigentlichen Daten stehen hinter einem zum Typ passenden Schlüssel, bei Integern zum Beispiel hinter "intVal".

Um mit Python sofort einen Datensatz für ein Geburtsdatum zu speichern, entsteht damit diese Struktur:

```
now_nanos = int(
          datetime.now().timestamp() *
          1000000000)
birthday_data_source = {
  "minStartTimeNs": str(now_nanos),
  "maxEndTimeNs": str(now_nanos),
  "dataSourceId": birthdate_source_id,
  "point": [{
    "startTimeNanos": str(now_nanos),
    "endTimeNanos": str(now_nanos),
    "dataTypeName":
            "net.pinae.fit.birthdate",
    "value": [{
      "intVal": str(int(bday.timestamp()))
    }]
  }]
}
```

Bei Erfolg antwortet das API mit den abgespeicherten Daten, gibt also die gleiche Struktur zurück.

Da sich dieser Request ja nur auf den jetzigen Zeitpunkt bezieht, sammeln sich mehrere Datensätze in Google Fit. Zuvor gespeicherte Daten überschreibt der Fitness Store dabei nicht. Alte Daten bleiben aber auch dann bestehen, wenn man den Zeitraum auf oberster Ebene so erweitert, dass er die alten Datensätze umfasst. Google verhindert damit, dass man PATCH als DELETE missbraucht. Fürs Löschen gibt es eigene API-Aufrufe (siehe Doku unter ct.de/wmeh).

Fit mit Python

Ob Sie Ihre Fitness mit Googles API und einem eigenen Python-Programm verbessern können, hängt von ihrer Kreativität ab. Unser grafisches Qt-Programm zeichnet mit den Daten aus Google Fit Diagramme und zeigt viele Daten, über die man mit Googles App nicht herankommt. Sie können das Programm als Vorlage für eigene Fitness-Helfer verwenden, wofür Sie nach dem Fork erst mal in der Developer-Console eigene Schlüssel für Ihre Anwendung erstellen müssen. Wir freuen uns aber auch über Pull-Requests, um unser Programm zu erweitern und zu verbessern.

Um tatsächlich fitter zu werden, müssen Sie am Ende aber immer noch den inneren Schweinehund überwinden und Sport treiben. Aber mit Google Fit wissen Sie danach wenigstens auf Nanosekunden genau, wie lang Sie trainiert haben. Im Idealfall motiviert Sie das genug, um die Sport-Ziele von Neujahr umzusetzen. (pmk) c't

Beispielcode bei GitHub, Dokumentation:
www.ct.de/wmeh

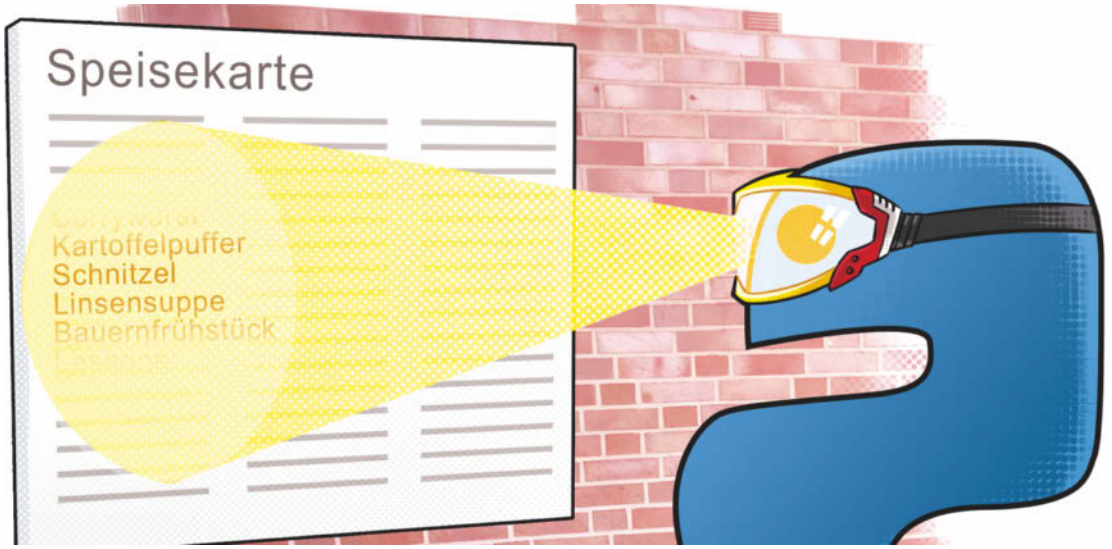


Bild: Albert Hulm

Texterkennung mit Tesseract

Die Python-Bibliothek pytesseract erkennt Text in Grafiken und liest diesen aus. Wir haben ein Programm geschrieben, das aus einer PDF-Datei den Essensplan der Kantine liest.

Von Otis Sotek

Mit OCR (Optical character recognition) können Programme Text aus Bildern lesen und dann als Zeichenkette weiterverarbeiten. Das ist beispielsweise nützlich, um gescannte Briefe maschinell zu durchsuchen oder in Bildern Kennzeichen zu erkennen. Unter der Haube von modernen OCR-Engines steckt meist künstliche Intelligenz. Die hat an tausenden von Beispielen gelernt, wie Zeichen und Schriften aussehen und welchen Text sie darstellen.

In diesem Beispiel soll die OCR-Bibliothek tesseract von Google den Speiseplan der Heise-Kantine im PDF-Format in Python auslesen und das Angebot des Tages bereitstellen.

Es mag übertrieben klingen, Bilderkennung auf ein PDF anzuwenden, wenn es Programme wie pdftotext gibt, die eingebetteten Text aus einem PDF von vorne bis hinten ausgeben. Bei einer Tabelle, die aus Excel exportiert wurde, wie dem Speiseplan, klappt das aber nicht. Zudem liest pdftotext nur Dateien, in denen der Text auch als Text eingebettet ist. Liegt der Plan als Bild vor, wird OCR benötigt.

Für die Bilder vom Speiseplan spannt das Programm neben pytesseract die Bibliotheken pdf2image und PIL („Python Imaging Library“) ein. Pdf2image kann Seiten aus PDF-Dokumenten in Bilder konvertieren, die im Programmcode als PIL-

Objekte bereitstehen. PIL ist eine Bibliothek zum Verarbeiten von Bildern in Python.

Vom PDF zum Bild

Der Speiseplan steht im internen Netz als PDF zum Download bereit. Diesen Plan soll pytesseract lesen, aber die Bibliothek akzeptiert kein PDF, sondern nur Bilder. Die Brücke vom PDF zum Bild schlägt pdf2image:

```
from pdf2image import convert_from_path
image = convert_from_path("plan.pdf")[0]
image.save("plan.png", "png")
```

`convert_from_path` liest ein PDF vom Dateisystem ein und konvertiert es in eine Liste von Bildern, eins für jede Seite des PDF. Der Speiseplan besteht nur aus einer Seite, also kommt die erste Seite (`[0]`) in die Variable `image`. Das Bild wird dann unter dem Namen `plan.png` abgespeichert. Das ist für die Funktion des Programms nicht notwendig, aber dadurch können Sie verfolgen, wie das ausgelesene Bild aussieht, falls es zu Fehlern kommt. Dieses Bild wandelt pytesseract in Text um:

```
import pytesseract
txt = pytesseract.image_to_string(image)
print(txt)
```

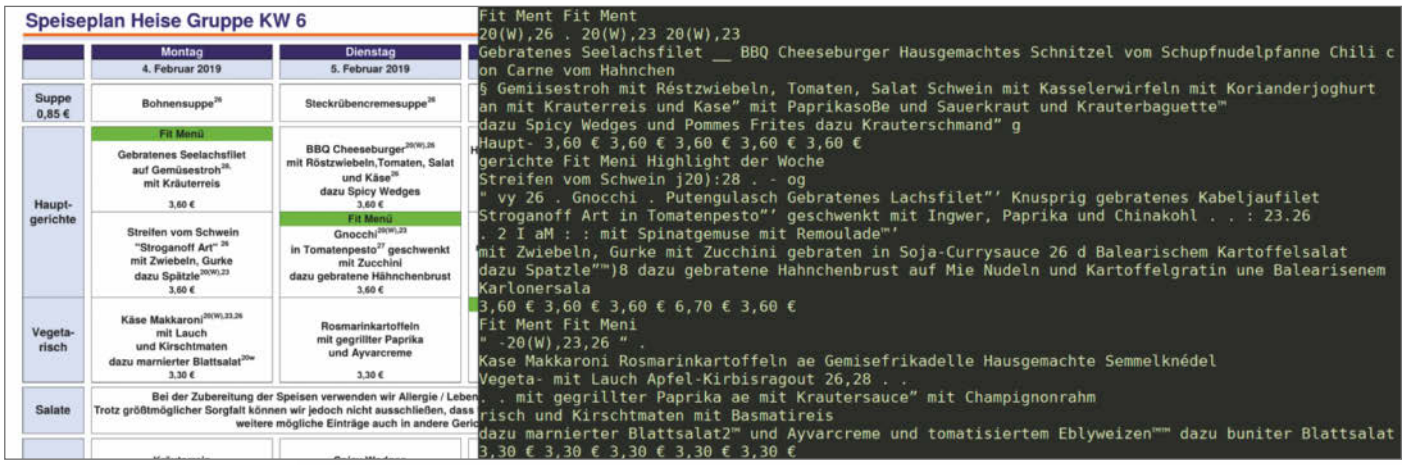
Wer lediglich Briefe oder sonstige Fließtexte auslesen will, ist hier fertig. Pytesseract hat das Bild von rechts nach links und oben nach unten gelesen. Bei einer Tabelle wie unserem Speiseplan funktioniert das nicht, denn sämtliche Titel, Beschriftungen oder Logos liest die Bibliothek mit ein. Daher kommt zunächst nur unsinniger Text heraus.

Die Schere ansetzen

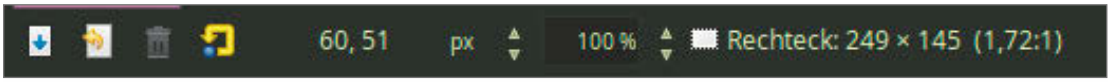
Der Speiseplan für eine Woche ist eine immer gleich aufgebaute Tabelle: Eine Spalte entspricht einem Tag, es gibt also fünf Spalten für Montag bis Freitag. Alle Spalten sind gleich breit. Die Zeilen entsprechen den einzelnen Kategorien, also zum Beispiel Suppen, Hauptgerichten und Beilagen. Die Höhen der Zeilen ändern sich nicht, allerdings sind die Zeilen für verschiedene Gerichte unterschiedlich hoch.

Ziel ist, den Speiseplan täglich vor dem Mittagessen auszugeben. Das bedeutet, jede Zelle einer Spalte isoliert auszulesen und den Text in eine Liste einzutragen.

Dazu muss pytesseract jede Zelle als einzelnes Bild übergeben bekommen. Es braucht eine Methode, das große Bild des gesamten Speiseplans in einzelne Bilder aufzuteilen, um diese dann an pytesseract weiterzugeben. Dafür kann das Bild entweder zuerst in Spalten und dann in Zellen oder zuerst in Zeilen und dann in Zellen zerlegt werden. In unserem Fall ist es sinnvoller, zuerst Zeilen zu bilden und diese dann in fünf gleiche Teile aufzuteilen, als die unterschiedlichen Zeilenhöhen auf jede Spalte erneut anwenden zu müssen.



Die Ausgabe von pytesseract ist völlig durcheinander und auch nicht immer richtig.



Die Statusleiste von GIMP zeigt die Position des Mauszeigers auf der Leinwand und die Größe der Auswahl auf einen Blick.

Natürlich muss das Programm die Position und Größe der einzelnen Zeilen kennen. Leicht herausfinden kann man diese mithilfe eines Bildbearbeitungsprogramms wie GIMP, denn GIMP verwendet das gleiche Koordinatensystem wie PIL: Der Ursprung befindet sich oben links, die x-Koordinaten wachsen nach rechts und die y-Koordinaten nach unten. Dazu öffnet man das Bild `plan.png` aus dem ersten Schritt. Mithilfe des rechteckigen Auswahlwerkzeugs markiert man eine Zeile und bekommt dabei in der unteren Statusleiste Position und Dimension der Auswahl angezeigt. Will man sichergehen, dass diese Werte für sämtliche Bildgrößen funktionieren, sind prozentuale Angaben statt absoluter Pixelzahlen sinnvoll. Ein Beispiel: Für das zweite vertikale Viertel des Bildes legt man die Box `(0, 0.25, 1, 0.5)` fest.

Das Programm funktioniert so auch dann noch, wenn der Speiseplan versehentlich statt in DIN A4 in DIN A3 erstellt wird. Der Aufruf `image.save()` kann jetzt entfernt werden, denn die Bilddatei wird im eigentlichen Programm nicht benötigt. Für die erste Tabellenzeile, in der die Tagessuppe eingetragen ist, gelten folgende Konstanten:

```
# Position und Größe der Zeile
# im Verhältnis zum Plan
soup_x = 0.102
soup_y = 0.21
soup_w = 0.86
soup_h = 0.05
```

Das Programm schneidet nun die Bilder auf einzelne Zeilen und schließlich Zellen zu. Da das besonders viele Schnittvorgänge sind, erleichtert eine eigene Funktion `crop()` die Arbeit:

```
def crop(image, box):
  w, h = image.size
  x = int(box[0]*w)
  y = int(box[1]*h)
  cropped_w = int(box[2]*w)
  cropped_h = int(box[3]*h)
  return image.crop((x, y,
    x + cropped_w, y + cropped_h))
```

An dieser Stelle müssen auch die prozentualen Angaben wieder in Pixelangaben umgerechnet werden. Dazu multipliziert man sie mit der Breite beziehungsweise Höhe des Bildes. Diese sind im Tupel `size` gespeichert. PIL schneidet Bilder mit der Methode `crop()` zu. Die Koordinaten und Dimensionen wandelt die Funktion in Integer um, denn es gibt keine halben Pixel.

In Kombination mit den Konstanten für die Suppen-Zeile kann das Programm jetzt einen langen String mit allen Suppen der Woche ausgeben:

```
soup = crop(plan,
           (soup_x, soup_y,
            soup_h, soup_w))
txt = pytesseract.image_to_string(
                              soup)
print(txt)
```

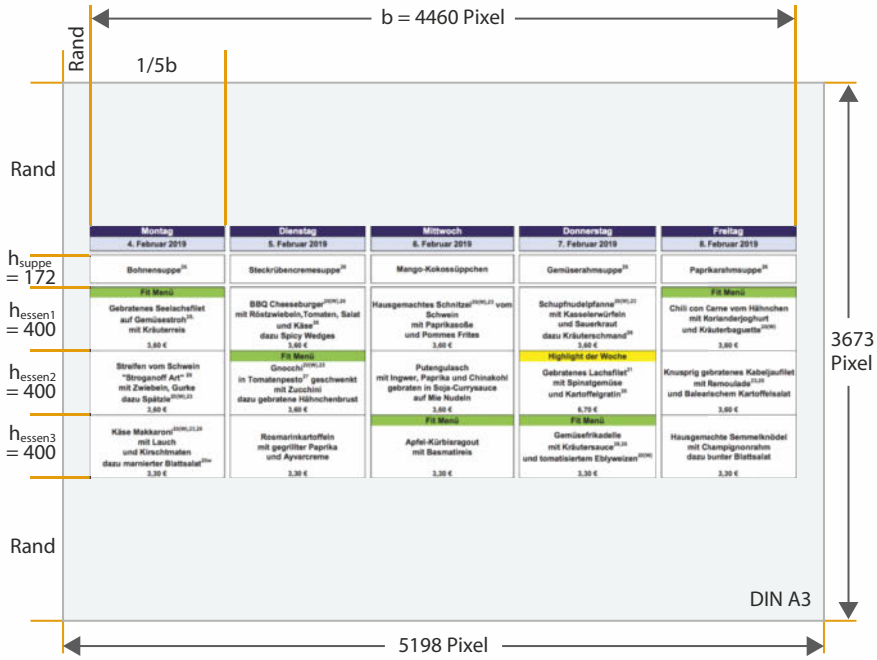
Das Programm gibt zwar noch eine Menge Kauderwelsch aus, aber die Zeilen- und Spaltenbeschriftungen sind rausgeschnitten. Um die falschen Zeichen kümmern wir uns später.

Als Nächstes müssen die Suppen als einzelne Strings vorliegen, sodass der Code sie einer Liste zuordnen kann. Die Suppen aufzutrennen ist aber einfach, da jedes Feld genau ein Fünftel der Zeile einnimmt. Eine weitere Funktion gibt eine Liste von fünf Teilen eines Bildes zurück, jeweils mit der vollen Höhe, aber immer um ein Fünftel (`0.2`) des gesamten Ausschnitts verschoben. Die Funktion `fifths()` verwendet wieder die prozentualen Angaben – die Berechnung der eigentlichen Pixel übernimmt `crop()`:

```
def fifths(image):
  fifths = []
  for i in range(0, 5):
    #      x      y  w    h
    box = (i*0.2, 0, 0.2, 1)
```

Aufbau des Speiseplans

Das Programm rechnet mit relativen Angaben. So kann der Plan in A4 oder A3 ausgewertet werden.




```
    fifth = crop(image, box)
    fifths.append(fifth)
  return fifths
```

Nun soll das Programm jedes dieser fünf Einzelbilder lesen und den erkannten Text in die Konsole schreiben:

```
soup = crop(plan,
            (soup_x, soup_y,
             soup_h, soup_w))
for f in fifths(soup):
  t = pytesseract.image_to_string(
                              soup)
  print(t)
```

Die Ausgabe frisieren

Es ist Zeit für eine neue Funktion, damit das bisherige Prozedere leichter fällt. So ist das Programm besser um Hauptgerichte und Beilagen erweiterbar und es fallen weniger for-Schleifen an. Diese Funktion muss zuerst eine zugeschnittene Zeile entgegennehmen, aufteilen und dann eine Liste mit allen Gerichten zurückgeben:

```
def parse(image):
  food = []
  for f in fifths(image):
    t = pytesseract.image_to_string(f)
    food.append(t)
  return food
soup = crop(plan,
            (soup_x, soup_y,
             soup_h, soup_w))
main1 = crop(plan,
             (main1_x, main1_y,
              main1_h, main1_w))
# ... und weitere Speisen! ...
print(str(parse(soup)))
print(str(parse(main1)))
# ...
```

```
otis@ct:~/Dokumente/essen$ python3 ocr4.py
Bohnensuppe" Steckrübencremesuppe"° Mango-Kokossüppchen Gemüserahmsuppe" Paprikarahmsuppe"
otis@ct:~/Dokumente/essen$
```

tesseract sieht hier zwar noch einige rätselhafte Zeichen, aber die Ausgabe beschränkt sich auf die Suppen.

Im Moment liegt jede Speise als String in einer Liste vor. Es gibt aber noch einige Probleme: Zum einen enthält der Text manchmal Sonderzeichen, die pytesseract in hochgestellten Zahlen, die für Allergene stehen gefunden hat. Zum anderen erkennt pytesseract viele Wörter nicht richtig. Der Grund dafür ist, dass die Bibliothek standardmäßig nach englischen Wörtern sucht, also zum Beispiel keine Umlaute erwartet.

Die Sprache, in der pytesseract liest, ist im Quellcode schnell mit `lang = "deu"` auf Deutsch umgestellt:

```
pytesseract.image_to_string(f,
                            lang="deu")
```

Unter Umständen müssen Sie noch Sprachpakete installieren. Unter Ubuntu führt man dazu den Befehl `sudo apt-get install tesseract-ocr-deu` aus, für andere Betriebssysteme gibt es Installationsanleitungen im Wiki der GitHub-Seite von tesseract. Den Link zum tesseract-Wiki sowie alle weiteren Links und Downloads zum Artikel finden Sie unter ct.de/w53k.

Als Nächstes soll die Funktion `parse()` die überflüssigen Zeichen unter Verwendung von regulären Ausdrücken entfernen. Übrig bleiben Buchstaben, Zahlen, Interpunktion, Leerzeichen und das Euro-Zeichen.

Dieser reguläre Ausdruck filtert alle Zeichen heraus, die in der Ausgabe nichts zu suchen haben:

```
import re
illegal_char_re = re.compile(
  "[^a-zA-Z0-9äöüÄÖÜß€,\.\- ]")
```

Die eckigen Klammern definieren ein Set an Zeichen, die der reguläre Ausdruck findet. `\.` und `\-` sind Escape-Sequenzen, damit der Punkt und das Minus, die besondere Funktionen haben, als echte Zeichen gelten. Das `^` am Anfang negiert das gesamte Set, nun findet der Ausdruck sämtliche Zeichen, die nicht zwischen den Klammern stehen. Diese Funktion ersetzt nun alle diese Zeichen durch einen leeren String, das heißt, sie löscht sie:

```
def prettify(t):
  t = re.sub(illegal_char_re, "", t)
  return t
```

Es lohnt sich, diese String-Operationen in eine separate Methode auszulagern. Das sorgt für bessere Erweiterbarkeit. Das Programm für den Speiseplan ersetzt zum Beispiel noch doppelte Leerzeichen und Zeilenumbrüche durch einfache:

```
linebreak_re = re.compile("\n+")
double_space_re = re.compile("\s+")
```

Am Beispiel des Speiseplans sind ein Telegram-Bot, der über das Angebot des nächsten Tages berichtet, eine automatische Suche nach dem eigenen Lieblingsessen oder das Filtern nach Nahrungsmittelunverträglichkeiten vorstellbar. (jam) c't

```
otis@ct:~/Dokumente/essen$ python3 ocr3.py
Suppe: Paprikarahmsuppe für 0,85 €
Hauptmenüs:
        Fit Menü Chili con Carne vom Hähnchen mit Korianderjoghurt und Kräuterbaguette für 3,60 €
                oder
         Knusprig gebratenes Kabeljaufilet mit Remoulade und Balearischem Kartoffelsalat für 3,60 €
                oder
         Hausgemachte Semmelknödel mit Champignonrahm dazu bunter Blattsalat für 3,30 € (vegetarisch)
Beilagen für 0,70 €:
       Balearischer Kartoffelsalat, Kräuterbaguette, Salatbeilage, Tagesgemüse
otis@ct:~/Dokumente/essen$
```

So könnte die durch prettify() verschönerte Version der Ausgabe in der Konsole aussehen.

Downloads und Links:

ct.de/w53k

in Business, Web & DevOps

DIE NEUE KONFERENZ FÜR PYTHON

Verschoben auf November 2020

enterPy

23. – 24. November 2020

Print Media Academy, Heidelberg

iX MAGAZIN FÜR PROFESSIONELLE INFORMATIONSTECHNIK

heise Developer

dpunkt.verlag

Python statt Bash

Aus Python heraus ruft man auch ohne große Mühe Programme auf, die in anderen Sprachen geschrieben wurden. Für typische Programmieraufgaben brauchen Python-Programme aber wesentlich weniger Code als Bash-Skripte. Daher eignet sich Python, um mit eleganterem Code Bash-Skripte zu ersetzen.

Von Pina Merkert

Zum Testen neuer Nvidia-Grafikkarten (siehe c't 09/2018, S. 92) führen wir in der c't-Redaktion unter anderem den Machine-Learning-Benchmark DeepBench aus. Der in C programmierte Benchmark besteht aus drei ausführbaren Einzeltests, die je zwei Parameter akzeptieren und jeweils eine ganze Batterie an Zeitmessungen ausspucken, beispielsweise für Matrixmultiplikationen mit verschieden großen Matrizen. Um verfälschte Ergebnisse durch zufällig dazwischengrätschende Systemprozesse zu vermeiden, führen wir jeden Benchmark dreimal aus und verwenden von jedem Einzelwert das beste Ergebnis. Diese Ergebnisse dampfen wir mit dem geometrischen Mittel auf einen einzelnen Wert pro Test ein, um ein Gesamtergebnis für die Performance einer Grafikkarte bei dieser Art von Berechnung zu erhalten. Man könnte die dafür nötigen 45 Tests per Hand starten, die Ergebnisse in Excel kopieren und dort auswerten. Mit einem Python-Skript macht der Testrechner das aber automatisch. Unser Beispiel zeigt, wie leicht Python Bash-Skripte und Batch-Dateien ersetzt.

Ein Bash-Skript könnte mit ein paar Schleifen relativ leicht die 45 Einzeltests starten. Beim Auswerten der Ergebnisse müsste man aber einige recht magische sed-Befehle erfinden, um aus den Konsolenausgaben der Benchmarks die Zahlenwerte der Ergebnisse herauszupicken. Spätestens beim Berechnen des geometrischen Mittels wird man sich eine Bibliothek wie NumPy wünschen, die es in so angenehmer Form nicht für die Bash gibt.

Python führt die Tests mit ähnlich wenig Code wie ein Bash-Skript aus, extrahiert danach aber relativ leicht mit regulären Ausdrücken die Ergebnisse. Möglich machen das die Module `subprocess`, `re` und `numpy`. Das erste ruft beliebige Programme als eigene Prozesse auf. Die anderen extrahieren und verarbeiten die Daten.

Mit `subprocess.Popen()` starten Sie aus Python-Skripten alle Programme, die Sie auch in der Konsole starten können. Als Parameter erwartet der Befehl eine Liste. Die enthält als ersten Eintrag den Namen des Programms und als weitere Einträge die Parameter. Aus `../DeepBench/code/bin/conv_bench train float` wird dann:

```
prc = subprocess.Popen([
   "../DeepBench/code/bin/conv_bench",
   "train", "float"])
```

Gibt man `Popen()` noch den Parameter `stdout=subprocess.PIPE` mit, leitet der Prozess seine Ausgabe in eine Pipe um. An die kommt man mit `communicate()`:

```
out = prc.communicate()[0]
```

In `out` stehen danach sämtliche Ausgaben, die das aufgerufene Programm normalerweise auf die Konsole geschrieben hätte.

Ausgaben auswerten

DeepBench gibt seine Ergebnisse als mit Leerzeichen formatierte Tabelle aus – ein zum Weiterverarbeiten ungünstiges Format. Aber immerhin sieht die Ausgabe immer gleich aus. Deswegen konnten wir die Ergebnisse mit regulären Ausdrücken filtern. Das übernimmt Pythons `re`-Modul. Unser regulärer Ausdruck erwartet Ganzzahlen `\d+`, die durch mindestens ein Leerzeichen `\s+` getrennt werden. Da wir die Zahlen weiterverarbeiten möchten, definieren wir eine Capturing-Group, indem wir die Zahlen in runde Klammern einpacken (`\d+`). `re.compile()` erzeugt aus dem String ein Regular-Expression-Objekt. Führt man mit dem die Methode `match()` aus und übergibt je eine Zeile der Benchmark-Ausgaben, entsteht dabei ein `Match`-Objekt. An die vom regulären Ausdruck definierten Gruppen (das sind die Teile des Ausdrucks, die in runden Klammern stehen) kommt man mit `match.groups()`. Etwas verkürzt sieht der Code dafür so aus:

```
res_line = ⤶
    ↳re.compile(r"^\s*(\d+)\s+(\d+)\s+(\w+)\s*$")
for line in out.splitlines():
    match = res_line.match(line)
    if match:
        p, fw_time, f_alg = match.groups()
```

Matrixrechnen

Wir haben unsere per `re` extrahierten Benchmarkergebnisse einfach in eine Liste `results` geschrieben. Jeder Eintrag der Liste besteht seinerseits aus einer Liste aus drei Werten, nämlich dem Ergebnis je eines Benchmarkdurchlaufs. Aus der Listen-Liste macht `np.array(results)` eine Matrix. Mit der kann NumPy nun bequem rechnen. Beispielsweise sucht `min(axis=0)` den kleinsten Wert in jeder Spalte der Matrix, sodass dabei ein Vektor entsteht:

```
np.array(results).min(axis=0)
```

Danach stehen in diesem Vektor die Laufzeiten des besten aus drei Benchmarkdurchläufen für jeden Einzeltest.

Nun wollten wir aber lieber einen einzelnen Wert statt eines ganzen Vektors angeben. Diesen Wert berechnen wir als geometrisches Mittel über die Ergebnisse der Einzeltests beziehungsweise Zahlen im Vektor. Eine Funktion zum Berechnen des geometrischen Mittels findet sich im Modul `scipy.stats` unter dem Namen `gmean`. SciPy stammt von den gleichen Entwicklern wie NumPy und enthält Hunderte von Funktionen für wissenschaftliche Berechnungen. Die Berechnung des Gesamtergebnisses passt so in eine Zeile:

```
gmean(np.array(results).min(axis=0))
```

Nie wieder Handarbeit

Python statt Bash lohnt sich gerade dann, wenn man nicht nur mit `subprocess.Popen()` Programme aufruft, sondern deren Ausgaben auch weiterverarbeiten will. Diese Berechnungen definiert man in Python in wesentlich weniger Zeilen als auf der Konsole.

Das gesamte Beispiel zum Auswerten von DeepBench finden Sie unter ct.de/wyu8. Es enthält einige zusätzliche Codezeilen, um verschieden formatierte Ausgaben der einzelnen Tests jeweils korrekt auszuwerten. Alle wesentlichen Ideen des Skripts haben Sie aber soeben kennengelernt. (pmk) c't

Quellcode:

www.ct.de/wyu8

```
cttest@TitanV-ct: ~/Code/DeepBench/code/bin
cttest@TitanV-ct:~/Code/DeepBench/code/bin$ ./gemm_bench train float
                    Running training benchmark
                              Times
--------------------------------------------------------------------------------
    m       n       k       a_t     b_t     precision       time (usec)
   1760      16    1760       0       0       float                116
   1760      32    1760       0       0       float                 57
   1760      64    1760       0       0       float                111
   1760     128    1760       0       0       float                160
   1760    7000    1760       0       0       float               6141
   2048      16    2048       0       0       float                153
   2048      32    2048       0       0       float                 89
   2048      64    2048       0       0       float                118
   2048     128    2048       0       0       float                189
   2048    7000    2048       0       0       float               7757
   2560      16    2560       0       0       float                227
   2560      32    2560       0       0       float                134
   2560      64    2560       0       0       float                171
   2560     128    2560       0       0       float                286
   2560    7000    2560       0       0       float              12810
   4096      16    4096       0       0       float                997
   4096      32    4096       0       0       float                309
```

DeepBench gibt zahlreiche Zeitmessungen als Tabelle formatiert aus. Das Python-Skript wertet die Daten automatisiert aus.

PDFs aus Code

Ein paar Zeilen Python mit der Bibliothek fPDF erzeugen automatisch druckbare Dokumente. Liegen beispielsweise Kochrezepte als strukturierte Daten vor, entstehen so im Handumdrehen hübsche Ausdrucke für die Küche.

Von Pina Merkert

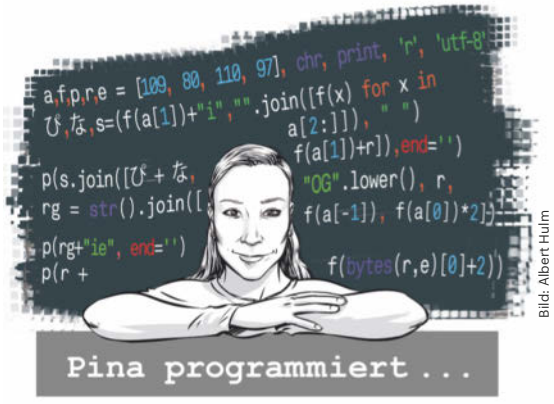


Bild: Albert Hulm

PDFs eignen sich hervorragend für Belege, Rechnungen und andere Dokumente, die als Kopie auf Papier im Aktenordner im Regal landen sollen. Programme wie LibreOffice exportieren zwar ordentliche PDFs, aber bevor ich mich mit den Tücken von datenbankgetriebenen Serienbriefen auseinandersetze, programmiere ich die PDFs lieber mit Python.

Im konkreten Beispiel habe ich einige Rezepte für persische und indische Köstlichkeiten als strukturierte Datensätze im JSON-Format gesammelt (`recipes.json` im GitHub-Repository, zu finden über ct.de/wp82). Die Rezepte haben einen Titel, geben die Portionsgröße an, listen die Zutaten auf und beschreiben die Zubereitung als Liste von Absätzen. Die Bibliothek fPDF und eine Handvoll Zeilen Python-Code verwandeln das wenig ansehnliche JSON in ein hübsches Dokument mit einem Rezept pro Seite. Um eine dieser Seiten sofort sehen zu können, habe ich den Python-Code direkt in einem Jupyter-Notebook programmiert (siehe Seite 154).

Dokument programmiert

Die Arbeit mit fPDF beginnt damit, ein `FPDF`-Objekt für das Dokument zu erzeugen. Hoch- oder Querformat, Seitengröße und die verwendeten Einheiten legt man am besten direkt fest:

```
from fpdf import FPDF
pdf = FPDF(orientation='P', unit='mm',
           format='A4')
```

Methoden von fPDF-Objekten

Name	Nutzen
`add_page()`	neue Seite
`set_font()`	Schriftart und -größe
`write()`	Absatz mit Text
`cell()`	Textblock an einer bestimmten Stelle
`image()`	Bild
`line()`	gerade Linien

Das pdf-Objekt stellt nun Methoden bereit, mit denen man das Dokument mit Inhalten befüllt. Die wichtigsten finden Sie in der kleinen Tabelle, den Rest in der Doku über ct.de/wp82.

Exotische Schriften

fPDF bringt zwar praktische Aliase für die Standardschriften `Courier`, `Helvetica` beziehungsweise `Arial`, `Times`, `Symbol` und `ZapfDingbats` mit, ich wollte für meine Rezepte aber eine verschnörkelte Fantasy-Schriftart nutzen (`Titans2.ttf` im Repository). Damit fPDF die findet, muss man im Paket die Variable `SYSTEM_TTFONTS` setzen. Man kann dafür das Schriftarten-Verzeichnis des Users angeben (unter Linux üblicherweise `~/.local/share/fonts/`), das des Systems (unter Windows zumeist `C:\Windows\Fonts`) oder auch das aktuelle Verzeichnis:

```
import fpdf
fpdf.SYSTEM_TTFONTS = '.'
pdf.add_font('Titans2', '',
             'Titans2.ttf', uni=True)
```

Die Funktion `pdf.add_font()` sorgt dafür, dass die neue Schrift im Dokument einen eindeutigen Namen bekommt (erster Parameter). Der zweite Parameter legt den Schriftstil (`B`: Bold, `I`: Italic, `BI`: BoldItalic) fest. Ein leerer String steht hier für „Regular“, also normale Schrift. Danach kommt der Dateiname. `uni=True` besagt, dass die Schrift Unicode unterstützt.

Eine so registrierte Schrift nutzt man anschließend wie die Standardschriften mit `set_font()`:

```
pdf.set_font('Titans2', size=12)
```

Hintergrundbild

Um den Fantasy-Eindruck zu unterstreichen, fügt je ein `image()` auf allen Seiten ein Hintergrundbild von altem Pergament ein (Dank an Fotograf Caleb Kimbrough für das Bild mit Creative-Commons-Lizenz):

Vorschau im Jupyter-Notebook

Um im Jupyter-Notebook sofort eine Vorschau einer Seite des PDFs zu sehen, muss man eine zweite Bibliothek einspannen: Wand. Jupyter-Notebooks können nämlich keine PDFs anzeigen, sondern nur Bilder. Wand konvertiert Seiten aus PDFs mithilfe von ImageMagick in Bilder.

Genau das verbietet die Standardkonfiguration von ImageMagic aber bei vielen Linux-Distributionen. Um das Verbot aufzuheben, fügen Sie in `/etc/ImageMagick-6/policy.xml` oder `/etc/ImageMagick-7/policy.xml` (je nach installierter Version) folgende Zeile ein:

```
<policy domain="coder"
        rights="read|write"
        pattern="PDF" />
```

Manche Distributionen wie Arch verbieten die Konvertierung in der Datei auch explizit, sodass Sie zusätzlich `PDF` aus der folgenden Zeile löschen müssen:

```
<policy domain="coder" rights="none"
 pattern="{PS,PS2,PS3,EPS,PDF,XPS}" />
```

Außerdem verhindert die folgende Zeile den Aufruf von Wand (Sie können sie einfach mit `<!-- -->` auskommentieren):

```
<policy domain="delegate"
        rights="none" pattern="gs" />
```

Sobald ImageMagick die Konvertierung nicht mehr verbietet, ist Wand ganz leicht einzusetzen:

```
from wand.image import Image as WImage
WImage(filename="recipe-book.pdf[9]",
       resolution=110)
```

Die eckige Klammer hinter dem Dateinamen enthält die Seite aus dem PDF, die Wand in ein darstellbares Bild konvertiert. `resolution=110` setzt die Auflösung des Bilds auf 110 dpi. Jupyter kann das `WImage`-Objekt direkt als Bild einbetten und anzeigen.

```
pdf.image("Parchment.jpg", x=-4, y=-8,
                                   w=217, h=313)
```

Die Angabe der Größe streckt das Bild auf die angegebene Größe in Millimetern. Lässt man eine Größenangabe weg, erhält fPDF das Format und skaliert beide Seiten. Lässt man die Position weg, platziert fPDF es dort, wo es auch den nächsten Textblock platzieren würde.

Rezept programmiert

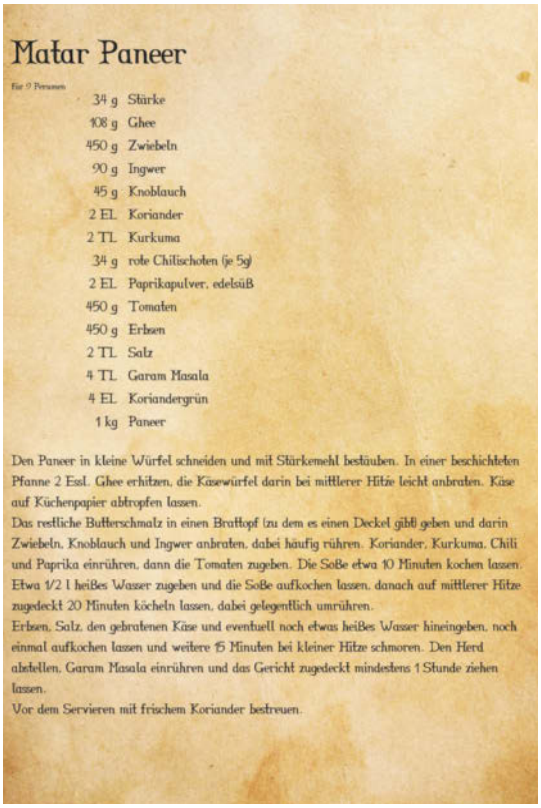
Jede Seite des kleinen Rezeptbuchs entsteht durch Textblöcke für die Überschrift und Mengenangabe (`cell()`), die Liste an Zutaten als Textblöcke mit rechts (`align="R"`) und links (`align="L"`) ausgerichteten Textblöcken (ein leerer Textblock dient als Abstandshalter) und der Beschreibung als Fließtext mit `write()`:

```
for recipe in recipes:
  pdf.add_page()
  pdf.image("Parchment.jpg",
           x=-4, y=-8, w=217, h=313)
  pdf.set_font('Titans2', size=28)
  pdf.cell(0, pdf.font_size * 1.8,
           txt=recipe["title"],
           ln=1, align="L")
  pdf.set_font('Titans2', size=8)
  pdf.cell(0, pdf.font_size * 2,
           txt=recipe["servings"],
           ln=1, align="L")
  pdf.set_font('Titans2', size=12)
  for ig in recipe["ingredients"]:
    pdf.cell(40, pdf.font_size * 1.0,
           txt=ig[0], ln=0, align="R")
    pdf.cell(2, 0,
           txt="", ln=0, align="R")
    pdf.cell(0, pdf.font_size * 1.0,
           txt=ig[1], ln=1, align="L")
    pdf.ln()
  pdf.ln()
  for line in recipe["description"]:
    pdf.write(h=pdf.font_size * 1.8,
           txt=line)
    pdf.ln()
pdf.output("recipe-book.pdf")
```

Die Funktion `ln()` fügt jeweils einen Zeilensprung ein.

Die Methoden sammeln inhaltsschwere Seiten zunächst nur im `pdf`-Objekt. Erst die `output()`-Funktion speichert das Dokument im angegebenen Pfad.

fPDF eignet sich sicherlich nicht, um kunstvolle PDFs für den nächsten Design-Wettbewerb zu programmieren. Das Framework ist aber leicht zu bedienen und baut verlässlich Dokumente zusammen. Das eignet sich nicht nur, um Daten automatisch in Jupyter-Notebooks druckreif aufzubereiten, sondern beispielsweise auch, um in einer Django-Anwendung Belege zu erzeugen, für ASCII-Art [1] oder Daumenkino [2]. (pmk) c't



fPDF platziert Text entweder mit millimetergenauer Angabe (Zutatenliste) oder automatisch als Absätze (Beschreibung).

Literatur

[1] Oliver Lau, **ASCII-Art,** Mit Python PDF-Dateien erstellen, c't 7/2016, S. 172

[2] Oliver Lau, **YouTube analog,** Kinderleicht vom Video zum Daumenkino in vier Schritten, c't 11/2016, S. 140

Beispielcode bei GitHub, Dokumentation:
www.ct.de/wp82

Best of IT-Security

Die Online-Konferenz für Security-Experten

AUSZUG AUS DEM VORTRAGSPROGRAMM:

- **Was tun, wenn's richtig knallt?**
  Umgang mit Sicherheitsvorfällen im Datenschutz –
  Joerg Heidrich, Justiziar / Datenschutzbeauftragter Heise Medien GmbH & Co. KG

- **IT vs. OT: Wir sind uns viel ähnlicher als wir uns unterscheiden –**
  Vergleich von Leitwarte und SOC Operationen –
  Marina Krotofil, Senior Security Engineer in einem global agierenden Konzern

- **Cyberangriffe gegen Unternehmen: Erste Ergebnisse einer repräsentativen Unternehmensbefragung in Deutschland –**
  Prof. Dr. Gina Rosa Wollinger, Professorin für Kriminologie und Soziologie an der Fachhochschule für öffentliche Verwaltung NRW

- **Informationssicherheit (nicht nur) in KMUs: Drei Methoden im Vergleich –**
  Tobias Glemser, Geschäftsführer der secuvera, BSI-zertifizierter Penetrationstester und Technischer Leiter für Penetrationstests

- **Incident Response in der Zukunft –**
  Bruce Schneier, Kryptografie-Experte aus den USA

Jetzt als Aufzeichnung im heise shop erhältlich: **shop.heise.de/katalog/best-of-it-security**

Urlaubsbilder sortieren

Alle Digitalkameras speichern das Aufnahmedatum in den EXIF-Daten der Bilddateien. Mit einem kleinen selbst geschriebenen Python-Skript sind die Urlaubsbilder ruckzuck nach Tagen in Ordner sortiert und aussagekräftig umbenannt.

Von Pina Merkert

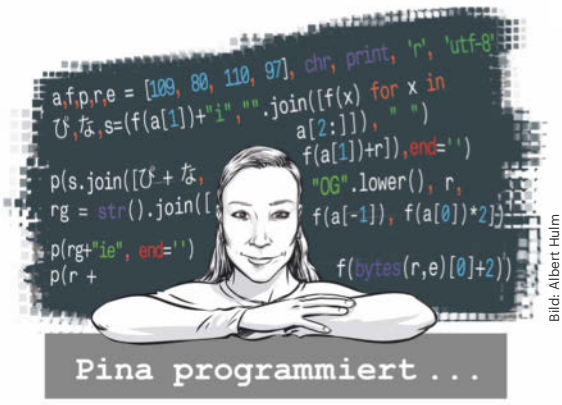


Bild: Albert Hulm

Bei meiner letzten Reise hatten wir zu dritt fünf Kameras dabei: drei Smartphones und zwei Digitalkameras. Die Bilder landeten am Abfahrtstag jeweils in eigenen Ordnern für jedes Gerät. Daran erkenne ich zwar, wer ein Bild geschossen hat, für die Diashow bei den Großeltern würde ich aber lieber alle Bilder in zeitlicher Reihenfolge zeigen – und zwar mit je einem Ordner pro Tag, damit ich den langweiligen Regentag leicht überspringen kann, wenn Oma während der Show erste Anzeichen von Müdigkeit zeigt.

Das Änderungsdatum der Dateien enthält nicht das Datum, wann die Bilder entstanden sind, sondern nur, wann ich sie kopiert habe. Glücklicherweise speichern Kameras aber mit jedem Bild auch Metadaten (EXIF-Daten) wie Aufnahmeort, Belichtungszeit, Linse und den Aufnahmezeitpunkt. Während ich nur von Fotos aus Kameras mit GPS-Empfänger erwarten kann, dass sie den Aufnahmeort enthalten, findet sich in den EXIF-Daten aller Fotos ein genauer Zeitstempel der Aufnahme. Um die Bilder nach Tagen zu ordnen, muss ein Programm also nur diesen Zeitstempel auslesen. Mit der `exif`-Bibliothek aus dem Python-Package-Index (PyPI) geht das ziemlich einfach.

Exif lesen

Ein einfaches `pip install exif` lädt die `exif`-Bibliothek auf die Platte. Anaconda-Nutzer ersetzen wie üblich `pip` durch `conda`: `conda install -c zegami exif`.

Alle Funktionen stellt die `Image`-Klasse bereit, die man einfach mit einem File-Handle der geöffneten Binärdatei füttert:

```
from exif import Image
with open(filename, 'rb') as file:
  image_obj = Image(file)
```

Ein `dir(image_obj)` enthüllt anschließend, auf welch vielfältige Bild-Metainformationen man mit der `exif`-Bibliothek zugreifen könnte. Fürs Aufnahmedatum reicht `image_obj.datetime`.

Datum konvertieren

Das per `image_obj.datetime` gelesene Datum ist leider nur ein String, mit dem sich nicht so bequem rechnen lässt wie mit Pythons `datetime`-Objekten. Mit `datetime.strptime()` ist der String aber in einem Wimpernschlag konvertiert:

```
from datetime import datetime
image_time = datetime.strptime(
                 image_obj.datetime,
                 "%Y:%m:%d %H:%M:%S")
```

`datetime` formatiert den Zeitstempel mit `strftime()`. Um beispielsweise einen String mit „Jahr_Monat_Tag-Wochentag“ zu erzeugen, reicht die folgende Zeile:

```
day_str = image_time.strftime(
                        "%Y_%m_%d-%A")
```

Mein Skript verwendet das für die Namen der Unterordner für die Tage, weil dann bei alphabetischer Sortierung im Dateimanager alle Ordner in der richtigen Reihenfolge stehen.

Dem gleichen Gedanken folgend erzeugt `image_time.strftime("%H%M%S_")` ein Präfix für die Dateinamen der Bilder, sodass diese innerhalb der Tages-Ordner nach Uhrzeit sortiert stehen.

Sammeln und kopieren

Pythons integriertes `os`-Modul listet mit `os.listdir()` den Inhalt von Ordnern auf und bastelt mit `os.path.join()` plattformübergreifend Pfadnamen zusammen. Zwei Schleifen lesen damit ohne Mühe alle Dateinamen der ursprünglichen Urlaubsbilder aus dem Dateisystem:

```
import os
for sfld in os.listdir(image_folder):
  for im in os.listdir(
    os.path.join(image_folder, sfld)):
```

`os` legt auch Ordner an, allerdings nur ohne Fehler, falls es den Ordner nicht schon bereits gibt:

```
if not os.path.exists(os.path.join(
            output_folder, day_str)):
  os.makedirs(os.path.join(
            output_folder, day_str))
```

Kopieren kann `os` aber nicht. Das übernimmt `shutil`, ebenfalls eines von Pythons Standardmodulen. Die Kopierfunktion `copy2()` erhält beim Kopieren einer Datei die ursprünglichen Änderungszeiten, sodass es aussieht, als hätte man das Bild von der SD-Karte direkt in die neue Ordnerstruktur kopiert:

```
import shutil
shutil.copy2(
 os.path.join(image_folder, sfld, im),
 os.path.join(output_folder, day_str,
      image_time.strftime("%H%M%S_") +
      sfld + "_" + filename + "." +
      extension.lower()))
```

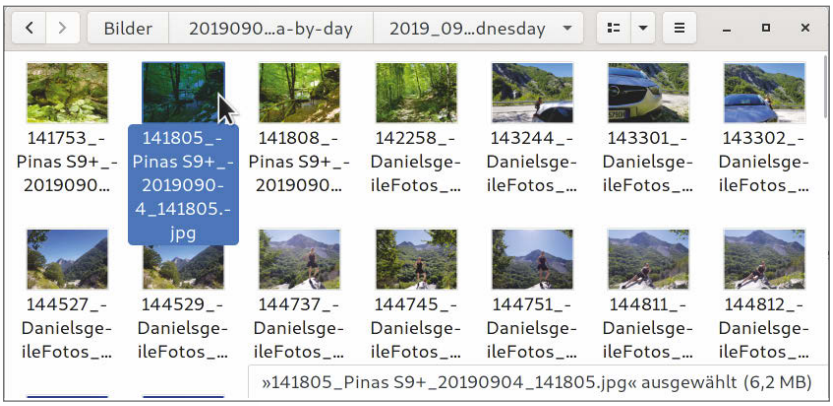
Die Variablen `filename` und `extension` pult mein Skript per Regex (`re`-Modul) aus dem Dateinamen und prüft dabei auch gleich das Namensformat. Das Skript finden Sie in Gänze über ct.de/wtnz im GitHub-Repository.

Es nutzt zusätzlich noch `sys` für eine textbasierte Fortschrittsanzeige, da es einige Dutzend Sekunden dauert, gigabyteweise Bilder zu kopieren. Da ich für die Bequemlichkeit alles in einem Jupyter-Notebook programmiert habe, kommt dort auch noch `from IPython.display import clear_output` für die Anzeige des Fortschrittsbalkens zum Einsatz.

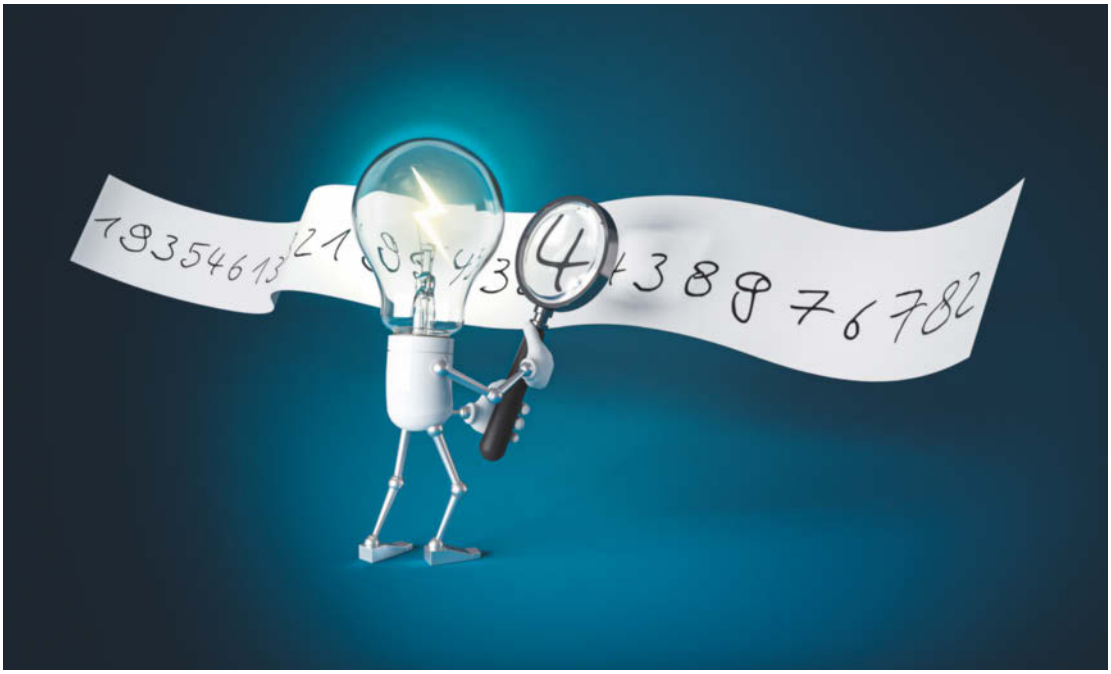
Wenn Sie mein Skript verwenden wollen, passen Sie zuerst die Variablen `image_folder` und `output_folder` an die Pfade an, wo Ihre Urlaubsbilder liegen und geordnet hinkopiert werden sollen. Danach führen Sie das Skript nur noch mit `python imagesbyday.py` aus. (pmk) c't

Skript bei GitHub:

www.ct.de/wtnz



Das Skript schiebt die Bilder nicht nur in einen Ordner pro Tag, sondern benennt sie auch so um, dass sie in der zeitlich richtigen Reihenfolge erscheinen.



KI für Einsteiger

Neuronale Netze klingen nach Gehirn und komplexer Biologie. Maschinelles Lernen leiht dort aber nur primitive Ideen. Eigentlich erstellt der Rechner nämlich nur selbstständig Statistiken – und die wenigen dafür nötigen Zeilen Python kann jeder auf dem heimischen Rechner leicht selbst programmieren. Um die Mathematik dahinter kümmert sich das Framework Keras automatisch.

Von Pina Merkert

Computer sind dumme Maschinen. Sie arbeiten zwar Millionen von Befehlen pro Sekunde ab, es muss aber stets ein Programmierer exakt festlegen, was die Maschine wann tun soll. Das ist das Prinzip klassischer Programme von Menschenhand.

Einen ganz anderen Ansatz verfolgt künstliche Intelligenz in Form des maschinellen Lernens (ML): Der Rechner bleibt zwar gleich dumm, aber Framework-Programmierer haben lernfähige Algorithmen vorgegeben, die sich selbstständig an eine Herausforderung anpassen können. Diese Anpassung erfolgt in einer Trainingsphase, in der die Algorithmen Tausende verschiedener Beispiele zu sehen bekommen und interne Parameter dabei so verändern, dass sie möglichst gut den Beispielen entsprechen. Ein Lernalgorithmus startet also zumeist in einem Zustand, in dem er kein Problem zufriedenstellend löst. Durch das Training spezialisiert er sich auf die Aufgabe, die der Datensatz implizit vor-

gibt. Erst nach dem Training meistert der Algorithmus die gestellte Herausforderung, was Data Scientists als „Inferencing“ bezeichnen.

Es gibt bereits seit vielen Jahrzehnten solche lernfähigen Algorithmen. Die „klassischen Verfahren“ bauen auf einfachen Ideen auf. K-Nearest-Neighbour sucht beispielsweise eine vorgegebene Zahl (k) an Beispielen aus den Trainingsdaten, die am ehesten zu einer neuen Eingabe passt, und mittelt für seine Vorhersage die Ausgaben aus diesen Beispielen. Entscheidungsbäume lernen, anhand einer Kaskade von Wenn-Dann-Entscheidungen die passenden Ausgaben zu liefern. In der ersten Kinect hat diese einfache Idee gereicht, damit Microsoft damit Körperteile erkannte. Bayesian-Networks, Hidden-Markov-Models und Support-Vector-Machines erstellen alle automatisierte Statistiken und liefern teils erstaunlich gute Ergebnisse. Die Qualität der Vorhersagen hängt nämlich nur bedingt vom verwendeten Algorithmus ab. Auch klassische effiziente ML-Algorithmen liefern mit guten Trainingsdaten akkurate Resultate.

Gute Daten sind teuer

Trainingsdaten zu erzeugen war aber lange Zeit ein zu großer Kostenfaktor. Oft war es billiger, wenn ein Programmierer das Problem durchschaute und dann den für die Lösung notwendigen Code per Hand zusammenschrieb. Das änderte sich jedoch mit dem „Deep Learning“. Das „tiefe“ Lernen nutzt neuronale Netze, einen Lernalgorithmus, den es bereits seit den 1950er-Jahren gibt. Dessen simulierte Neuronen lassen sich leicht stapeln, was für tiefe Strukturen sorgt. Die Tiefe nutzt das Deep Learning dazu, aus wenig vorstrukturierten Datensätzen automatisch Features zu extrahieren. Mit denen kann das Netzwerk erlernen, die richtigen Ausgaben zu berechnen, obwohl kein Programmierer die Daten aufwendig per Hand vorverarbeitet hat. Man braucht für diese Art von Lernalgorithmus zwar noch mehr Trainingsbeispiele als für die klassischen Verfahren, spart aber Programmierarbeit.

Maschinelles Lernen ist damit etwas für Programmierer, die es nicht so genau nehmen wollen: Statt eines exakten Algorithmus wählt man nur ein grobes Modell für die Berechnung. Dazu sammelt man den Datensatz mit Beispielen und lässt den Rechner selbst herausfinden, wie die Details aussehen müssen, damit das Modell zu den Beispielen passt.

Gerade beim Deep Learning bestehen die Modelle aber aus unüberschaubar vielen Neuronen mit Millionen von Parametern, die sich während des Trainings alle gleichzeitig ändern. Mancher fühlt sich daher von der Komplexität solcher Systeme erschlagen. Die zugrunde liegenden Ideen sind aber ausgesprochen einfach und die Netzwerke wenden lediglich die immer gleiche Idee wieder und wieder an. Mit dem Code aus diesem Artikel trainieren Sie selbst auf betagten Notebooks im Handumdrehen ein neuronales Netz. Keine Angst vor komplexen Netzen: Der beschriebene Code fängt ganz klein an.

Datensatz: Pina schläft?

Zeit	x1: Schloss	x2: Strom	y: Pina schläft
15:00	0	0	0
17:00	0	0	0
19:00	1	0	0
21:00	1	20	0
23:00	1	20	0
01:00	1	0	1
03:00	1	0	1
05:00	1	0	1

Ein Neuron

Maschinelles Lernen beginnt stets mit einem Datensatz (siehe Tabelle). Im ersten Beispiel geht es darum, aus dem Status von Pinas smartem Fahrradschloss und dem Stromverbrauch gemessen an ihrem Lichtschalter vorherzusagen, ob Pina gerade schläft oder nicht. Eine einfache Funktion, um das zu berechnen, wäre:

```
y_ = max(w1 * x1 + w2 * x2 + b, 0)
```

Bei `x1` handelt es sich um den Status des Schlosses, bei `x2` um den gemessenen Stromverbrauch in Watt. `w1`, `w2` und `b` sind Parameter, die der Rechner aus den Daten lernen soll. Die `max()`-Funktion mit `0` ist eine Aktivierungsfunktion, die sogenannte „rectified linear unit“ (ReLU). Die ReLU sorgt dafür, dass das Ergebnis `y_` nie negativ wird. Diese einfache Berechnung als Funktion zu definieren reicht als Modell schon aus.

Beim Training geht es nun darum, die drei Parameter automatisch so zu wählen, dass es für die Daten aus der Tabelle keinen Unterschied zwischen der errechneten Vorhersage für den Schlafstatus `y_` und dem gemessenen Status `y` gibt. Da der Rechner dabei nicht sofort richtig liegt, definiert man die Loss-Funktion. Ihr Wert muss klein

sein, wenn der Rechner seine Aufgabe gut macht, und groß, wenn es große Unterschiede zwischen y_ und y gibt. Da sie nicht negativ werden soll, bietet sich die quadrierte Differenz als Loss an:

```
loss = (y - y_) ** 2
```

Das Training besteht nun darin, die Parameter mit zufälligen Werten zu befüllen und einen Optimierungsalgorithmus anzuwenden, der die Parameter in kleinen Schritten so anpasst, dass `loss` immer kleiner wird. Eine effiziente Methode nutzt die „Backpropagation of Errors“ in Form der Ableitung (Gradient) der Loss-Funktion. Weniger mathematisch ausgedrückt schaut der Algorithmus in jedem Schritt, welcher Parameter wie stark dazu beigetragen hat, dass `loss` größer 0 war, und passt diesen Parameter entsprechend mehr oder weniger an. Der Algorithmus heißt „Gradientenabstieg“, weil er die analytisch berechnete Ableitung nutzt, um herauszufinden, in welche Richtung die Loss-Funktion am steilsten abfällt. Der Gradientenabstieg funktioniert mit beliebig vielen Parametern, sodass man auch sehr komplexe Modelle mit vielen Parametern mit dem immer gleichen Algorithmus optimieren kann.

Im Beispiel, ob Pina schläft, findet der Gradientenabstieg die Parameter `w1=1.0`, `w2=-0.05` und `b=0.0`. In die Formel eingesetzt kommen damit die Werte `0.0`, `0.0`, `1.0`, `0.0`, `0.0`, `1.0`, `1.0` und `1.0` heraus. Der dritte Wert (19:00 Uhr) sollte aber `0.0` sein. Das Modell ist hier zu einfach und erreicht nach dem Training deshalb nur eine Genauigkeit von 87,5 Prozent. So etwas passiert beim maschinellen Lernen oft. Um die Genauigkeit zu steigern, könnte man die Eingabedaten um die Uhrzeit erweitern: dann lernt das System, dass Schlafen nachmittags unwahrscheinlich ist.

Bei der einfachen Formel für das Modell handelt es sich übrigens um ein einzelnes simuliertes Neuron. Nutzt man die Ergebnisse einer solchen Berechnung als Eingaben für weitere Neuronen, hat man bereits ein mehrschichtiges neuronales Netz definiert. Mit den komplexen Strukturen in echten Gehirnen hat das also wenig zu tun. Die Formeln ähneln eher den bedingten Wahrscheinlichkeiten aus der Bayes’schen Statistik.

Handgeschriebene Ziffern

Mit einem so einfachen Beispiel, bei dem man perfekte Parameter in Sekunden erraten kann, erschließt sich der Hype ums Deep Learning natürlich nicht. Deswegen gilt als „Hallo Welt“ der KI auch das Erkennen von handgeschriebenen Ziffern aus dem MNIST-Datensatz. Der enthält 70.000 28 × 28-Pixel große Bilder und dazu jeweils die passende Ziffer, die die KI erkennen soll.

Um mit solchen Datensätzen zu arbeiten, eignet sich ein Framework mit Keras-API, beispielsweise TensorFlow. Es kümmert sich nicht nur um die Mathematik hinter den Algorithmen, sondern bringt für einen schnellen Einstieg auch Funktionen mit, um übliche Datensätze wie MNIST automatisch zu laden. Der Befehl `pip install tensorflow` installiert das Framework samt Abhängigkeiten in jede Python-Umgebung (bis Python 3.7), die `pip` kennt (beispielsweise ein frisches Virtualenv unter Linux oder eine Anaconda-Installation unter Windows). Den Datensatz laden dann folgende vier Zeilen Python:

```
from tensorflow.keras.datasets import mnist
train_da, test_da = mnist.load_data()
x_train, y_train = train_da
x_test, y_test = test_da
```

`load_data()` lädt die Daten aufgeteilt in zwei Teile: 60.000 Bilder (`x_train`) und Labels (`y_train`) dienen zum Training. Die 10.000 Bilder und Labels in

An diesen zwölf Bildern aus dem MNIST-Datensatz wird klar, dass die Erkennung für den Rechner nicht trivial ist. Beispielsweise könnte auch ein Mensch die 5 ganz rechts in der dritten Zeile fälschlicherweise als 6 identifizieren.

```
import tensorflow.keras.backend as K
from tensorflow.keras.utils ↵
            ↳import to_categorical
dat_form = K.image_data_format()
rows, cols = 28, 28
train_size = x_train.shape[0]
test_size = x_test.shape[0]
if dat_form == 'channels_first':
  x_train = x_train.reshape(
        train_size, 1, rows, cols)
  x_test = x_test.reshape(
        test_size, 1, rows, cols)
  input_shape = (1, rows, cols)
else:
  x_train = x_train.reshape(
        train_size, rows, cols, 1)
  x_test = x_test.reshape(
        test_size, rows, cols, 1)
  input_shape = (rows, cols, 1)
# norm data to float in range 0..1
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
# conv class vecs to one hot vec
y_train = to_categorical(y_train,10)
y_test = to_categorical(y_test, 10)
```

Vor dem Training muss man die Daten normieren und die korrekte Ausgabe als Vektor darstellen.

`x_test` und `y_test` bekommt der Trainingsalgorithmus bewusst nicht zu sehen. Mit ihnen prüft man nach dem Training, ob die KI nur Beispiele auswendig gelernt hat oder auch mit unbekannten Daten zurechtkommt.

Neuronale Netze erwarten ihre Eingaben normalerweise als Gleitkommazahlen mit 32 Bit, normalisiert auf den Wertebereich zwischen 0 und 1. Der Code im Kasten konvertiert die Daten ins richtige Format und gibt den Bilddaten auch gleich eine zweidimensionale Struktur (erst später relevant, um ein Convolutional Network zu trainieren).

Um für die ersten Tests schnell loszulegen, haben wir die Menge der Trainingsdaten zunächst auf 100 Datensätze zusammengestutzt:

```
x_train = x_train[:100]
y_train = y_train[:100]
```

Ein ganz einfaches Modell

Die meisten neuronalen Netze bestehen aus Schichten mit voneinander unabhängigen „Neuronen“, von denen jedes wie die einfache Funktion vom Anfang funktioniert (Eingaben mit Gewichten multiplizieren, Bias addieren, nicht lineare Aktivierungsfunktion auf das Ergebnis anwenden). Die Ausgaben einer Schicht verwenden diese Netze jeweils als Eingabe der nächsten Schicht. Die Ausgabeschicht enthält bei Klassifikationsaufgaben für jede zu unterscheidende Klasse ein Neuron. Ein solches Modell erzeugt Keras weitgehend automatisch mit der Klasse `Sequential()`:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Flatten
model = Sequential()
model.add(Flatten())
model.add(Dense(10,activation='softmax'))
```

Dieses Modell verwandelt mit `Flatten()` die Eingabedaten (Array aus 28 × 28 Zahlen) zuerst von einem 2D-Bild in einen eindimensionalen Vektor mit 28 × 28 = 784 Dimensionen. Die Funktion `model.add()` fügt die Berechnungen jeweils als eine Schicht zum `'Model'`-Objekt hinzu. Danach berechnet `Dense()` die Aktivierungen für zehn Neuronen, eines für jede Ziffer, die herauskommen kann. Im Idealfall gibt für jedes Eingabebild nur ein Neuron einen nennenswert großen Wert aus.

Die Aktivierungsfunktion `'softmax'` normiert die Aktivierungen der zehn Neuronen, sodass sie in der Summe 1 ergeben. Dadurch kann man mit dem Ergebnis wie mit Wahrscheinlichkeiten rechnen. Ist eine der zehn Zahlen im Ausgabevektor beinahe 100 Prozent, ist sich das neuronale Netz sehr sicher. Sind die zehn Zahlen in der Nähe von 10 Prozent, hat das Netz keine Ahnung, welche Ziffer im Eingabebild geschrieben wurde.

TensorFlow baut intern einen Graphen an Berechnungen auf, optimiert diesen automatisch und bringt ihn in ein Format, damit der Rechner ihn effizient auf der Hardware ausführen kann. Das Framework nutzt dabei auch Grafikkarten und KI-Beschleuniger, sofern diese im Rechner stecken.

Der Python-Code für das Modell definiert die Berechnungen von den Eingaben zu den Ausgaben (forward pass). Zum Trainieren bestimmt TensorFlow vollautomatisch die Ableitung der Loss-Funk-

tion (backward pass) und fügt die Berechnungen dafür zusammen mit dem Algorithmus für den Gradientenabstieg zum Berechnungsgraph hinzu. Um all das zu definieren, reicht eine Zeile:

```
from tensorflow.keras.losses import ↵
            ↳categorical_crossentropy
from tensorflow.keras.optimizers import Adam
model.compile(
  loss=categorical_crossentropy,
  optimizer=Adam(),
  metrics=['accuracy'])
```

Die `categorical_crossentropy` ist eine Loss-Funktion, die den quadrierten Abstand zwischen berechneten Ausgaben und gewünschten Daten (One-Hot-Vektor der gewünschten Klasse) berechnet. Sie eignet sich für alle Klassifizierungsprobleme, bei denen der Datensatz korrekte Antworten enthält (supervised Learning).

Der Optimierungsalgorithmus `Adam()` ist eine Variante des Gradientenabstiegs, die Parameter wie Schrittweite selbst während des Trainings anpasst. Der Adam-Algorithmus funktioniert recht robust für die meisten Probleme beim Deep Learning, ohne dass man viele Hyperparameter per Hand einstellen müsste.

Der Parameter `metrics=['accuracy']` veranlasst Keras, beim Training nicht nur Loss zu berechnen, sondern nachzuzählen, bei wie vielen Bildern das Netz die korrekte Ziffer errät.

Training starten

Um das Training zu starten, reicht ein Aufruf von `model.fit()`:

```
history = model.fit(x_train, y_train,
    batch_size=128,
    epochs=12, verbose=1,
    validation_data=(x_test, y_test))
```

Die Funktion nimmt als ersten Parameter die Eingaben und als zweiten Parameter die Ausgaben entgegen. Die `batch_size` legt fest, wie viele Datensätze das Netz auf einmal berechnet. Man wählt die Batches möglichst groß, da das die Vektoreinheiten der CPU und mehr noch der Grafikkarten auslastet. Außerdem stabilisieren große Batches das Training, da Ausreißer im Datensatz dann nicht zu starken Ausschlägen bei den Gradienten führen, die den Optimierungsalgorithmus in die falsche Richtung lenken. Die natürliche Grenze für die Batchgröße ist der Arbeitsspeicher, da für eine effiziente Berechnung alle Daten eines Batches in den Speicher passen müssen.

Der Parameter `epochs` legt fest, wie oft der Trainingsalgorithmus alle 128 Trainingsdaten durchläuft. Zwölf dieser Epochen reichen für dieses Beispiel aus. Für Probleme, die beim Training mehr Schwierigkeiten bereiten, muss man auf eine kleinere Lernrate (ein Hyperparameter des Optimierungsalgorithmus) und mehr Epochen ausweichen.

Damit TensorFlow nach jeder Epoche testet, wie gut das Netz auf unbekannten Daten abschneidet, reicht es, die Testdaten lediglich als `validation_data=(x_test, y_test)` anzugeben.

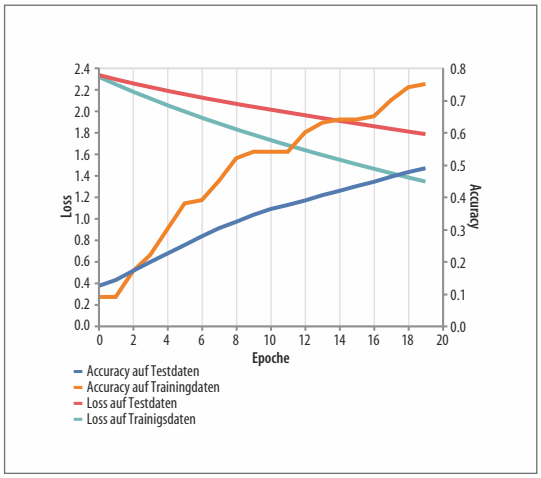
Die Funktion `model.fit()` gibt ein `History`-Objekt mit statistischen Daten über das Training zurück. Um die später auszuwerten, lohnt es, diese in der Variable `history` zu speichern. Wir haben mit diesen Daten die Diagramme in diesem Artikel erzeugt. Wie das geht, sehen Sie im Repository auf GitHub, das Sie über ct.de/wd2e finden.

Tieferes Netz

Das Netz mit nur einer Schicht errät nur mit einer Wahrscheinlichkeit von 49 Prozent die richtige Ziffer (Accuracy auf Testdaten). Das bescheinigt dem Netz noch keine besondere Intelligenz (immerhin besser als raten). Um es intelligenter zu machen, muss man ihm nur mehr Neuronen geben und dafür direkt hinter `Flatten()` eine weitere Schicht einfügen:

```
model.add(Dense(200,activation='relu'))
```

Die zusätzliche Schicht hat 200 Neuronen (mit der Anzahl können Sie gern herumspielen), die zuerst



Mit einer einzigen Schicht Neuronen erreicht das Netz eine Genauigkeit von nur 49 Prozent.

über die Eingaben „nachdenken“. Aus deren Ergebnissen berechnet dann die bereits vorher definierte Ausgabeschicht ihre Vermutung, welche Ziffer im Bild zu sehen ist.

Da die neue Schicht von außen nicht sichtbar ist (keine ihrer Aktivierungen landet direkt im Ausgabevektor), bezeichnet man sie als „hidden layer“. Beim Deep Learning bestehen die Netze größtenteils aus solchen versteckten Schichten.

Ein erneuter Trainingsdurchlauf zeigt: Die zusätzlichen Neuronen verbessern das Netz erheblich. Bei den Trainingsdaten liegt es nach 20 Epochen bereits in zehn von zehn Fällen richtig. Bei den Testdaten schneidet es mit knapp 70 Prozent allerdings deutlich schlechter ab. Was ist passiert?

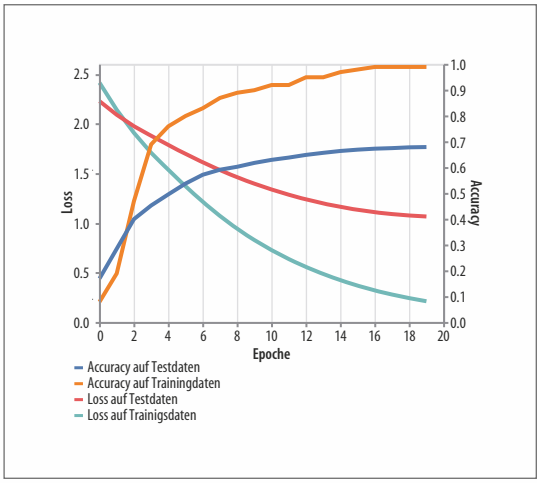
Nichts verstanden

Neuronale Netze verhalten sich oft wie faule Schüler: Manchmal reicht es für ein gutes Ergebnis auf den Trainingsdaten, diese auswendig zu lernen. Es würde einfach mehr Mühe kosten, den tatsächlichen Zusammenhang zu verstehen, als die Beispiele in den Parametern zu kodieren. Beim Gradientenabstieg gab es in diesem Fall einen steileren Abhang fürs Auswendiglernen als fürs Verstehen des Zusammenhangs.

Dieses Phänomen heißt „Overfitting“ und man erkennt es daran, dass ein Netz auf unbekannten Daten deutlich schlechter abschneidet als auf den Trainingsdaten.

Die naheliegende Maßnahme gegen Overfitting ist ein größerer Datensatz. Muss das Netz beim Training mit mehr unterschiedlichen Daten zurechtkommen, lohnt es sich irgendwann nicht mehr, einzelne der vielen Daten auswendig zu lernen. Das Verstehen des Zusammenhangs entpuppt sich dann als der einfachere Weg. Statt mit 100 Bildern sollten Sie daher immer mit allen 60.000 trainieren, was entsprechend länger dauert.

Bei vielen KI-Aufgaben ist es aber sehr aufwendig und teuer, einen genügend großen Datensatz zusammenzustellen, der Overfitting verhindert. Daher hat sich ein Trick beim Definieren des Modells eingebürgert: Zwischen den Schichten mit Neuronen schiebt man `Dropout()` ein. Diese Schicht setzt zufällig einen Teil der Aktivierungen der vorherigen Schicht auf 0. Damit kommt das Netz nicht mehr verlässlich an jeden Parameter und tut sich schwerer im Auswendiglernen. Der Nachteil davon: Das Training dauert länger und verschlingt mehr Rechenleistung. Eine Schicht, die zufällig die Hälfte



Nichts verstanden, nur Trainingsdaten auswendig gelernt: Beim gefürchteten „Overfitting“ schneidet das Netz auf den Trainingsdaten viel besser ab als auf den Testdaten.

aller Informationen verwirft, definiert die folgende Zeile (eingefügt zwischen den hidden layer und der Ausgabeschicht):

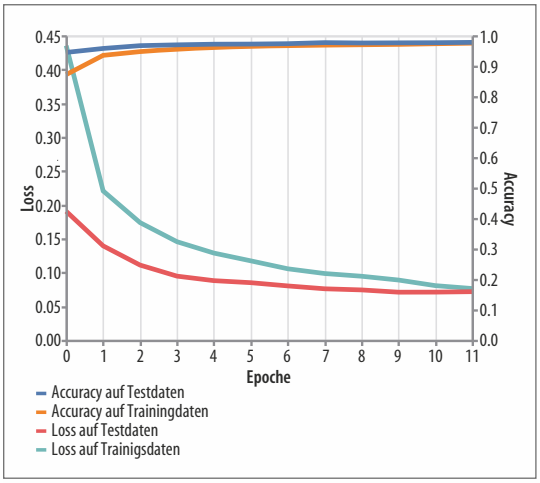
```
from keras.layers import Dropout
model.add(Dropout(0.5))
```

Mit dem so erhöhten Aufwand dreht der Lüfter beim Training schon einige Sekunden hoch. Das Netz erkennt dafür aber bereits in fast 98 Prozent der Fälle die richtige Ziffer. Mit mehr Rechenleistung (beispielsweise von einer Grafikkarte) oder längerer Rechenzeit als bei den ersten Versuchen geht es aber noch besser.

Convolutional Network

Mit Convolutional Networks verabschieden sich neuronale Netze endgültig vom Gehirn als Vorlage. Die Netze nutzen hier die mathematische Operation der Faltung, um kleine Matrizen schrittweise übers Bild zu schieben und dabei Formen zu erkennen. Das lässt sich effizient berechnen, da die Schicht dabei wenige Parameter für die Gewichte immer wieder verwendet. Stellt man sich das mit einem biologischen Gehirn vor, wäre das, als würde ein Neuron die Synapsen eines benachbarten Neurons kopieren – das kommt in einem echten Gehirn sicher nicht vor.

Mit Faltungen lassen sich ganz hervorragend Filter erlernen, um Formen in einem Bild zu erken-



Trainiert mit 60.000 Datensätzen und Dropout, kommt es nicht mehr zum Overfitting – und das Netz liegt in knapp 98 Prozent der Fälle richtig.

nen. Die so erkannten Features nutzt dann die nächste Schicht zum Erlernen noch komplexerer Formen und so weiter. Kommen auf diese Weise mehr als vier Schichten zusammen, spricht man von „Deep Learning“.

In dem `model` in Keras zwei zusätzliche Schichten mit Faltungen zu definieren, geht genauso einfach wie bei anderen versteckten Schichten:

```
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
model.add(Conv2D(
            32, kernel_size=(3, 3),
            activation='relu',
            input_shape=input_shape))
model.add(Conv2D(
            64, kernel_size=(3, 3),
            activation='relu'))
model.add(MaxPooling2D(
            pool_size=(2, 2)))
model.add(Dropout(0.25))
```

Die neuen `Conv2D()`-Schichten gehören vor die bestehende `Flatten()`-Schicht, da sie die zweidimensionale Struktur der Eingabebilder nutzen (den `Dense()`-Layern war diese Struktur egal). Die erste Schicht lernt 32 Filter mit einer Matrixgröße von 3 × 3, die zweite 64 davon.

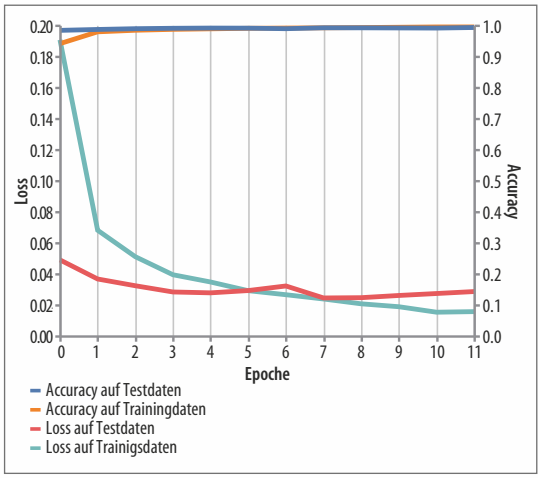
Danach folgt ein Pooling-Layer, der von je vier Aktivierungen (2 × 2) immer nur die größte weitergibt. Solche Schichten reduzieren die Auflösung des Bilds und verwerfen dabei einen Teil der Pixel. Da mit der Anzahl der Filter aber auch die Zahl der Kanäle pro Pixel steigt, muss das Netz hier nur lernen, die Information über die von den Filtern erkannten Formen zu kodieren.

Dieses Netz zu trainieren dauert auf einem älteren Notebook mit vier Kernen nun schon mehr als 15 Minuten. Die Trainingszeit hängt sehr stark daran, wie viele CPU-Kerne mitarbeiten oder wie viele Compute-Cores die Grafikkarte mitbringt. Dafür erkennt das Netz nach dem Training mehr als 99 Prozent der Ziffern richtig. Damit schneidet es bei dieser Aufgabe besser ab als ein Mensch.

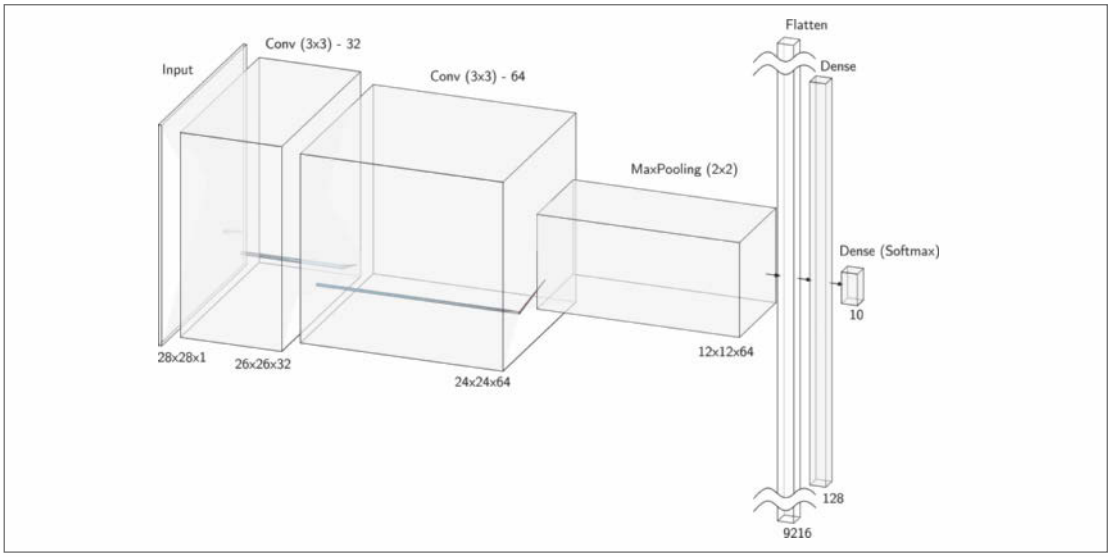
Spielwiese für Experimentierer

Das Keras-API in TensorFlow macht es extrem einfach, Modelle für verschiedenste Netzwerkarchitekturen zu definieren. Das lädt zum Experimentieren ein, denn am Optimierungsalgorithmus und der Loss-Funktion muss man nur in seltenen Fällen drehen.

Möchte man eine bestehende Lösung, beispielsweise aus einem Forschungspapier, für einen eigenen Datensatz anpassen, reicht es meist, die Struktur zuerst nachzubauen und an den Hyperparametern zu drehen. Wer dabei nicht die Übersicht über die berechneten Experimente verlieren will, dem sei das Framework Sacred empfohlen (siehe Seite 70), das systematisch Parameter und Ergebnisse durchsuchbar protokolliert. Der Code in unserem Repository (siehe ct.de/wd2e) nutzt bereits Sacred.



Die Trainingskurve zeigt: Das Convolutional Network lernt ohne Overfitting und liegt in mehr als 99 Prozent der Beispiele richtig.



Das Convolutional Network nutzt die zweidimensionale Struktur der Bilddaten, um auf den ersten beiden Schichten Features zu lernen, die überall im Bild vorkommen können.

Wir raten sehr dazu, Deep-Learning-Experimente selbst auszuführen. Man bekommt dadurch ein Gefühl dafür, wo die Probleme bei Datensätzen und Netzwerkstrukturen liegen, und lernt den Hype ums Deep Learning besser einzuschätzen. Nützlich ist die Technik auf jeden Fall und viele Ergebnisse sind faszinierend, wenn man sich vor Augen führt, wie einfach die zugrunde liegenden Algorithmen sind.

Microsoft, Google und Amazon bieten in ihren Clouds über APIs Zugriff auf trainierte Netze, die transkribieren, vorlesen, übersetzen, Zustimmung messen und Bilder analysieren. Auch zahlreiche Start-ups erlauben API-Zugriff auf ihre trainierten Netze, geben dabei aber ebenfalls die genauen Parameter nicht preis.

Leider ist nicht jede Anwendung, die mit „KI“ beworben wird, wirklich schlau. Manche Firmen verwenden kein maschinelles Lernen, andere trainieren neuronale Netze mit ungenügenden Daten. Leider gibt es beim Deep Lerning viele zu kleine Datensätze oder solche mit einem Bias, der die Ergebnisse wertlos macht. Wer neuronale Netze mal selbst trainiert hat, erkennt solche Probleme oder weiß, die richtigen Fragen zu stellen. (pmk) c't

Code bei GitHub, Keras-Doku:
www.ct.de/wd2e

Sacred verwaltet KI-Experimente

Wer mit Programmen wie neuronalen Netzen, Simulationen oder Benchmarks experimentiert, investiert in jeden Versuch eine Menge Rechenzeit. Zu blöd, wenn man dann Parameter oder Ergebnisse aus früheren Experimenten vergisst, die beim aktuellen Versuch hätten helfen können. Sacred automatisiert das Pflegen der Datenbank, der nichts mehr entgeht.

Von Pina Merkert

Experimentiert man wie beim Machine Learning mit Algorithmen, die Stunden oder sogar Tage brauchen, um Ergebnisse zu liefern, verursacht jedes Experiment Kosten. Der Strom und vor allem die investierte Zeit schlagen so deutlich zu Buche, dass es sich lohnt, die Ergebnisse aller – auch fehlgeschlagener – Versuche aufzubewahren und alle Parameter zu notieren. In der Praxis ist der Dokumentationsaufwand aber zu groß, um das per Hand zu machen.

Die Python-Bibliothek Sacred löst dieses Problem. Sie bietet eine Schnittstelle, um die Parameter rechenleistungshungriger Experimente elegant zu definieren, sammelt automatisch Informationen zum System, archiviert den ausgeführten Quellcode und speichert alles in einer Datenbank. Dabei kontrolliert sie die verwendeten Zufallszahlengeneratoren, sodass sich archivierte Experimente komplett reproduzieren lassen.

Um das zu schaffen, ohne den Code der Experimente zu strecken, greift Sacred tief in die Python-Trickkiste. Mit ein paar strategisch platzierten Dekoratoren macht man eigenen Code ruckzuck fit fürs Archiv. Ein bequemes Starten der Experimente über die Kommandozeile gibt es automatisch dazu. Worauf man dabei achten sollte, erklärt dieser Artikel.

Automatisch protokollieren

Das Hallo-Welt-Beispiel der KI-Forschung ist das Klassifizieren von handgeschriebenen Ziffern aus dem MNIST-Datensatz. Als Beispiel dient eine Implementierung aus der Dokumentation des Machine-Learning-Fameworks Keras (siehe Kasten auf S. 72), die mit einem Convolutional Network eine Erkennungsrate von über 99 Prozent erreicht. Ein paar zusätzliche Zeilen reichen, damit der Code Sacred benutzt und jedes Experiment in einer Datenbank sichert. Mit dem so erweiterten Code wird es Ihnen leicht fallen, an den Hyperparametern des Netzwerks zu drehen, um mit etwas Glück eine noch etwas bessere Erkennungsrate zu erreichen.

Der Code importiert zuerst die Layer-Klassen und Hilfsfunktionen aus Keras (Zeile 1–7). In Zeile 9 bis 19 definiert er die Parameter, die die Struktur des Netzwerks und die Rahmenbedingungen des Trainings festlegen. Durch Ändern dieser Parameter lassen sich mit minimalen Änderungen am Quellcode verschiedenste Experimente starten. Der folgende Code lädt den Datensatz, bereitet ihn fürs Training vor (Zeile 21–26), baut das neuronale Netz auf (Zeile 28–44) und startet Training und Validierung (Zeile 46–54).

Um für diesen Code Sacred zu nutzen, erzeugen Sie zunächst ein `Experiment`:

```
from sacred import Experiment
ex = Experiment("MNIST-Convnet")
```

Die Klasse enthält die gesamte Logik, um Parameter über Sacred zu verwalten. Dem Konstruktor übergibt man einen Namen für das Experiment, über den man später die Protokolle der Durchläufe in der Datenbank findet.

Ein `Experiment`-Objekt enthält eine Methode `config`, die man als Dekorator (`@ex.config`) vor eine Funktion schreiben kann, die Parameter des Experiments definiert und berechnet. Diese Parameter kann man anschließend in anderen Funktionen nutzen, sofern sie mit `@ex.capture`, `@ex.main` oder `@ex.automain` dekoriert sind. Dem Beispielcode fehlt dafür nur die Funktion für die Konfiguration in Zeile 9:

```
@ex.config
def confnet_config():
  batch_size = 128
  epochs = 12
  # ...
  final_dropout = 0.5
```

Sacred baut sich aus den lokalen Variablen der `config`-Funktion ein Dictionary mit allen Parametern auf. Es kann dabei mit allen Werten umgehen, die sich als JSON darstellen lassen, also auch verschachtelte Listen und Dictionaries. Diese Parameter sichert Sacred nicht nur in seiner Datenbank, sondern übergibt sie auch automatisch an Funktionen mit den Dekoratoren `@ex.capture`, `@ex.main` oder `@ex.automain`. Eine mit `automain` dekorierte Funktion führt Sacred automatisch beim Start des Skripts aus, was das übliche `if __name__ == '__main__':` spart.

Konfigurationen nutzen

Möchte man Parameter nutzen, übergibt man sie einfach mit dem Namen an die Funktion, mit dem man sie auch in der Konfiguration festgelegt hat. Beim Aufruf der Funktion muss man diese Parameter nicht zwingend angeben, da Sacred sie aus der Konfiguration lädt. Gibt man sie trotzdem an, nimmt Sacred den explizit übergebenen Parameter statt des Standardwerts aus der Konfiguration und speichert den direkt übergebenen Wert auch in seiner Datenbank.

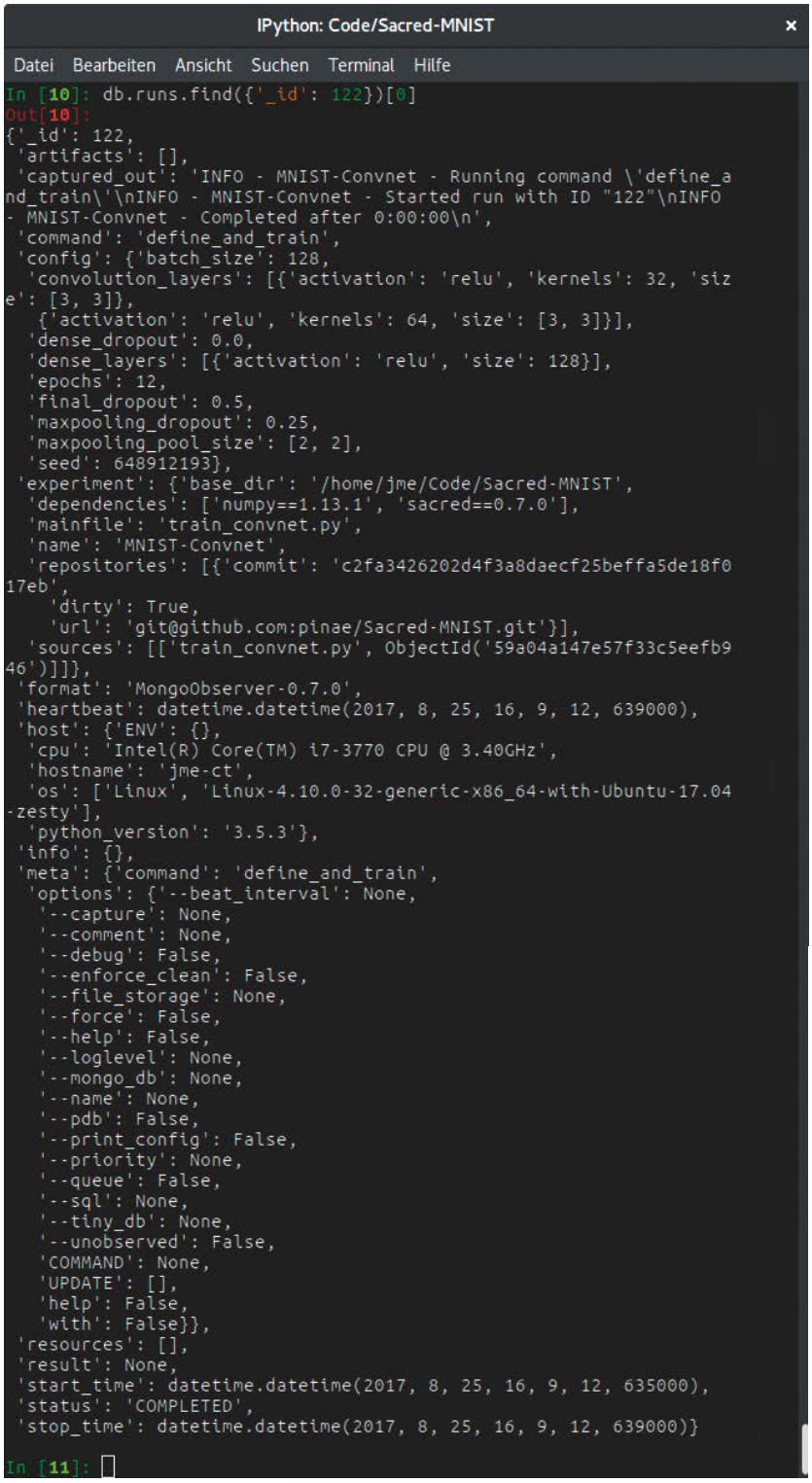
Damit der Beispielcode Sacred nutzt, reicht es also, die Netzwerkdefinition und das Training (alles ab Zeile 21) in eine mit `automain` dekorierte Funktion zu packen:

```
@ex.automain
def define_and_train(batch_size, ...
```

Durch die so dekorierte Funktion erhält der Code ohne weitere Arbeit ein mächtiges Kommandozeilen-

```python
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.losses import categorical_crossentropy
from tensorflow.keras.optimizers import Adadelta
from tensorflow.keras import backend as K

batch_size = 128
epochs = 12
convolution_layers = [
    {'kernels': 32, 'size': (3, 3), 'activation': 'relu'},
    {'kernels': 64, 'size': (3, 3), 'activation': 'relu'}
]
maxpooling_pool_size = (2, 2)
maxpooling_dropout = 0.25
dense_layers = [{'size': 128, 'activation': 'relu'}]
dense_dropout = 0.0
final_dropout = 0.5

(x_train, y_train), (x_test, y_test) = mnist.load_data()
input_shape = (1, 28, 28) if K.image_data_format() == 'channels_first' ↵
                  else (28, 28, 1)
x_train=x_train.reshape(x_train.shape[0],*input_shape).astype('float32')/255
x_test=x_test.reshape(x_test.shape[0],*input_shape).astype('float32')/255
y_train=to_categorical(y_train, 10)
y_test=to_categorical(y_test, 10)

model = Sequential()
model.add(Conv2D(convolution_layers[0]['kernels'],
                 kernel_size=convolution_layers[0]['size'],
                 activation=convolution_layers[0]['activation'],
                 input_shape=input_shape))
for layer in convolution_layers[1:]:
    model.add(Conv2D(layer['kernels'], kernel_size=layer['size'],
                 activation=layer['activation']))
model.add(MaxPooling2D(pool_size=maxpooling_pool_size))
model.add(Dropout(maxpooling_dropout))
model.add(Flatten())
for layer in dense_layers:
    model.add(Dense(layer['size'], activation=layer['activation']))
    if layer != dense_layers[-1]:
        model.add(Dropout(dense_dropout))
model.add(Dropout(final_dropout))
model.add(Dense(10, activation='softmax'))

model.compile(loss=categorical_crossentropy,
              optimizer=Adadelta(), metrics=['accuracy'])
model.fit(x_train, y_train,
          batch_size=batch_size, epochs=epochs, verbose=1,
          validation_data=(x_test, y_test))

score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Das „Hallo Welt“ der KI-Forschung erkennt handgeschriebene Ziffern. Da ein Durchlauf auf einem i5 39 Minuten dauert, lohnt es sich, stets die Parameter zu archivieren.



Interface. Hinter einem `with` kann man damit auf der Konsole jeden Parameter festsetzen, der auch in der Konfiguration steht. Beispielsweise:

```
python train_convnet.py with epochs=42
```

Ähnlich wie beim Übergeben von Parametern an die Funktion überschreiben die auf der Konsole angegebenen Parameter die Standardwerte aus der Konfiguration. Sacred registriert die überschriebenen Parameter und sichert sie zusammen mit den Ausgaben des Programms in der Datenbank.

Intern speichert Sacred die Konfiguration als Dictionary. Als solches können Sie sie auch übergeben, was ebenfalls die Standardwerte überschreibt. Um ein Experiment zu wiederholen, reicht es daher, die Konfiguration aus dem Eintrag in der Datenbank zu laden (beispielsweise mit PyMongo) und an ein `Experiment`-Objekt zu übergeben. Wir nutzen dieses Vorgehen beim Abarbeiten der Warteschlange am Ende des Artikels.

Experimentedatenbank

Sacred arbeitet mit verschiedenen Datenbank-Backends, um Parameter und Ergebnisse zu speichern. Den vollen Funktionsumfang erhält man aber nur mit MongoDB. Bei einer lokal installierten MongoDB ohne Authentifizierung, wie sie beispielsweise Ubuntu standardmäßig einrichtet, reicht es, dem Experiment einen `MongoObserver` hinzuzufügen (direkt hinter dem Erzeugen des `Experiment`):

In der MongoDB landet zu jedem Experiment ein strukturiertes Dokument mit automatisch erhobenen und manuell eingetragenen Informationen. Über den Schlüssel „config” lesen Sie sämtliche Parameter aus, mit denen das Experiment gestartet wurde. Unter „captured_out” sammeln sich die Konsolenausgaben des Experiments an, während es läuft. In dieser Zeit aktualisiert Sacred auch den Timestamp von „heartbeat” alle 10 Sekunden. Stürzt das Experiment ohne Fehlermeldung ab, informiert der Herzschlag zumindest, wann das passiert ist.

```
from sacred.observers ↵
                ↳import MongoObserver
ex.observers.append(
              MongoObserver())
```

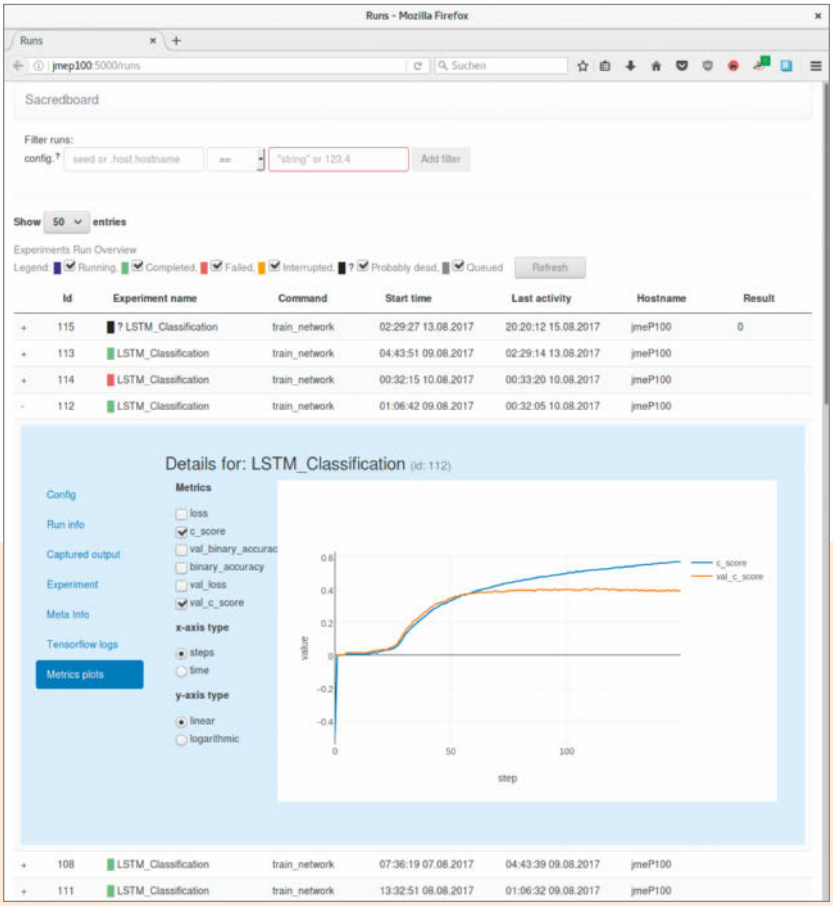
Sacred nutzt für die Datenbankverbindung PyMongo, das auch entfernte Datenbanken mit Authentifizierung ansprechen kann:

```
ex.observers.append(
   MongoObserver(url='mongodb: ↵
       ↳//user:password@example.com/ ↵
                     ↳the_database? ↵
          ↳authMechanism=SCRAM-SHA-1',
                    db_name='MY_DB'))
```

Für jedes gestartete Experiment erzeugt Sacred ein Dokument in der Datenbank. In `'config'` speichert Sacred die verwendeten Parameter, in `'captured_out'` die Konsolenausgabe. Dazu sammelt es automatisch zahlreiche Informationen über das verwendete System. Neben Start- und Endzeiten sind das Informationen zur CPU, zu GPUs, zum Betriebssystem, welche Version des Repository in Git ausgecheckt war, Dependencies der Installation, welche Python-Version den Code ausgeführt hat und wie der Befehl genau hieß.

Metriken speichern

Dazu kann man eigene Informationen abspeichern. Vorgesehen sind „Metrics“, Werte, die schon während der Laufzeit des Experiments Rückschlüsse auf den Erfolg erlauben. Beim Machine Learning wäre beispielsweise das Ergebnis der Loss-Funktion



Sacredboard zeigt die von Sacred archivierten Experimente übersichtlich an. Wer „Metrics“ gespeichert hat, bekommt schöne Diagramme dazu.

nach jedem Batch ein solcher Wert. Dazu kommen „Artifacts“, ganze Dateien wie die aktuell verwendeten Gewichte eines Netzwerks als hdf5-Datei. Die MongoDB speichert problemlos auch größere Dateien als Teil der Dokumente, ohne davon langsam zu werden.

Damit Keras die nötigen Werte preisgibt, definiert man Callbacks. Manche wie den `ModelCheckpoint`, der die aktuellen Gewichte des Netzwerks in eine Datei schreibt, bringt Keras bereits fertig mit. Man schreibt sie in die Liste der `callbacks`, die der `model.fit()`-Funktion übergeben werden:

```
from tensorflow.keras.callbacks import ↵
                    ↳ModelCheckpoint
model.fit(x_train, y_train,
          # ...
          callbacks=[ModelCheckpoint(
                      "weights.hdf5",
                   monitor='val_loss',
                 save_best_only=True,
             mode='auto', period=1)])
```

Ein Callback zum Speichern der aktuellen Genauigkeit des Netzwerks als „Metric“ in Sacred müssen Sie allerdings selbst schreiben:

```
from tensorflow.keras.callbacks import ↵
                              ↳Callback
class LogPerformance(Callback):
  def on_epoch_end(self, _, logs={}):
    log_performance(logs=logs)
```

Diese Klasse fügen Sie der Liste der Callbacks hinzu. Die eigentliche Arbeit geschieht in der Funktion `log_performance`:

```
@ex.capture
def log_performance(_run, logs):
  _run.add_artifact("weights.hdf5")
  _run.log_scalar("loss",
               float(logs.get('loss')))
  _run.log_scalar("accuracy",
            float(logs.get('accuracy')))
  _run.log_scalar("val_loss",
            float(logs.get('val_loss')))
  _run.log_scalar("val_accuracy",
       float(logs.get('val_accuracy')))
  _run.result = float(
              logs.get('val_accuracy'))
```

Die Funktion nutzt den Decorator `@ex.capture`, sodass ihr sämtliche Konfigurationsvariablen und auch Sacreds interne Variablen zur Verfügung stehen. Die Variable `_run` steht jeder per Sacred-Dekorator eingefangenen Funktion zur Verfügung, solange das Experiment läuft. Ihre Methoden speichern die vom Callback empfangenen Werte in der Datenbank.

Damit speichert Sacred zwar alle Werte, die während des Trainings anfallen, aber kein Gesamtergebnis. Das sichern Sie nämlich besonders einfach, indem sie es in der mit `@ex.automain` dekorierten Funktion zurückgeben:

```
return score[1]
```

Mit dieser Struktur stellt Sacred sicher, dass kein Experiment zu verschwendeter Rechenzeit wird. Jeder Lauf landet mit allen Parametern, die nötig sind, um ihn zu wiederholen, in der Datenbank.

Sacred versorgt sämtliche Zufallszahlengeneratoren so mit Seeds, dass der Rechner bei einem zweiten Durchlauf wieder dasselbe Ergebnis berechnet. Aber auch Zwischenergebnisse und verwendete Dateien sichert Sacred, sodass Sie im Normalfall ohne neue Rechenzeit an frühere Ergebnisse kommen. Ein Repository mit dem kompletten Quellcode finden Sie über ct.de/wr9z.

Sacredboard

Eine Datenbank voller wertvoller Zwischenergebnisse nutzt nicht viel, wenn man nicht auch leicht an die archivierten Dokumente herankommt. Wer nicht fließend MongoDB spricht, wünscht sich eine grafische Übersicht. Genau die liefert Sacredboard, eine Webanwendung, die die Dokumente zu den archivierten Experimenten aus der lokalen MongoDB fischt und über einen kleinen Webserver darstellt. Sacredboard installieren Sie ganz leicht mit pip:

```
pip install sacredboard
```

Den Webserver starten Sie, indem Sie Sacredboard den Namen der verwendeten Collection (standardmäßig „sacred“) mitgeben:

```
sacredboard -m sacred
```

Der Befehl öffnet direkt ein Browserfenster mit dem Webinterface auf Port 5000. Sacredboard zeigt standardmäßig alle Experimente in der Datenbank in chronologischer Reihenfolge an. Sie können die Liste aber auch nach anderen Kriterien sortieren und filtern.

Ein Klick auf das „+“ zu Beginn jeder Zeile öffnet die Detailansicht des Experiments. Die listet in mehreren Tabs die Konfiguration und die Ausgaben.

„Metrics plots“ zeichnet Diagramme zu den als Metrik gespeicherten Werten. Einen Zugriff auf die Artefakte erlaubt die aktuelle Version (0.4.2) von Sacredboard noch nicht.

Experiment-Warteschlange

Über die Option `-q` auf der Kommandozeile trägt Sacred auch Experimente in seine Datenbank ein, die es nicht sofort ausführt. Das erzeugt aber bisher nur ein Dokument für das Experiment mit `'status': 'QUEUED'`. Sacred bringt leider noch keine Funktion mit, um die Experimente aus der Warteschlange auch auszuführen.

Um unseren GPU-Server auch übers Wochenende mit Experimenten auszulasten, haben wir für den Artikel auf Seite 90 ein Python-Skript geschrieben, das die Warteschlange automatisch abarbeitet. Sie finden den Link zu dem Skript auch über ct.de/wr9z.

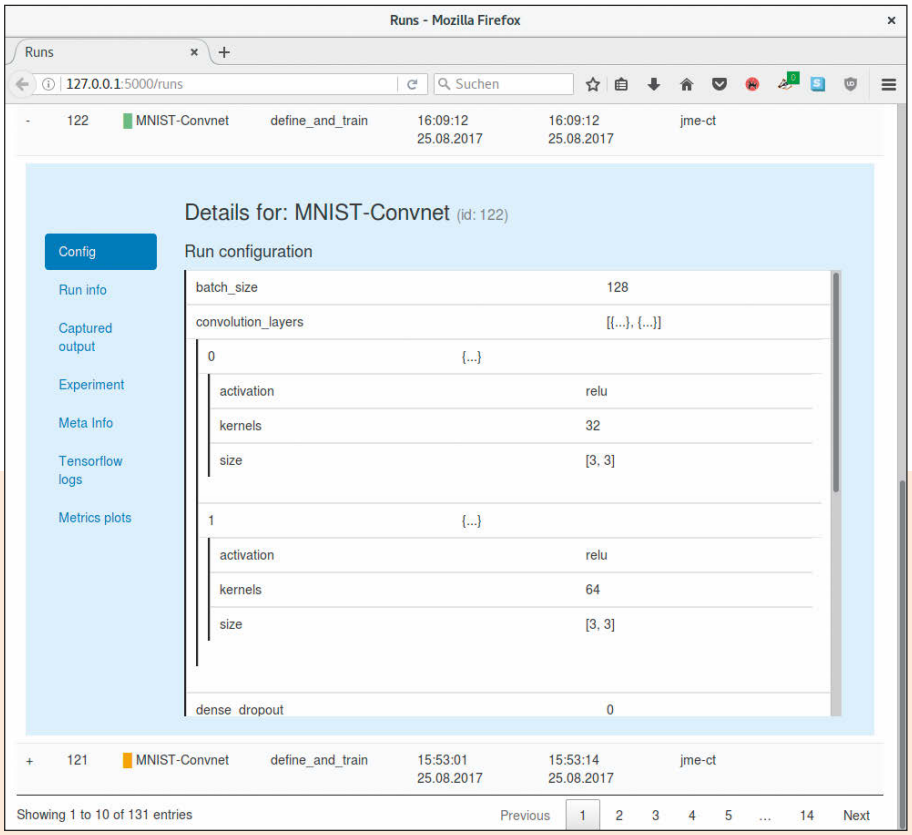
Die Idee ist einfach: Das Skript fragt die Datenbank, wann immer es nichts zu tun hat, nach einem Experiment aus der Warteschlange:

```
from time import sleep
def main_loop():
  while True:
    check_for_work()
    sleep(10)
```

Findet es nichts, wartet es 10 Sekunden und versucht es erneut.

Zum Abfragen der Datenbank nutzt das Skript `pymongo`:

```
from pymongo import MongoClient
client = MongoClient()
db = client.sacred

# ...
def check_for_work():
```



Enthält die Konfiguration verschachtelte Listen und Dictionaries, zeigt Sacredboard diese nach dem Anklicken eingerückt an.

```
try:
  queued_run = db.runs.find(
            {'status': 'QUEUED'})[0]
except IndexError:
  return None
config = queued_run['config']
db.runs.delete_one(
          {'_id': queued_run['_id']})
start_experiment(config)
```

Wenn `find()` ein Dokument findet, extrahiert das Skript die Konfiguration und löscht das Dokument aus der Warteschlange. Mit der gespeicherten Konfiguration startet es dann regulär ein neues Experiment:

```
def start_experiment(config):
  from train_convnet import ex
  run = ex.run(config_updates=config)
```

Jedes Mal, wenn man ein anderes Experiment starten möchte, muss man das Skript anpassen, um das richtige Experiment zu importieren. Sollten Sie häufiger mit mehreren Experimenten aus verschiedenen Modulen experimentieren, könnten Sie die Zuordnung auch aus der Konfiguration laden.

Mit Sacred ging bei unserem LSTM-Projekt auf Seite 90 kein Zwischenergebnis mehr verloren. Dank des Queue-Managers starteten wir für den Artikel über 150 Experimente auf drei verschiedenen Rechnern. Auf dem GPU-Server liefen oft zwei Experimente parallel, um sowohl GPU als auch CPU voll auszulasten. Sacredboard erwies sich als äußerst nützlich, um die Ergebnisse auszuwerten. Wie auch bei früheren Experimenten mit neuronalen Netzen erlebt, entschieden oft schon kleine Änderungen an den Parametern darüber, ob das Training funktionierte oder ob sich die Loss-Funktion auf hohen Werten einpendelte. Eine systematische Übersicht darüber, was bereits funktioniert hatte, war dabei Gold wert. Ohne Sacred hätte uns die Disziplin gefehlt, all diese Experimente zu archivieren. (pmk) c't

Repository, Dokumentation:
www.ct.de/wr9z

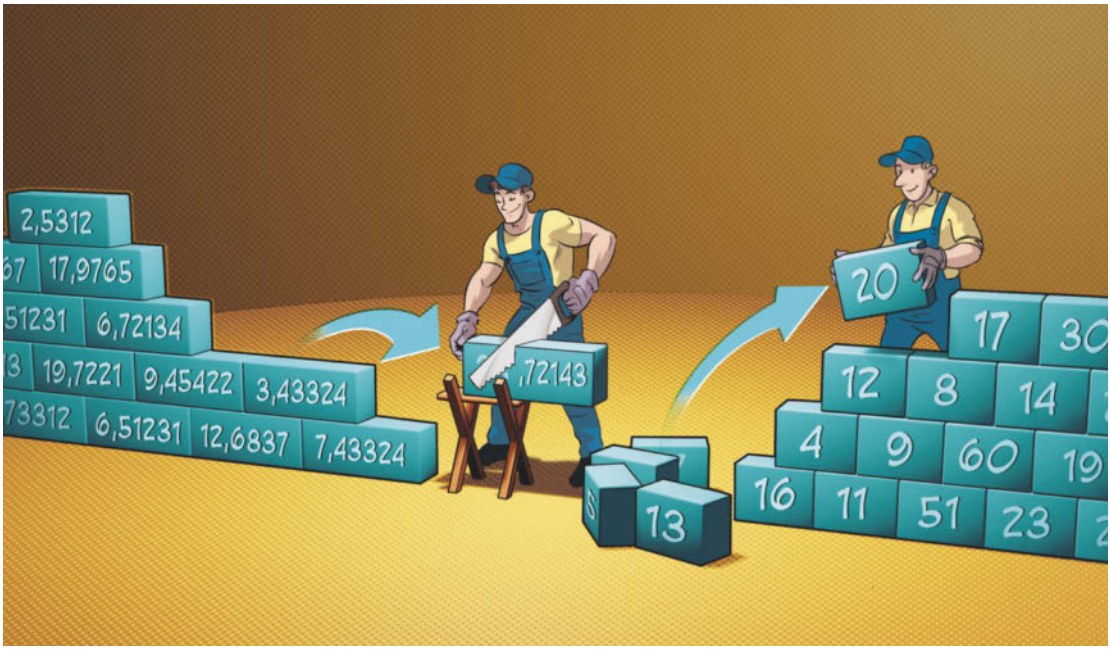


Bild: Albert Hulm

8-Bit-KI mit TensorFlow-Lite

Neuronale Netze rechnen mit Millionen von 32 Bit langen Zahlen. Baut man sie so um, dass sie stattdessen mit 8-Bit-Zahlen rechnen, spart das drei Viertel des Speicherplatzes und sie begnügen sich mit kleineren Rechenwerken im Chip. Mit den richtigen Parametern erledigt Googles Framework TensorFlow diesen Umbau vollautomatisch.

Von Pina Merkert

Neuronale Netze rechnen üblicherweise in Gleitkommazahlen mit 32 Bit. Die Gewichte der Synapsen, mit denen das neuronale Netz Schicht für Schicht seine Eingabedaten multipliziert, nehmen dabei oft sehr kleine positive und negative Werte an, die eine Gleitkommazahl dank ihres Exponenten mit hoher Genauigkeit kodiert. Zu Beginn des Trainings initialisieren viele KI-Forscher ihre Netze bewusst mit Zufallszahlen nahe 0. Der Trainingsalgorithmus passt dann diese Zahlen so an, dass die Ausgaben des Netzes besser zu den Trainingsdaten passen, geht dabei aber in sehr kleinen Schritten vor. Sollte ein solcher Schritt kleiner ausfallen, als die Gleitkommazahl für dieses Gewicht kodieren kann, lernt das Netz an dieser Stelle nichts.

Mit diesem Wissen überrascht es, dass KI-Beschleuniger-Chips wie der Myriad in Intels Neural Compute Stick (fp16) oder Googles Coral gar nicht mit langen Gleitkommazahlen rechnen können. Coral bringt nur Rechenwerke für Integer mit lediglich 8 Bit mit. Integer-Rechenwerke brauchen wesentlich weniger Transistoren, sodass die Hersteller mehr davon auf gleicher Fläche unterbringen. Das spart nicht nur bei den Produktionskosten, sondern vor allem beim Stromverbrauch: Die kleinen KI-Chips begnügen sich mit weniger als zwei Watt, sollen aber so viele Daten wie eine Grafikkarte verarbeiten.

Damit ein neuronales Netz auf einer solch effizienten Hardware läuft, muss es eine Transformation durchlaufen, bei der aus langen Gleitkommazahlen mit unterschiedlicher Genauigkeit über den darstellbaren Wertebereich (Gleitkommazahlen bilden den Zahlenbereich um den Nullpunkt viel feiner ab als sehr große oder kleine Werte) kurze Ganzzahlen mit gleichbleibender Genauigkeit in einem begrenzten Wertebereich werden. Dieses Runden der bereits trainierten Gewichte eines neuronalen Netzes nennt man „Quantisieren“. Dieser Artikel erklärt, wie Sie ein Keras-`Model` für TensorFlow-Lite quantisieren.

Mehr Rechnen für weniger Bit

Mit einfachem Runden ist es beim Quantisieren nicht getan. Neuronen üblicher Netze mit Gleit-

TensorFlow 2 und OpenCV

TensorFlow 2 vereinfacht die Arbeit mit neuronalen Netzen durch zwei Neuerungen: Zum einen ist es nun nicht mehr nötig, den Berechnungsgraphen völlig abstrakt zu definieren und dann dessen Variablen in einer `Session` mit Werten zu befüllen und auszuführen. Stattdessen rechnet TensorFlow jetzt sofort Werte aus, sobald man sie benutzt. Das Framework verhält sich damit mehr wie ganz normaler Python-Code und Entwickler müssen weniger umdenken.

Zum anderen implementiert TensorFlow das Keras-API nun selbst und unterstützt das High-Level-Framework als Standardschnittstelle, um neuronale Netze zu definieren. Deswegen ist es bei TensorFlow 2.0 nicht mehr nötig, die Keras-Referenzimplementierung als eigenes Paket zu installieren. Stattdessen kommt man mit `tensorflow.keras` an alles, was die Keras-Dokumentation auflistet, einschließlich vortrainierter Netze wie InceptionV3.

Wie immer bietet es sich an, TensorFlow in eine virtuelle Umgebung zu installieren. Folgende zwei Befehle legen eine solche Umgebung in einer Shell an und aktivieren sie:

```
python3.7 -m venv env
source env/bin/activate
```

Python 3.7, da TensorFlow mit der Version 3.8 noch nicht kompatibel ist. Das funktioniert zwar auch unter Windows, es ist aber leichter auf Anaconda zu setzen und im Anaconda-Navigator grafisch eine frische virtuelle Umgebung einzurichten. In der installieren Sie mit `conda install pip` den Python-Paketmanager, mit dem die folgenden Befehle funktionieren.

Prüfen Sie auf allen Betriebssystemen zuerst, dass `pip` und `wheel` auf aktuellem Stand sind:

```
pip install -U pip wheel
```

Danach installieren Sie TensorFlow 2 zusammen mit einer etwas aktuelleren Version von `setuptools`:

```
pip install setuptools==46.0.0
pip install tensorflow==2.1.0
```

Für den Zugriff auf die Webcam fehlt dann nur noch OpenCV:

```
pip install opencv-python
```

kommazahlen berechnen meist Aktivierungen im Bereich zwischen –1 und 1. Frameworks wie TensorFlow multiplizieren die Werte beispielsweise mit dem Faktor 128, um den Wertebereich von –1 bis 1 auf –128 bis 127 zu strecken und damit die Genauigkeit eines Integers mit 8 Bit voll auszunutzen. Statt einer linearen Transformation kann das Framework aber auch eine logarithmische Skala verwenden.

Fällt Tensorflow dabei auf, dass es nach einer Neuronenschicht eine Transformation rückgängig macht, die es für die nächste Schicht gleich wieder anwendet, kürzt es die beiden eingefügten Berechnungsschritte kurzerhand wieder aus dem Berechnungsgraphen.

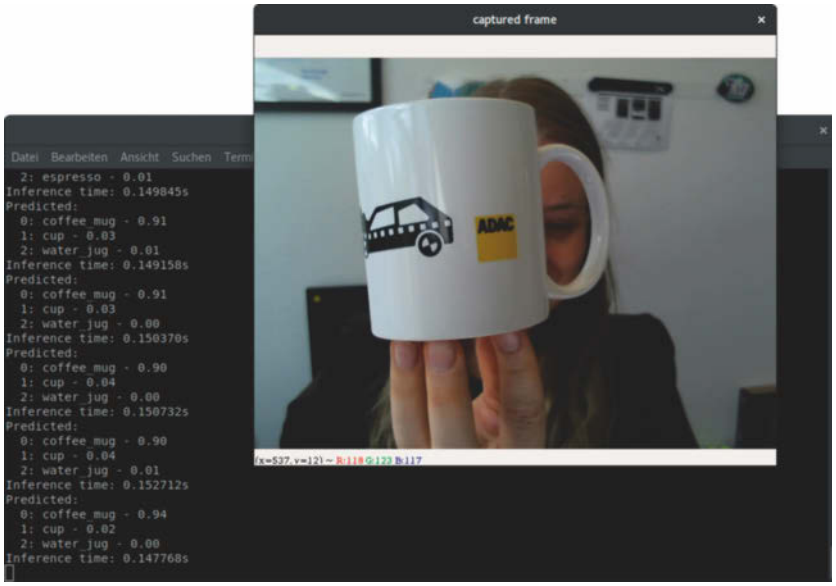
Diese komplexen Optimierungen nimmt das Framework vollautomatisch vor. Anwender legen lediglich fest, ob das erzeugte Netz nur die Gewichte mit 8 Bit darstellt oder auch für Ein- und Ausgaben mit kurzen Zahlen rechnet. Für die KI-Chips muss das Framework das komplette Netz quantisieren.

Damit TensorFlow dabei die begrenzten Wertebereiche der kurzen Zahlen effizient nutzt, will es fürs Quantisieren mit einigen typischen Eingabedaten gefüttert werden. Es braucht aber nur wenige hundert Beispiele dafür und keinen so großen Datensatz wie fürs Training. Außerdem nutzt es nur Eingabedaten, sodass die passenden Labels fehlen dürfen.

Wir zeigen an einem Giganten der Bilderkennung, wie das Quantisieren mit TensorFlow 2.0 in der Praxis funktioniert. Unser Beispiel lädt ein sehr tiefes Convolutional Network namens „InceptionV3“ aus den bei Keras mitgelieferten Beispielen. TensorFlow setzt seit Version 2 das Keras-API in eigenem Code um, sodass man Keras nicht extra installieren muss.

InceptionV3

Mit InceptionV3 gewannen Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe und Jonathon Shlens von Google sowie Zbigniew Wojna vom University College London 2015 die ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [1]. Bei dem Wettbewerb trainieren die Teilnehmer ihre KIs mit einem Datensatz namens ImageNet. ImageNet enthält inzwischen 14 Millionen Bilder und für jedes Bild eine Information, was dort zu sehen ist. Die KIs ordnen ein Bild einer von 1000 möglichen Klassen zu und werden daran gemessen, wie oft sie mit dieser Klassifikation richtig liegen. InceptionV3 gibt für 78 Prozent der Bilder die richtige Antwort.



Test mit Webcam: InceptionV3 ist sich zu mehr als 90 % sicher, dass hier eine Kaffeetasse zu sehen ist.

Um InceptionV3 selbst zu trainieren, müssten Sie gigabyteweise Bilder herunterladen und wochenlang den Rechner durcharbeiten lassen. Und das geht auch nur so schnell, wenn Sie eine mächtige Grafikkarte mit CUDA-Unterstützung haben. Auf einer normalen CPU dauert es noch mindestens zehnmal länger. Aber glücklicherweise bringt Keras die Zahlen eines fertig trainierten Netzwerks gleich mit. Das Framework erzeugt dafür ein Keras-`Model` und lädt seine 23.851.784 Parameter komplett automatisch. Einen niederschwelligen Einstieg, wie man mit einem solchen Keras-`Model` umgeht (siehe Seite 62). Um die Parameter zu laden, müssen Sie nur den Parameter `weights='imagenet'` angeben:

```
from tensorflow.keras.applications.↵
    ↳inception_v3 import InceptionV3
model = InceptionV3(
 include_top=True, weights='imagenet')
```

`include_top=True` sorgt dafür, dass das Netzwerk mitsamt seiner obersten Schicht im Speicher landet. Mit der obersten Schicht schätzt es Wahrscheinlichkeiten, welche der 1000 Klassen im Eingabebild zu sehen sind.

Ein anschaulicher Test für das so geladene Netzwerk besteht darin, mit OpenCV von einer Webcam ein Bild abzugreifen, die Bilddaten für das Netzwerk ins richtige Format zu bringen und die KI auf die Daten loszulassen:

```
from tensorflow.keras.applications.↵
            ↳inception_v3 import preprocess_input
import numpy as np
import cv2
capture_device = cv2.VideoCapture(0)
_, i = self.capture_device.read()
i = cv2.resize(image, dsize=(299,299),
         interpolation=cv2.INTER_AREA)
i = np.expand_dims(i, axis=0)
i = preprocess_input(i)
predictions = model.predict(i)
```

`cv2.VideoCapture(0)` erlaubt den Zugriff auf die erste Webcam. In Notebooks ist das üblicherweise die eingebaute Kamera. Die Methode `read()` liest ein Bild von der Kamera und legt es in einem Format im Speicher ab, das Numpy direkt weiterverarbeiten kann. Das gut optimierte Framework Numpy ist der Quasi-Standard in der Python-Welt, um mit großen Vektoren, Matrizen und Arrays zu rechnen.

InceptionV3 verarbeitet nur Bilder mit einer festen Größe von 299 × 299 Pixeln. `cv2.resize()` verkleinert das Bild auf diese Größe. Da das beim Training viel Zeit spart, Berechnet das Netz immer einen ganzen Schwung Bilder („Batch“) auf einmal. Es erwartet daher als Eingabe ein Array mit den Dimensionen Batchgröße × Bildbreite × Bildhöhe × Farbkanäle. Da die Webcam nur ein einzelnes Bild liefert, fügt `np.expand_dims(i, axis=0)` dieses in einen Batch aus einem Bild ein.

Bei `preprocess_input()` hilft Keras noch einmal mit, die Eingabedaten tatsächlich ins richtige Format zu bringen: InceptionV3 erwartet nämlich Farbwerte der Pixel als 32-Bit-Gleitkommazahlen, die auf einen Wertebereich von –1 bis 1 normiert sind, wobei die durchschnittliche Helligkeit der Pixel auf dem Nullpunkt liegen soll. Da solche Rahmenbedingungen zwischen verschiedenen Netzwerkstrukturen variieren, bringt das Modul `keras.applications` für jedes der vortrainierten Netze auch jeweils eine `preprocess_input()`-Funktion mit.

Die eigentliche Arbeit des neuronalen Netzes passiert in `model.predict()`. In der Variable `predictions` landet danach ein Array aus 1000 Zahlen zwischen 0 und 1, die sich wie Wahrscheinlichkeiten lesen. `predictions[12]` gibt also die von InceptionV3 geschätzte Wahrscheinlichkeit an, dass ImageNet-Klasse Nummer 13 im Bild zu sehen ist. Da das für Menschen wenig übersichtlich ist, bringt Keras die Funktion `decode_predictions()` mit, die nicht nur Labels in Textform dazu schreibt, sondern auch die Liste auf die in diesem Beispiel wahrscheinlichsten drei Klassen zusammenkürzt:

```
decode_predictions(predictions,
                   top=3)[0]
```

Weil auch diese Funktion mit Batches arbeitet, muss man mit `[0]` am Ende angeben, dass nur der erste und einzige Eintrag im Batch interessiert.

Schlankheitskur

InceptionV3 kam in diesem Beispiel nur als Keras-`Model` vor. Solch ein Objekt arbeitet aber nur zusammen mit einer kompletten TensorFlow-Installation. Für effizientes Inferencing, also das reine Berechnen von Ausgaben des neuronalen Netzes, ohne es dabei noch zu trainieren, stellt Google eine abgespeckte Version seines Frameworks namens „TensorFlow-Lite“ bereit. TensorFlow-Lite nutzt ein effizientes Datenformat zum Speichern der Modelle, erwartet aber, dass man sie in dieses Format überführt.

Ums Konvertieren für die Lite-Version kümmert sich der `tensorflow.lite.TFLiteConverter`. Er bringt die Factory-Methode `from_keras_model()`, die direkt ein Keras-`Model` konsumiert. Die Methode `convert()` stößt dann die Konvertierung an und gibt die Binärdaten so zurück, dass Python sie einfach in eine Datei schreiben kann:

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter↵
          ↳.from_keras_model(model)
tflite_model = converter.convert()
with open("incepV3.tflite",
          'bw') as model_file:
  model_file.write(tflite_model)
```

Der `TFLiteConverter` setzt automatisch überall dort am internen Berechnungsgraphen des Netzwerks das Messer an, wo das Netz Rechenoperationen nur fürs Training besitzt. So fällt `convert()` die Loss-Funktion zum Opfer, genau wie der komplette Gradientenabstiegsalgorithmus samt der Ableitung für die Backpropagation. Übrig bleibt der entkleidete Forward-Pass (das, was `model.predict()` berechnet hat) von InceptionV3, also nur die Berechnungen, um aus Bildern Wahrscheinlichkeiten für die 1000 Klassen zu schätzen.

TensorFlow-Lite mit Float32

Um mit dem für TensorFlow-Lite gespeicherten Netz zu rechnen, kann man beispielsweise das Python-API des schlanken Frameworks verwenden. Dieses lädt das Netz zunächst in ein `Interpreter`-Objekt und belegt den nötigen Speicher für Eingaben, Aktivierungen und Ausgaben:

```
incepV3 = tf.lite.Interpreter(
    model_path="incepV3.tflite")
incepV3.allocate_tensors()
```

Auch für TensorFlow-Lite müssen die Bilddaten zunächst ins richtige Format umgerechnet werden, was aber ganz genauso wie beim Keras-`Model` funktioniert. Lediglich der Aufruf fürs Inferencing, also das Berechnen der Ausgabe des Netzes, sieht in TensorFlow-Lite etwas anders aus:

```
in_det = incepV3.get_input_details()
incepV3.set_tensor(
    in_det[0]["index"], i)
incepV3.invoke()
out_det = incepV3.get_output_details()
predictions = incepV3.get_tensor(
    out_det[0]["index"])
```

Die Schlankheitskur spart in dieser Form aber nur wenige Millisekunden, da auch TensorFlow-Lite mit den gleichen Gleitkommazahlen mit 32 Bit rechnen muss. Wirklich kleiner wird das Netzwerk erst, wenn man dem `TFLiteConverter` Anweisung gibt, die Zahlen zu kürzen.

Größe vierteln

Setzt man diverse Eigenschaften des `TFLiteConverter`, baut dieser den Berechnungsgraphen beim Konvertieren für TensorFlow-Lite um. Beispielsweise enthält `optimizations` eine Liste gewünschter Optimierungsstrategien. Setzt man `supported_ops` auf `[tensorflow.lite.OpsSet.TFLITE_BUILTINS_INT8]`, baut der Konverter alle Rechenoperationen auf 8-Bit-Integer um und quantisiert dabei auch Gewichte und Biases. Ein- und Ausgaben würden dabei aber zunächst bei langen Gleitkommazahlen bleiben, damit das Netz sich nach außen gleich verhält. Um es komplett auf 8 Bit umzubauen, muss man `inference_input_type` und `inference_output_type` auf `tensorflow.uint8` setzen:

```
conv = tf.lite.TFLiteConverter␣
          ↳.from_keras_model(model)
conv.optimizations = [
  tf.lite.Optimize.DEFAULT]
conv.target_spec.supported_ops = [
  tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
conv.inference_input_type = tf.uint8
conv.inference_output_type = tf.uint8
conv.representative_dataset = tf.␣
          ↳lite.RepresentativeDataset(
    input_gen=get_representative_data)
tflite_quantized_mdl = conv.convert()
with open("quant.tflite", 'bw') as f:
  f.write(tflite_quantized_mdl)
```

Damit der Konverter beim Quantisieren mit den richtigen Wertebereichen rechnet, versorgt man ihn mit einem `representative_dataset`. Das nutzt die selbst geschriebene Funktion `get_representative_data()`. Sie liefert eine Handvoll Bilder aus dem für ImageNet üblicherweise genutzten Validierungsdatensatz ILSVRC2012. Dafür bedient sie sich eines TensorFlow-Datengenerators (siehe Kasten).

Optimierungsnotstand

Mit diesen Parametern quantisiert der `TFLiteConverter` das Netz und spuckt eine Binärdatei mit einem Viertel der Größe aus. Das Inferencing funktioniert wie beim vorher exportierten Netz mit einem `Interpreter`. Im Grunde tauschen Sie nur den Dateinamen aus (fertige Skripte, um beide Varianten zu testen, finden Sie im Repository zum Artikel bei GitHub über ct.de/wx6w).

Benchmarkt man die beiden Varianten jedoch auf einem x86-Prozessor, kommt es zu einer großen Überraschung: Das quantisierte Netz rechnet fast 150 Mal langsamer und nutzt nur einen Prozessorkern. Statt 0,1 Sekunden pro Bild bei 32 Bit verschwendet es mehr als 16 Sekunden. Die Genauigkeit bleibt bei InceptionV3 zwar praktisch gleich, andere Netze können durchs Quantisieren aber durchaus ungenauere Ergebnisse produzieren. Man sollte ein quantisiertes Netz daher immer mit einem Testdatensatz validieren.

Im Bugtracker zu TensorFlow finden sich bereits Issues, in denen Nutzer fragen, warum ihre quantisierten Netze langsamer laufen als die großzahligen Vorlagen. Google-Entwickler antworteten darauf etwas nebulös, dass es wohl sein könne, dass man bisher manche Instruktionen für quantisierte Netze noch nicht für x86 optimiert habe. Stattdessen habe man sich auf die ARM-Architektur konzentriert.

Literatur

[1] Szegedy et al., **Rethinking the Inception Architecture for Computer Vision,** 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 2015, S. 2818

[2] Pina Merkert, **Bilderstürmer,** Mit Keras effizient Trainingsdaten fürs maschinelle Lernen generieren, c't 16/2018, S. 174

Quellcode bei GitHub, Paper zum Download:
www.ct.de/wx6w

Das Beispiel zeigt, wie wichtig es ist, beim maschinellen Lernen stetig zu benchmarken und zu validieren. Fehlt eine Softwareoptimierung, bricht die Performance schnell um zwei Größenordnungen ein. Wer auf der CPU oder mit CUDA trainiert, will bis auf Weiteres bei 32-Bit bleiben. Rundet das Framework beim Quantisieren ungünstig, gehen schnell mal ein oder zwei Prozent Genauigkeit flöten. Glücklicherweise braucht man nur ein paar Zeilen Code, um die Varianten auszuprobieren. (pmk) c't

Datengeneratoren in TensorFlow 2.0

Noch einfacher als mit Keras-Datengeneratoren [2] versorgt eine selbst geschriebene Funktion im Zusammenspiel mit einem `Dataset`-Objekt aus TensorFlow 2.0 ein neuronales Netz mit Daten. Eine Implementierung für den Validierungsdatensatz ILSVRC2012 mit zu ImageNet passenden Bildern und Labels zeigt anschaulich, wie man einen solchen Datengenerator aufbaut.

ILSVRC2012 besteht aus 50.000 Bildern im JPEG-Format, deren Dateinamen eine fortlaufende Nummer enthalten. Die Labels stehen in einer Textdatei, bei der die erste Zeile das dezimal dargestellte Label für das erste Bild enthält, die zweite Zeile das für das zweite Bild und so weiter. Eine Schleife läuft über die ganze Liste an Bildern und liefert mit der `yield`-Anweisung Datensätze aus fürs Netzwerk skalierten Bilddaten als Numpy-Array zusammen mit dem aus der Textdatei gelesenen Label:

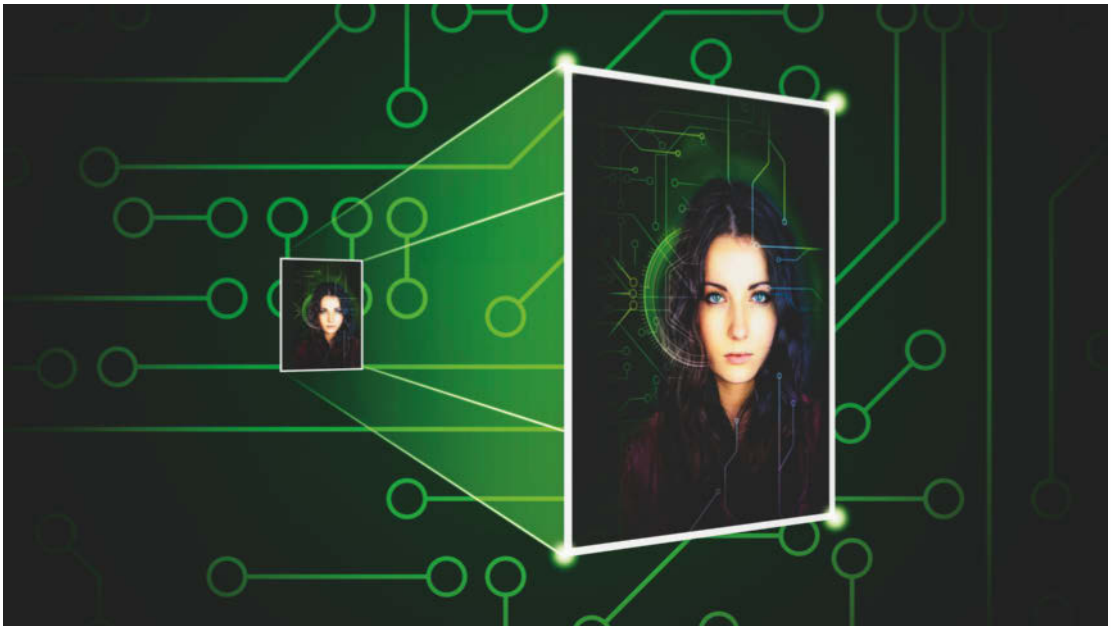
```
def data_generator():
  i = 0
  dataset_dir = os.path.join(
          os.path.curdir, "..", "..",
          "Datasets", "ILSVRC2012")
  with open(os.path.join(dataset_dir,
      "ILSVRC2012_validation_↵
      ↳ground_truth.txt"), 'r') as f:
      labels = f.readlines()
  while i < len(labels):
    img_filename = "ILSVRC2012_val_↵
            ↳000%05d.JPEG" % (i + 1)
    img_path = os.path.join(
                      dataset_dir,
                      "images",
                      img_filename)
    img = cv2.imread(img_path,
         cv2.IMREAD_COLOR)
    img = cv2.resize(img,
        dsize=(299, 299),
        interpolation=cv2.INTER_AREA)
    label = int(labels[i])
    yield (img, label)
    i += 1
```

Die Funktion enthält lediglich die Operationen, die nötig sind, um die Daten von der Festplatte zu laden und für das neuronale Netz vorzubereiten. Optimierungen für ein effizientes Training wie Batches kommen hier noch nicht vor. Um die kümmert sich nämlich TensorFlows `Dataset`-Klasse. Die folgende Zeile erzeugt aus der Generatorfunktion eine `Dataset`-Instanz:

```
def get_dataset():
  return Dataset.from_generator(
    data_generator,
    output_types=(uint8, uint32),
    output_shapes=(
          TensorShape([299, 299, 3]),
          TensorShape([])))
```

Die Factory-Methode `from_generator()` produziert aus der einfachen Funktion ein vollwertiges `Dataset`. Dafür verlangt sie nach lediglich zwei weiteren Infos: Wie sehen die Ausgabedaten aus und welche Datentypen haben sie? Die hier angegebenen Tupels besagen: Die Bilddaten bestehen aus drei 8-Bit-Integern (Helligkeit von Rot, Grün und Blau) in einem Array aus 299 × 299 Pixeln. Die Labels bestehen aus einem einzelnen Integer mit 32 Bit, der als Skalar keine Größe für ein Array angeben muss (`[]`).

Das so erzeugte `Dataset` kann so einiges: Beispielsweise liefert `.take(12)` die ersten zwölf Paare aus Bild und Label. Die Methode `.batch(128)` fasst jeweils 128 Bild-Label-Paare zu einem Batch zusammen. Die Funktionen lassen sich kombinieren. Beispielsweise liefert `.batch(64).take(3)` die ersten drei Batches mit je 64 Datensätzen.



Bilder skalieren mit TensorFlow 1

Googles Bibliothek TensorFlow eignet sich als Universalwerkzeug für Experimente mit neuronalen Netzen. Wir haben damit ein Netz trainiert, das die Auflösung von Bildern verbessert. Das Netz erzeugt sichtbar schärfere Bilder als die Standardskalierung in Grafikprogrammen, da es Formen im Eingabebild erkennt und fehlende Pixel sinnvoll ergänzt.

Von Pina Merkert

Auch wenn neuronale Netze meist in der Cloud gerechnet werden: Die nötige Software ist Open Source und läuft auf ganz normalen PCs. Die Frameworks nutzen dabei zwei grundsätzlich unterschiedliche Methoden. Die erste verdrahtet vordefinierte Neuronenschichten miteinander, für die die Entwickler sowohl die Berechnung der Neuronenaktivität (Forward Pass) als auch die Ableitung (Backward Pass) im Code definiert haben. Die Optimierungsalgorithmen, die während des Trainings die Gewichte an den Synapsen herausfinden, nutzen die Ableitung der Fitness-Funktion. Die zweite Methode erlaubt völlig frei, einen Forward Pass als Formel zu definieren, und berechnet die Ableitung automatisch. So arbeitet Googles TensorFlow und ermöglicht daher beliebige Experimente mit neuronalen Netzen. Im Prinzip eignet sich das Framework für jedes Problem, bei dem ein

Installation

TensorFlow ist im Python Package Index enthalten und lässt sich einfach mit pip installieren:

```
pip install tensorflow
```

Der Befehl installiert eine platzsparende Version ohne CUDA-Unterstützung. Wer eine moderne Grafikkarte von Nvidia besitzt, installiert lieber tensorflow-gpu – eine Version, die sowohl auf der CPU als auch auf der GPU rechnen kann. TensorFlow nutzt die Grafikkarte automatisch für Operationen, für die eine CUDA-Implementierung verfügbar ist. Das Framework stellt Befehle zur Verfügung, um Rechenarbeit auf die CPU oder bestimmte Grafikkarten zu verteilen. Auf einem PC mit nicht mehr als einer Grafikkarte müssen Sie im Code jedoch praktischerweise nichts davon einstellen.

Neben der Grafikkarte brauchen Sie für den CUDA-Betrieb einen aktuellen Nvidia-Treiber, CUDA selbst und optional noch CUDNN. CUDNN ist eine kostenlose Bibliothek, die typische Berechnungen für neuronale Netze deutlich beschleunigt, die Nvidia aber erst herausrückt, nachdem Sie einen kostenlosen Entwickleraccount erstellt haben.

Der hier gezeigte Code nutzt Befehle aus TensorFlow 1.x. Die aktuelle Version 2.1 unterstützt die aber mit einer Kompatibilitätsschicht. Leider läuft TensorFlow bisher nur mit Python bis Version 3.7. Python 3.8 wird aktuell noch nicht unterstützt.

Optimierungsalgorithmus automatisch aus Trainingsdaten die Parameter einer Formel schlussfolgern soll.

Die Formeln definiert man in TensorFlow als ganz normalen Quellcode. Das Framework hat Bindings für Python, C++, Go und Java. Wir haben das Python-API verwendet, da es den vollen Funktionsumfang abdeckt und bei TensorFlow-Projekten auf GitHub mit Abstand am häufigsten eingesetzt wird.

TensorFlow unterstützt seit Version 2.0 „eager execution“, also das Ausführen von Berechnungen, sobald die beteiligten Werte zur Verfügung stehen. Das ist praktisch und erleichtert den Einstieg für Python-Programmierer mit Numpy-Erfahrung. Wie TensorFlow die Berechnungen intern ausführt, zeigt die Kompatibilitätsschicht zu Version 1 noch deutlicher. Deswegen zeigen wir bei diesem Beispiel, wie die Berechnungen mit explizit definierten Graphen und Sessions funktionieren.

Der Python-Code

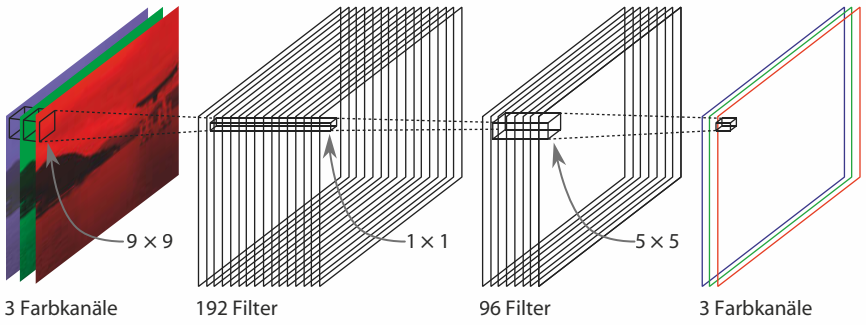
```
x = (a + b) / c
```

sorgt wie üblich dafür, dass in `x` die Summe aus `a` und `b` geteilt durch `c` steht. Normaler Python-Code müsste aber vorher die Variablen `a`, `b` und `c` mit Zahlen befüllen, damit der Interpreter `x` berechnen kann, sobald er die Zeile verarbeitet. Bei TensorFlow 1.x definiert man dagegen die Variablen als `Variable`- oder `placeholder`-Objekte, die zunächst keine Zahlen enthalten. `placeholder` dienen dazu, Eingabedaten aufzunehmen, während `Variable`-Objekte mit Werten initialisiert werden, die von Optimierungsalgorithmen angepasst werden können. TensorFlow verfolgt, was mit seinen Variablen passiert und baut intern einen Graphen auf, der die Formel repräsentiert. Die Knoten des Graphen sind Tensoren, also multidimensionale Arrays, die Kanten die Rechenoperationen. Mit diesem Graphen optimiert das Framework die Formel so gut es geht, ohne konkrete Werte einzusetzen. Optimierungsalgorithmen wie der stochastische Gradientenabstieg erweitern den Graphen vollautomatisch: Sie müssen für eine effiziente Optimierung der Fitness-Funktion deren Ableitung kennen. TensorFlow berechnet die Formel automatisch.

Um mit konkreten Zahlen zu rechnen, erzeugt man eine TensorFlow-`Session` und ruft deren Methode `run()` auf. `run()` übergibt eine Liste mit den symbolischen Variablen, die man berechnen möchte, beispielsweise `x` aus der eben genannten

Netzwerkstruktur

Das neuronale Netz nimmt die drei Farbkanäle des kleinen Bildes als Eingaben. Mit Filtern der Größe 9 × 9 extrahiert es per Faltung 192 Features. Aus diesen Features gewinnt es in der nächsten Schicht mit 1 × 1 Filtern 92 Kanäle mit abstrakteren Erkenntnissen, die es mit 5 × 5 Filtern im letzten Schritt wieder zu drei Farbkanälen kombiniert.



Formel, und ein Dictionary mit den Variablennamen und den Werten, die TensorFlow in die Formel einsetzen soll. Der komplette Code, um `x` für `a=2`, `b=3` und `c=4` zu berechnen, sieht dann so aus:

```
a = placeholder(float32)
b = placeholder(float32)
c = placeholder(float32)
x = (a + b) / c
sess = Session()
result = sess.run([x], {
  a: 2, b: 3, c: 4
})
print(result)  # [1.25]
```

Netze bauen

Der erste Schritt beim Bauen eines neuronalen Netzes ist das Vorbereiten der Eingabedaten. Sie sollten aus Gleitkommazahlen zwischen 0 und 1 bestehen. Beim klassischen Machine-Learning würde man einen Vektor aus möglichst bedeutungstragenden Eigenschaften berechnen. Für Gesichtserkennung könnten das beispielsweise der Augenabstand, die Nasenlänge, Mundbreite et cetera sein. Das Extrahieren solcher Daten erfordert aber viel handgeschriebenen Code. Hier liegt die Stärke von tiefen neuronalen Netzen: Sie nehmen normierte Eingabedaten direkt an und lernen auf ihren unteren Schichten zunächst, welche Eigenschaften für die gewünschte Ausgabe relevant sind. Auf den oberen Schichten stellen sie dann wie beim klassischen Machine-Learning Schlussfolgerungen an.

Fürs Skalieren von Bildern soll das neuronale Netz daher aus mehreren Schichten an Neuronen bestehen, die Formen erkennen. Die Eingabedaten sind dabei ein vierdimensionaler Tensor: Batch-Größe × Bildbreite × Bildhöhe × drei Farbkanäle. Ein Batch besteht in unserem Beispiel immer aus fünf Bildern. Um den Tensor zu normieren, reicht es, die Integer-Farbwerte der Pixel durch 256 zu teilen und die resultierenden Gleitkommazahlen zu verwenden. Die Rechenoperationen in den Hidden-Layern des Netzwerks berücksichtigen zwar benachbarte Pixel, ordnen aber trotzdem jedem Neuron der Eingabeschicht genau ein Neuron der Ausgabeschicht zu. Das Netzwerk kann die Auflösung also nicht erhöhen. Es kann aber aus einem unscharfen Bild ein scharfes berechnen. Als Vorbereitung skaliert der Code das Eingabebild daher zunächst bikubisch:

```
resized = resize_bicubic(
          inputs / 256.0, [480, 640])
```

Auf der höchsten Ebene sollen die Neuronen so „feuern“, dass ein scharfes Bild in der skalierten Auflösung entsteht. Da dort Zahlen zwischen 0 und 1 entstehen, muss dieser Tensor wieder mit 256 multipliziert und auf ganze Zahlen gerundet werden, damit ein Bild entsteht.

Faltungsnetz

Die einfachste Schicht eines neuronalen Netzes besteht aus Neuronen, die eine Synapse zu jedem Neuron auf der darunterliegenden Schicht haben (fully connected layer). Bei einem Bild würde das bedeuten, dass das Netz für jede Position im Bild jede Form und jedes Muster lernen muss. Eine Kante links oben wäre für das Netz etwas völlig anderes als eine Kante rechts unten. Außerdem entstünden durch die vielen Synapsen zu viele Parameter, die der Optimierungsalgorithmus während des Trainings anpassen müsste.

Aus diesem Grund haben sich bei der Bilderkennung Convolutional-Layer durchgesetzt. Sie erzwingen, dass eine Kante links oben genau wie eine Kante links unten behandelt wird, da sie für beide Positionen die gleichen Gewichte an den Synapsen verwenden. Stellen Sie sich dafür ein ganz kleines neuronales Netz vor, das beispielsweise nur ein einzelnes Neuron mit 27 Synapsen hat, die einen Bereich von 3 × 3 Pixeln „sehen“. Den Bereich, den dieses Neuron sieht, schieben Sie in Ihrer Vorstellung zeilen- und pixelweise über das Eingabebild. Aus dem Level der Aktivierung dieses Neurons an jeder Position entsteht wieder eine Art Schwarz-Weiß-Bild. Mathematisch entspricht das einer Faltung (englisch „convolution“) mit einem Filter, der den Gewichten an den Synapsen entspricht:

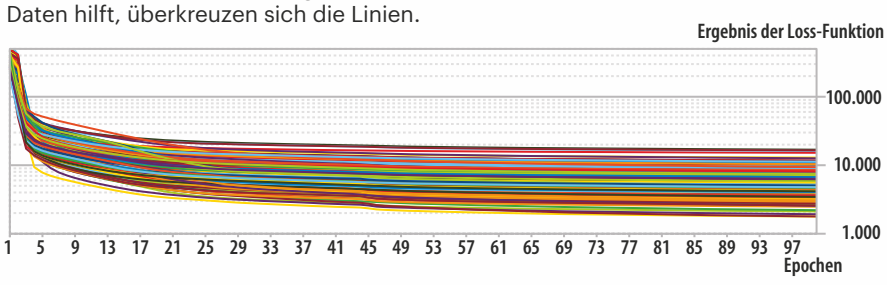
```
tf.nn.conv2d(eingabedaten, gewichte,
             strides=[1, 1, 1, 1], padding='VALID')
```

Der Parameter `strides=[1, 1, 1, 1]` legt fest, dass die Operation den Filter auf jedes „Pixel“ der `eingabedaten` faltet. Stünden dort höhere Werte, würde sie bei der ersten Dimension ganze Bilder im Batch, bei den folgenden beiden „Pixel“ eines Bildes (X- und Y-Richtung) und bei der Vierten „Farbkanäle“ überspringen. Mit höheren Werten ließen sich also „Bilder“ mit niedrigerer Auflösung erzeugen. `padding` legt fest, wie die Operation an den Rändern verfährt. Die Einstellung `"VALID"` sorgt dafür, dass die Faltung erst dort anfängt, wo alle Werte zur Verfügung stehen.

Damit deswegen keine kleineren Bilder entstehen, sollte man die Eingabedaten vorher mit `tf.pad()` um die Hälfte der Filtergröße minus 1 vergrößern. Bei der Einstellung `"SYMMETRIC"` spiegelt TensorFlow die Pixel am Rand, um nicht mit Nullen auffüllen zu müssen. Für eine Faltung mit 5 × 5-

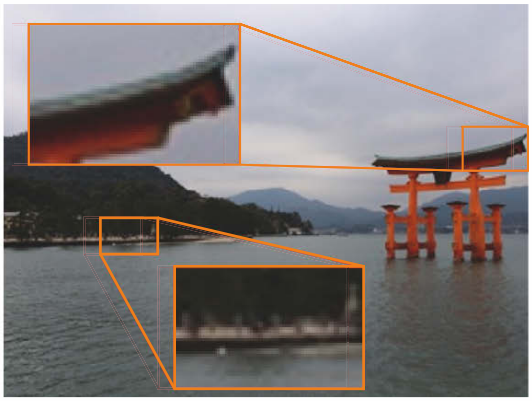
Fitness-Funktion

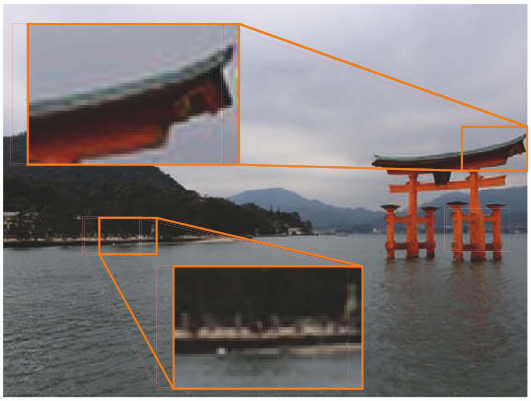
Konvergiert das Netz, sinkt der Wert der Fitness-Funktion im Verlauf des Trainings bei allen Batches der Trainingsdaten. Lernt das Netz etwas, das nur bei manchen Daten hilft, überkreuzen sich die Linien.



Das Ergebnis

Aus dem Eingabebild (unten) errechnet das neuronale Netz ein deutlich schärferes Bild (rechts) als der bikubische Filter (Mitte).





Filtern fehlen also oben und unten zwei Zeilen und rechts und links zwei Spalten. Das Padding sieht dafür so aus:

```
padded = tf.pad(data, [[0, 0], [2, 2],
        [2, 2], [0, 0]], "SYMMETRIC")
```

Die Batchgröße und die Kanäle erhalten kein Padding.

Netzwerktopologie

Handgeschriebene Algorithmen zum scharfen Skalieren von Bildern arbeiten meist in zwei Schritten: Zuerst extrahieren sie Bereiche mit definierbaren Eigenschaften, beispielsweise eine schräge Kante mit großem Hell-Dunkel-Kontrast. Danach rekonstruieren sie die fehlenden Pixel anhand der erkannten Bereiche im Eingabebild.

Unser neuronales Netz imitiert diese Struktur mit drei Convolutional Layern. Die erste Schicht besteht aus 192 Kerneln, die Bereiche von 9 × 9 Pixeln des Eingabebilds filtern. Nach der Faltung wendet das Netz die Funktion f(x)=max(0, x) auf die multiplizierten Werte an. Diese nichtlineare Funktion wird als Rectified Linear Unit (ReLU) bezeichnet. Erst durch eine nichtlineare Aktivierungsfunktionen können Neuronen logische Zusammenhänge berechnen und damit Schlüsse aus den Daten ziehen.

Die zweite Schicht dient dazu, aus den in der ersten Schicht extrahierten Mustern weitergehende Schlüsse zu ziehen. Dafür muss sie keine Werte mitteln, sodass die Filter hier nur die Größe 1 × 1 haben. Da die erste Schicht 192 Kanäle erzeugt hat, haben die Neuronen dennoch 192 Synapsen. Wie bei allen Schichten des Netzwerks kommt die ReLU als nichtlineare Funktion zum Einsatz. Die zweite Schicht erzeugt 96 Kanäle.

Die dritte Schicht dient dazu, aus den Schlussfolgerungen der zweiten Schicht wieder ein Bild mit drei Kanälen zu ermitteln. Sie hat dafür Filter der Größe 5 × 5, produziert aber nur die drei Farbkanäle als Ausgabe.

Hyperparameter optimieren

Der Optimierungsalgorithmus passt zwar die Gewichte innerhalb des Netzwerks an, bringt aber auch ein paar Stellschrauben mit, die er nicht selbst optimiert (Hyperparameter). Der wichtigste Wert ist die Lernrate. Ist sie zu hoch, springt der Optimierungsalgorithmus im Suchraum umher, ohne sich zu verbessern. Ist sie zu klein, braucht der Algorithmus zu viele Schritte, um zu den Trainingsdaten passende Parameter zu finden. Eine Daumenregel bei der Suche nach einer passenden Lernrate ist: Je tiefer das Netzwerk, desto kleiner muss die Lernrate sein, um überhaupt eine Lösung zu finden. Meist ist es sinnvoll, die Lernrate im Verlauf des Trainings zu verringern. Die großen Schritte am Anfang beschleunigen das Training,

während am Ende kleine Schritte nötig werden, um das Ziel auch zu treffen. Um beispielsweise die Lernrate alle 10 Epochen um 5 Prozent zu verringern, schreibt man:

```
lr = exponential_decay(
              0.0001, epoch, 10, 0.95,
                       staircase=True)
```

Eine weitere Stellschraube sind die Werte, mit denen TensorFlow die Gewichte initialisiert. Sie sollten nicht zu weit von den Werten entfernt sein, die die Gewichte am Ende des Trainings annehmen, da es lokale Minima geben könnte, in die der Algorithmus hineinläuft. Sie sollten aber auch nicht 0 sein, da es keinen Gradienten gibt, wenn eine Aktivierung nicht zum Fehler beiträgt. Üblicherweise initialisiert man mit einem normalverteilten Rauschen und stellt die Standardabweichung ein (hier 0,1):

```
weight_variable = tf.truncated_normal(
                    shape, stddev=0.1)
```

Experimentierfeld

Ob ein neuronales Netz beim Training konvergiert und wie schnell das geht, hängt von den Daten, der Topologie und den Hyperparametern ab. Datenwissenschaftler haben eine Intuition für sinnvolle Werte, aber selbst die müssen zahlreiche Experimente starten, um gute Werte zu finden. TensorFlow bietet eine Basis für eigene Versuche, da es die Hardware voll ausnutzt und schnelle Modifikationen an der Netzwerkstruktur erlaubt. Um beim Experimentieren die Übersicht zu behalten, bietet es sich an Sacred zu nutzen (siehe S. 70).

Sie finden den Code unserer eigenen Experimente zum Bilderskalieren auf GitHub (ct.de/wyrp). Mit scale.py erzeugen Sie zunächst Bilder der passenden Größe im Ordner scaled_images. Das Skript erwartet im Unterordner images eine Auswahl an Bildern, die Sie zum Training heranziehen möchten. Dort sollten mehrere hundert Bilder liegen (min. 640 × 480). Platzieren Sie auch ein paar Bilder im Ordner images/validation, damit das Netzwerk prüfen kann, ob es auch auf Daten, die nicht im Trainingsdatensatz vorkommen, sinnvolle Ergebnisse liefert.

Das Training starten Sie mit train.py. Das Skript zeigt die Rückgabewerte der Fitness-Funktion (Loss) nach jedem Trainingsschritt an. Sie sollten tendenziell sinken, variieren aber stark, da manche Batches Bilder enthalten, die sich leichter skalieren lassen als andere. Immer wenn der Algorithmus alle Eingabebilder gesehen hat, ist eine Epoche abgeschlossen und train.py prüft den aktuellen Trainingsstand an den unbekannten Bildern aus dem validation-Ordner. Sollte der Loss-Wert bei der Validierung nicht mehr sinken, haben Sie entweder ein fertig trainiertes Netz oder eines, das gar nicht lernt.

Die Netzwerkstruktur legt network.py fest. Ihre Definition steht im Konstruktor der Klasse `Network`. Da alle Convolutional Layer der gleichen Struktur folgen, kümmert sich die Methode `conv_layer()` um das Festlegen ihrer inneren Struktur. Die Lernrate stellen Sie in der Funktion `my-config()` ein. Wer besonders experimentierfreudig ist, kann auch die Fitness-Funktion `loss()` anpassen oder einen anderen Optimierungsalgorithmus ausprobieren.

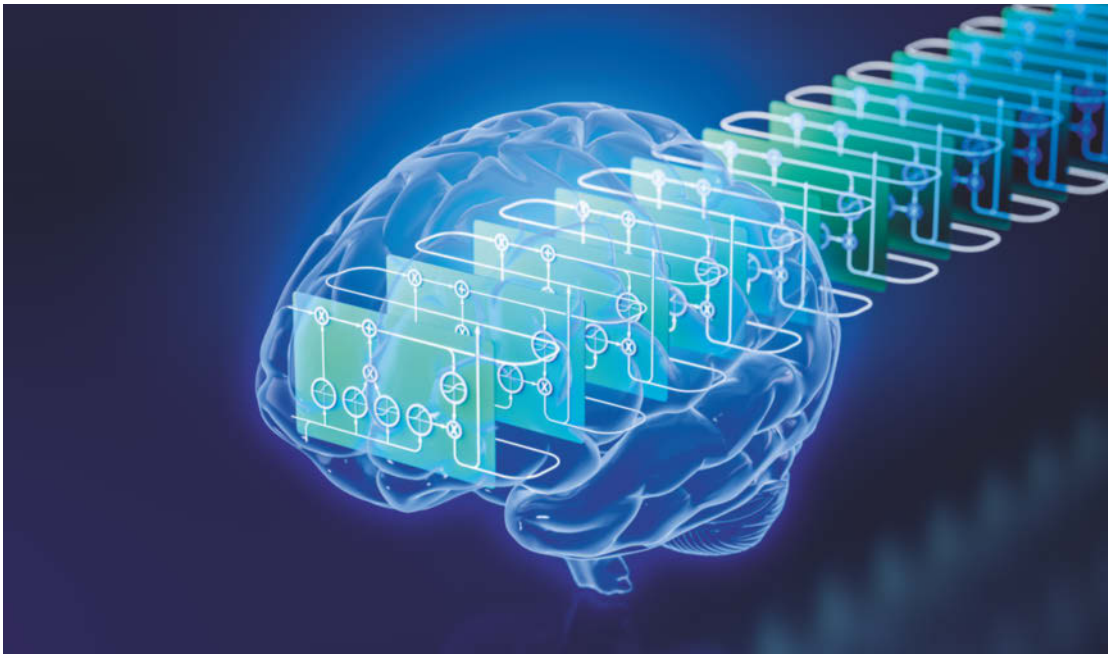
Um die Form des Netzes zu verändern, werden Sie einen Blick in die umfangreiche Dokumentation von TensorFlow werfen müssen. Dort findet sich auch ein Einsteigerbeispiel zum Klassifizieren von MNIST-Ziffern, das weitere Details zu Convolutional Networks erklärt.

Zum Anwenden des fertig trainierten Netzwerks dient inference.py. Das Skript skaliert das angegebene Bild und zeigt es an. Verwenden Sie dieses Skript als Grundlage, um Ihr neuronales Netz in bestehende Software, beispielsweise eine Django-Webanwendung, einzubauen.

Mit dem Repository checken Sie auch die Parameter eines trainierten Netzwerks aus. TensorFlow speichert neben network_params.data* auch noch eine Datei mit der Endung .index und eine mit .meta sowie eine checkpoint-Datei, die zusammen die Parameter samt Zusatzinformationen speichern. Falls Sie bei Epoche 0 anfangen möchten, löschen Sie diese vier Dateien und trainieren mit Ihren eigenen Bildern neu.

Wir nutzten fürs Training ein GeForce GTX 1080Ti mit CUDA 8.0 und CUDNN unter Ubuntu. Für die Parameter im Repository rechneten wir circa zwei Tage. Für erste eigene Experimente reicht aber auch schwächere Hardware, da man schon innerhalb der ersten Epochen erkennen kann, ob das Netz beim Training konvergiert. Die eigenen Experimente zahlen sich auf jeden Fall aus: Neuronale Netze gehören zu den wichtigsten technischen Innovationen der letzten Jahre und es lohnt sich, mehr darüber zu wissen, als Sie beim Lesen von Google und Amazons Marketingtexten erfahren. Einige Aspekte werden Sie erst verstehen, wenn Sie sie ausprobieren. (pmk) c't

Repository und Literatur:
www.ct.de/wyrp



Mit LSTMs Texte verschlagworten

Normale neuronale Netze kennen keine Zeit und können daher keine Sequenzen lernen. Rekurrente neuronale Netze dagegen versorgen sich selbst mit Hinweisen zum nächsten Schritt. Wirklich gut funktioniert das aber erst mit Long Short-Term Memory. Wir verschlagworten heise online und zeigen, wie Sie selbst ein LSTM trainieren.

Von Sebastian Stabinger

Auf heise online gibt es aktuell rund 50.000 Meldungen, die mit mindestens einem der 100 häufigsten Tags verschlagwortet wurden. Ein guter Leser schafft etwa 400 Wörter pro Minute und müsste zumindest 1000 Wörter jeder Meldung lesen, um sinnvoll aus den 100 häufigsten Schlagwörtern wählen zu können. Er bräuchte also mehr als 2000 Stunden, um die Meldungen zu lesen, was ihn über 254 Tage beschäftigen würde (er liest 8 Stunden pro Tag). Das will keiner machen.

Anhand der Häufigkeit bestimmter Wörter oder der Abfolgen von Wörtern ließe sich in vielen Fällen auch automatisch entscheiden, ob ein Schlagwort zum Artikel passt. Bevor man aber einen Entwickler gefunden hätte, der diese Zusammenhänge per

Hand – beispielsweise als reguläre Ausdrücke – programmiert, hat man auch sämtliche Texte gelesen.

Liegen genügend Beispiele vor, kann der Computer auch per Machine-Learning aus den Beispielen lernen, welche Schlagwörter zu einem Text passen. Fürs klassische Machine-Learning müsste der Regex-Programmierer aber dennoch Code zum Erzeugen eines Feature-Vektors schreiben – die Arbeit will er sich sparen. Also fällt die Wahl auf Deep Learning, das direkt mit den Texten als Eingabedaten arbeitet.

Im Trend liegen da neuronale Netze, da sie gut skalieren und sich auf Grafikkarten schnell berechnen lassen. Texte haben aber eine für neuronale Netze unangenehme Eigenschaft: Sie sind unterschiedlich lang und die Bedeutung ergibt sich durch die Abfolge der Wörter, nicht ihre absolute Position. Ein einfaches Feed-Forward-Netz müsste einen Zusammenhang für das gleiche Wort mehrmals erlernen, nur weil das Wort mal an zweiter und mal an fünfter Stelle im Text steht. Im Prinzip könnte es das. Es bräuchte dafür aber riesige Mengen an Trainingsdaten und hätte so viele Parameter, dass es nicht in den Speicher von Grafikkarten passt.

Rekurrente Netze

Rekurrente neuronale Netze (RNNs) versuchen dieses Problem zu lösen, indem sie die Eingabesequenz schrittweise verarbeiten. Ein rekurrentes Netz liest von den Texten im Beispiel immer ein Wort und erzeugt daraus eine Ausgabe, die es im folgenden Schritt als zusätzliche Eingabe neben dem nächsten Wort verwendet. Aus beidem erzeugt es dann die Ausgabe für das nächste Wort. Es berechnet in dieser Weise weitere Schritte, bis es das letzte Wort „gelesen“ hat. Die letzte Ausgabe dient als Gesamtergebnis, das der Optimierungsalgorithmus während des Trainings mit der gewünschten Ausgabe vergleicht.

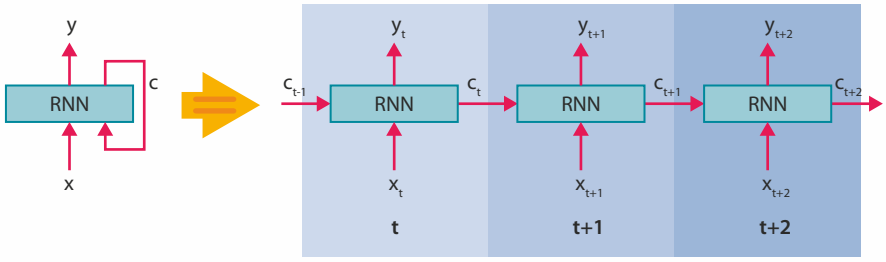
Rekurrente Netze sind dank ihres schrittweisen Vorgehens in der Lage, Sequenzen variabler Länge zu verarbeiten. Außerdem verwenden sie für jeden Schritt die gleichen Parameter, was deren Anzahl auf ein erträgliches Maß reduziert. Die für alle Schritte gleichen Parameter sorgen auch dafür, dass die Position keine Rolle mehr spielt und es stattdessen nur auf die Reihenfolge ankommt. Folgt beispielsweise „gut“ auf „nicht“, kann das Netzwerk lernen, sich selbst die nötigen Hinweise mitzugeben, um die Verneinung korrekt zu berücksichtigen. Die Hinweise für nachfolgende Schritte stellen das Gedächtnis des Netzwerks dar.

Die rekurrenten Netze eignen sich für Probleme mit einer zeitlichen Dimension wie Bewegungserkennung, Audio oder Handschrifterkennung. Daher wird die Auswertung einer Eingabe durch das Netzwerk üblicherweise als Zeitschritt bezeichnet, auch wenn es wie beim Textlesen nicht wirklich um einen zeitlichen Ablauf geht.

Da sich RNNs selbst beeinflussen, fällt es schwer sich vorzustellen, was wann passiert. Die

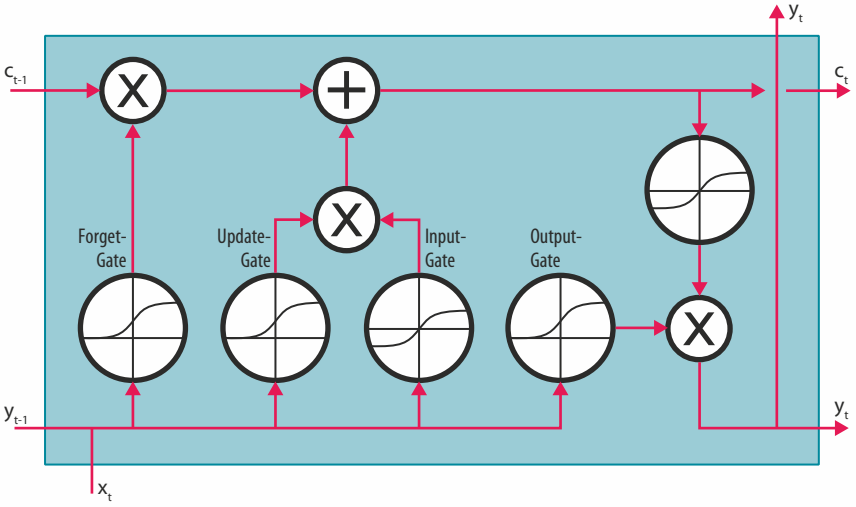
Rekurrente neuronale Netze

Das RNN erzeugt aus den Eingaben x_t und c_t die Ausgaben y_t und c_{t+1}. Rollt man die Darstellung für mehrere Zeitschritte ab, wird erkennbar, dass sich jeder Schritt wie ein Feed-Forward-Netz berechnet.



Struktur einer LSTM-Einheit

In LSTMs arbeiten mehrere Neuronen mit unterschiedlichen Aktivierungsfunktionen zusammen, um den inneren Zustand c anzupassen und danach eine Ausgabe y daraus zu erzeugen.



Selbstbezogenheit löst sich aber auf, wenn man die rekurrente Verbindung entlang der Zeitachse abrollt. Dann stehen mehrere neuronale Netze nebeneinander, die sich die gleichen Parameter teilen. Die gemerkten Daten fließen über die rekurrente Verbindung von einem Zeitschritt zum nächsten. Das so entrollte Netz trainiert man wie alle neuronalen Netze mittels Backpropagation, der Ableitung der durch das Netzwerk definierten Funktion, die Auskunft darüber gibt, welche Parameter wie stark zum Fehler beigetragen haben.

Long Short-Term Memory

Leider haben einfache RNNs Probleme damit, Informationen zu berücksichtigen, die mehr als ein paar Schritte in der Vergangenheit liegen. Das Problem liegt darin, dass eine Information aus der Vergangenheit in jedem Zeitschritt wieder durch Teile des neuronalen Netzes muss. Die Backpropagation erlaubt zwar bei jedem Schritt Rückschlüsse, welche Gewichte zum Fehler beigetragen haben. Wenn der Anteil aber nicht extrem groß ist, berechnet sie über mehrere Zeitschritte trotzdem nur ganz kleine Zahlen. Die nutzt der Optimierungsalgorithmus und passt die Gewichte an – stets viel zu wenig, wenn eine wichtige Information nicht kurz zuvor auftauchte. Mathematisch ausgedrückt wird der Gradient zu klein für den gewünschten Lernerfolg. Man bezeichnet dies als das Problem der verschwindenden Gradienten (vanishing gradient problem). Aufgrund dieser Limitierungen werden gewöhnliche RNNs in der Praxis wenig eingesetzt.

Zur Lösung dieses Problems haben Hochreiter und Schmidhuber 1997 sogenannte LSTM (Long Short-Term Memory) Netze entwickelt. Diese umgehen das Problem der verschwindenden Gradienten durch eine raffinierte innere Struktur. Um größere Gradienten zu bekommen, drehten Hochreiter und Schmidhuber den Spieß um und brachten den LSTMs bei, Informationen im Normalfall unverändert von Zeitschritt zu Zeitschritt weiterzugeben. Sie zu vergessen oder neue Informationen einzuspeisen wird zu einer Entscheidung, die Neuronen für den aktuellen Zeitschritt treffen müssen.

Um das zu erreichen, erhält das LSTM einen inneren Zustand c_{t-1} vom letzten Zeitschritt, den es

am eigentlichen neuronalen Netz vorbeischleust und an den nächsten Zeitschritt weitergibt (c_t). Die Information muss dadurch nicht durch feuernde Neuronen erhalten bleiben, was das Problem der verschwindenden Gradienten verhindert. Wie viel Information ein LSTM-Netz speichern kann, hängt direkt von der Anzahl der Werte in c_t ab.

Das System benötigt einen Mechanismus, um diesen internen Zustand verändern zu können. Hochreiter und Schmidhuber orientierten sich dafür an Flip-Flops, ersetzten die Logikgatter aber durch Neuronen, sogenannte Gates. Diese Neuronen nutzen die Sigmoid-Funktion, um Aktivierungen zwischen 0 und 1 zu erzeugen. Um Informationen zu vergessen, multipliziert das LSTM den internen Zustand mit der Aktivierung des Forget-Gate. Zu dem so bereinigten Zustand addiert das LSTM anschließend die Aktivierungen des zweiten Gatters. Das besteht aber eigentlich aus zwei verschiedenen Neuronen: Ein Neuron erzeugt mit dem Tangens-Hyperbolicus eine Aktivierung, die gespeichert werden könnte (Input-Gate). Ob das passiert, entscheidet ein zweites Neuron (Update-Gate) mit Sigmoid-Aktivierungsfunktion. Das LSTM multipliziert die Aktivierungen, bevor es sie zum internen Zustand addiert. Input- und Update-Gate arbeiten sehr eng zusammen und es gibt LSTM-Varianten, wo beide Gates durch ein gemeinsames Gate ersetzt werden (z. B. Gated Recurrent Units).

Seine eigentliche Ausgabe (y_t), die es auch an höhere Schichten weitergibt, erzeugt das LSTM über ein Neuron mit Tangens-Hyperbolicus als Aktivierungsfunktion aus dem internen Zustand nach dem Update (c_t). Diese Aktivierung geht aber auch nicht ungefiltert in den nächsten Zeitschritt und die nächste Aktivierung ein, da vorher noch das Output Gate entscheidet, welchen Teil der Aktivierung das LSTM weiterreicht.

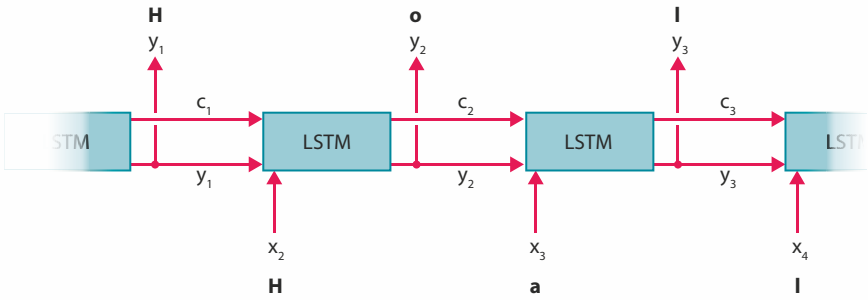
Heute sind LSTMs in unterschiedlichen Ausprägungen die Standardarchitektur für RNNs und werden zur Lösung vieler praktischer Probleme eingesetzt. So basieren zum Beispiel Google Translate und Apple Quicktype auf LSTMs. Amazons Alexa, Apples Siri und Googles Allo nutzen alle LSTMs für die Spracherkennung. Die richtigen Tags zu heise-online-Artikeln zu finden, stellt eine typische Anwendung dar.

Training

Ein LSTM wird generell immer auf Sequenzen von Daten trainiert, indem das Netzwerk entlang der rekurrenten Verbindungen abgerollt wird. Der Optimierungsalgorithmus trainiert das LSTM im Anschluss genau wie ein gewöhnliches Feed-Forward-Netz, mit dem Unterschied, dass für alle Zeitschritte die gleichen Gewichte für das Netzwerk verwendet werden. Entscheidet der Algorithmus also, dass das

Training von LSTMs

Ein entlang den Zeitschritten abgerolltes LSTM. Nach dem Abrollen kann man das LSTM wie ein normales (allerdings sehr großes) neuronales Netz trainieren.



Gewicht g_1 in Zeitschritt t_1 verkleinert und in Zeitschritt t_2 vergrößert werden muss, wird der Wert von g_1 alles in allem unverändert bleiben.

Letztlich lernt das Netzwerk durch das Training auf ganzen Sequenzen, nützliche Informationen im aktuellen Zeitschritt in den internen Zustand aufzunehmen, da sie zu einem späteren Zeitpunkt zu einer besseren Ausgabe des Netzes führen. Speichern und Nutzen der Information müssen dafür innerhalb einer Sequenz stattfinden, damit sich die Anpassung der dafür gebrauchten Gewichte in der Summe lohnt. Das Netz entscheidet daher eigenständig anhand der Trainingsdaten, welche Informationen in welcher Situation wichtig sind und wie diese im internen Zustand repräsentiert werden sollen. Man sollte bei dieser Interpretation allerdings nie vergessen, dass letztlich auch bei einem LSTM nichts anderes gemacht wird, als die Parameter einer riesigen Formel so zu verändern, dass die Resultate dieser Formel besser zu gewünschten Resultaten passen als vor dem Training. Alle „cleveren" Tricks und Kniffe, die ein LSTM nach dem Training verwendet, um das gewünschte Problem zu lösen, entstehen quasi von selbst aus dem gegebenen Datensatz.

Das führt zur vermutlich wichtigsten Erkenntnis über künstliche Intelligenz der letzten zehn Jahre: Mithilfe großer Datenmengen, viel Rechenleistung und simplen Algorithmen können intelligente Systeme scheinbar aus dem Nichts entstehen.

Ein LSTM selbst gebaut

Mit Frameworks wie TensorFlow haben Sie ein eigenes LSTM erstaunlich schnell in Python programmiert. Noch schneller geht es mit Keras, einem API, das TensorFlow implementiert. Keras abstrahiert die üblichen Schritte zum Aufsetzen neuronaler Netze, sodass Sie mit ein paar Zeilen Python-Code Ihr eigenes LSTM aufsetzen – einschließlich Nutzung der GPU.

Fürs Training haben wir die Newsmeldungen aus heise online bereits in Listen aus Wortnummern konvertiert. Die Wörter haben wir nach Häufigkeit sortiert, sodass Sie das Training leicht auf die 10.000 häufigsten Wörter eingrenzen können. Satzzeichen gelten als Wörter und die Zahl 0 steht für ein Nicht-Wort. Die 10.000 häufigsten Wörter stellen einen Kompromiss dar: Unter den 100 häufigsten Wörtern tummeln sich zwar sämtliche Artikel, Präpositionen und schwache Verben, die wenig darüber aussagen, welche Schlagwörter zum jeweiligen Text passen. Kommen im Artikel allerdings zu oft seltene Wörter vor, die in den Trainingsdaten nur im Zusammenhang mit einem einzigen Schlagwort stehen, lernt das Netz diesen Zusammenhang auswendig und versagt anschließend, wenn das Wort bei unbekannten Texten in einem anderen Zusammenhang vorkommt. Wenn sich ein neuronales Netz wie ein fauler Schüler verhält, nennt man das Overfitting, da es nur lernt, was zu den Trainingsdaten passt, ohne allgemeine Muster zu verstehen.

Um viele Texte parallel verarbeiten zu können, haben wir sie mit Nullen auf die gleiche Länge aufgefüllt und Gzip-Komprimiert in eine hdf5-Datei geschrieben, die Sie zusammen mit dem Quellcode im GitHub-Repository zum Artikel unter ct.de/wu73 finden.

Daten vorbereiten

Das Beispiel geht von TensorFlow als Backend in Keras aus und nutzt Numpy, um die Daten vorzubereiten. Außerdem kommt Sacred zum Einsatz, um die Konfiguration zu verwalten. Installieren Sie diese Frameworks mit `pip`:

```
pip install wheel
pip install numpy tensorflow-gpu
pip install sacred pymongo h5py
```

Um den Erfolg des Trainings zu prüfen, lohnt es sich, einen Teil der Daten zur Validierung abzuschneiden und sie nicht während des Trainings zu verwenden. Vorher sollte man die Daten durchmischen, damit das Netzwerk nicht nur mit News vor 2016 trainiert und auf neueren Artikeln validiert. Wir haben die Vorverarbeitung in ein Sacred-Ingredient gepackt, das Sie nutzen können, um gleich zum Wesentlichen zu kommen:

```
from sacred import Experiment
from heise_online_dataset import ↵
          ↳heise_online_ingredient, ↵
           ↳load_data, get_word_count
ex = Experiment('LSTM_Classification',
          ingredients=[heise_online_ingredient])
```

Sacred kümmert sich darum, die Konfigurationsvariablen zu verwalten und ein Kommandozeilen-Interface zur Verfügung zu stellen. Die Variablen stehen dafür in der mit `@ex.config` dekorierten Funktion `my_config()`. Was dort definiert wurde, übergibt Sacred automatisch an die mit `@ex.automain` dekorierte Funktion `train_network()`. Die Struktur er-

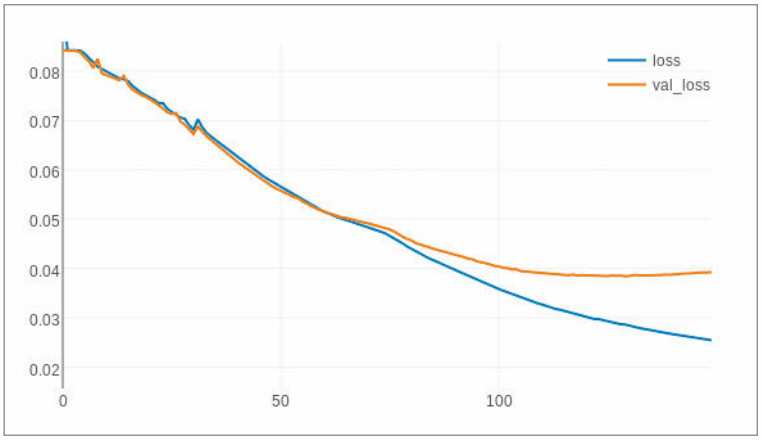


Diagramm der Loss-Funktion beim Training mit zu wenig Dropout: Während der ersten 70 Epochen schneidet das Netzwerk auf Trainings- und Testdaten gleich gut ab. Danach lernt es Beispiele aus den Trainingsdaten auswendig, was dazu führt, dass es auf unbekannten Daten ab Epoche 120 schlechter abschneidet als zuvor.

leichtert das systematische Ausprobieren der in `my_config()` definierten Hyperparameter. Wenn Sie das verwirrend finden, schreiben Sie stattdessen den Inhalt von `my_config()` an den Anfang von `train_network()`.

Laden Sie zuerst den vorbereiteten Datensatz:

```
X_train, y_train, X_test, y_test = ↵
                        ↳load_data()
```

In `X_train` und `X_test` stehen anschließend die Listen mit Wortnummern, während in `y_train` und `y_test` die Vektoren mit den korrekten Schlagworten dazu stehen. Diese Vektoren sehen wie Wahrscheinlichkeitsverteilungen aus: Passende Schlagwörter haben eine „Wahrscheinlichkeit“ von 1,0, unpassende eine von 0,0. Welche Schlagwörter zu diesen Zahlen gehören, verrät die Funktion `get_category_list()` aus heise_online_dataset.py.

Netzwerk definieren

Keras enthält die Klasse `Sequential()`, die neuronale Netze verwaltet, bei denen eine Schicht ihre Ausgaben immer als Eingaben an die nächste Schicht übergibt. Diese Struktur kommt bei den allermeisten neuronalen Netzen zum Einsatz.

```
from tensorflow.keras.models import ↵
  ↳Sequential
model = Sequential()
```

Schichten fügen Sie dem `model` mittels `add()` hinzu. Die erste Schicht nimmt eine Sonderrolle ein, da sie aus den Integer-Wortnummern in den Trainingsdaten Vektoren mit Gleitkommazahlen berechnet. Neuronale Netze können nämlich nur mit Gleitkomma-Tensoren rechnen. Die Konvertierung übernimmt eine `Embedding()`-Schicht, für die Sie mit dem Parameter `embedding_vector_dimensionality` einstellen können, wie breit der entstehende Vektor wird. Wir haben mit 128 Dimensionen gute Ergebnisse erzielt. Die Embedding-Schicht enthält Parameter, die zusammen mit den Parametern der Neuronen trainiert werden:

```
from keras.layers.embeddings import ↵
                              ↳Embedding
model.add(Embedding(get_word_count(),
      embedding_vector_dimensionality,
       input_length=X_train.shape[1]))
```

Nach ihr haben wir eine Dropout-Schicht eingefügt, die keine trainierbaren Parameter enthält. Sie setzt einen über `embedding_dropout_factor` festgelegten Anteil zufälliger Werte im Tensor auf 0.0, was es dem Netzwerk schwerer macht, einzelne Trainingsbeispiele auswendig zu lernen. Dropout ist neben dem Verzicht auf seltene Wörter eine zweite Möglichkeit, gegen Overfitting vorzugehen. Neuronale Netze stürzen sich sofort auf Korrelationen zwischen einzelnen Zahlen und einer gewünschten Klasse. Steht die Zahl manchmal nicht zur Verfügung, lohnt es sich für das Netzwerk, sich eher auf abstrakte Muster zu verlassen, was auf lange Sicht zu einem erfolgreicheren Training führt. Das Training dauert mit Dropout jedoch länger.

```
from tensorflow.keras.layers import Dropout
model.add(Dropout(
            embedding_dropout_factor))
```

Anschließend folgen eine oder mehrere Schichten mit LSTM-Einheiten:

```
from tensorflow.keras.layers import LSTM
model.add(LSTM(units=100,
               return_sequences=True,
               recurrent_dropout=0.1,
               dropout=0.1))
```

Der Parameter `units` bestimmt die Dimensionalität des internen Zustands und damit auch der Ausgabe dieser LSTM-Schicht. Der Wert 100 bedeutet, dass dem Netzwerk als Gedächtnis 100 Werte dienen und dass die Schicht für jede Eingabe auch 100 Werte als Ausgabe erzeugt. Je höher Sie die Dimensionalität wählen, desto mehr kann sich das Netzwerk merken. Dadurch steigt allerdings auch die Anzahl an Neuronen, die Sie trainieren müssen, rapide an. Wählen Sie den Wert größer als nötig, dauert das Training länger und bei zu großen Werten wird das Netzwerk größer als Ihr Arbeitsspeicher.

Keras unterstützt LSTM-Schichten in zwei Varianten. Im Standardfall (`return_sequences=False`) wird eine ganze Sequenz an Daten durch die LSTM-Schicht geschleust und nur die letzte Ausgabe des Netzes wird an die nächste Schicht weitergegeben. Das brauchen Sie für die letzte LSTM-Schicht in Ihrem Netzwerk. Alle vorherigen Schichten müssen mit `return_sequences=True` auch Zwischenwerte ausgeben, damit die darüberliegenden LSTM-Schichten damit arbeiten können.

Die beiden `dropout`-Parameter setzen ähnlich wie die `Dropout()`-Schicht den angegebenen Anteil an zufälligen Werten auf 0. Hier geht es aber um den Anteil der von Schritt zu Schritt weitergegebenen Informationen, die eine `Dropout()`-Schicht nicht beeinflusst.

Um das Netzwerk zu trainieren, vergleicht Keras die Ausgabe des Netzes mit der erwarteten Ausgabe. Die Ausgaben müssen daher die gleiche Größe haben und dürfen nur Werte zwischen 0,0 und 1,0 enthalten. Am leichtesten erreichen Sie das mit einer voll verbundenen Schicht aus 100 Neuronen mit Sigmoid-Aktivierungsfunktion:

```
from tensorflow.keras.layers import Dense
model.add(Dense(y_train.shape[1],
                activation='sigmoid'))
```

Damit enthält `model` das neuronale Netz. Um seine Parameter zu trainieren, wählen Sie einen Optimierungsalgorithmus für den Gradientenabstieg:

```
from tensorflow.keras.optimizers import ↵
  ↳Adam
optimizer = Adam(lr=0.001, decay=0.00)
```

TensorFlow setzt aus dem Netzwerk, seiner Ableitung und dem Optimierungsalgorithmus einen Graphen auf und kompiliert ihn. Bei diesem Schritt geben Sie an, welche Loss-Funktion `Adam` minimieren soll und welche Metriken das Framework zusätzlich berechnet, um den Erfolg des Trainings abzuschätzen:

```
model.compile(
 loss='binary_crossentropy',
 optimizer=optimizer,
 metrics=['binary_accuracy', c_score])
```

Die `binary_accuracy` steigt schnell auf fast 98 Prozent, da das Netzwerk zunächst lernt, für alle Tags zu raten, dass sie nicht passen. Der interessante Teil passiert aber bei den letzten zwei Prozentpunkten, was man an der Zahl nur schwer ablesen kann. Wir haben uns daher den „Critics Score“ (`c_score`) ausgedacht, bei dem das Netz für jedes richtig geratene Schlagwort einen Punkt gewinnt und für jedes falsch geratene einen abgezogen bekommt. Damit der Wert bei mehreren richtigen Schlagwörtern nicht über 1 steigt, teilt der `c_score` die Punktzahl durch die Anzahl der korrekten Schlagwörter. Das Netzwerk startet zu Beginn des Trainings mit stark negativen Werten, erreicht dann recht schnell 0, wenn es alle Schlagwörter ablehnt, und arbeitet sich im Erfolgsfall anschließend langsam bis auf Werte zwischen 0,3 und 0,4 vor.

Das Training starten Sie mit `model.fit()`. Der Funktion übergeben Sie die Datensätze für Training und Validierung, die Anzahl an Epochen, die TensorFlow trainieren soll, und eine `batch_size`, die festlegt, wie viele Datensätze TensorFlow parallel bearbeitet:

```
model.fit(X_train, y_train,
     validation_data=(X_test, y_test),
     epochs=epoch_no,
     batch_size=batch_size)
```

Je größer die Batches, desto besser lastet das Training die Grafikkarte aus. Wenn Sie den Parameter zu groß wählen, passt das Netzwerk jedoch nicht mehr ins RAM und das Training bricht ab. Auf einer Tesla P100 konnten wir sehr gut mit einer Batchgröße von

512 trainieren, mussten den Wert für eine ältere GTX 970 aber auf 256 senken. Ob Ihre Karte gut ausgelastet ist, sehen Sie mit dem Tool `nvidia-smi`.

Auch auf einer schnellen Grafikkarte dauert das Training einige Stunden. Mit nur einer Schicht aus 100 LSTM-Einheiten brauchten wir bereits über sechs Stunden auf der P100. Wenn Sie weitere Schichten einfügen, mit mehr als 500 Wörtern aus jedem Datensatz oder für mehr als 150 Epochen trainieren, brauchen Sie noch länger. Unser Code schreibt die aktuell besten Gewichte nach jeder Epoche in die Datei weights.hdf5 und sichert diese auch in der Sacred-Datenbank. Bei jedem neuen Experiment wird weights.hdf5 überschrieben.

Inferenz

Nach dem Training könnten Sie mit dem Netzwerk eigene Texte verschlagworten. Füttern Sie dafür die Funktion `model.predict()` mit Daten. Als Ausgabe erhalten Sie dann die Wahrscheinlichkeitsverteilung zu den 100 Schlagwörtern. Den Text müssen Sie dafür vorher in eine Liste aus Ganzzahlen verwandeln. Die Liste der Wörter bekommen Sie über die Funktion `get_word_list()` aus heise_online_dataset.py. Kommt ein Wort darin nicht vor, setzen Sie es auf 0.

Falls Sie wie wir lieber nach dem perfekten Netzwerk für die Aufgabe suchen, reicht es, die Einstellungen in `my_config()` in train_lstm.py anzupassen. Wir haben auch Experimente mit einem voll verbundenen neuronalen Netz gestartet, das zu jedem Wort eine Wahrscheinlichkeitsverteilung berechnet, die wir für den gesamten Text gemittelt haben (train_fc.py). Dieses Netz kam über einen `c_score` von 0 nie hinaus. Mit einem einfachen rekurrenten Netz (train_simpleRNN.py) kamen wir immerhin auf einen leicht positiven Score von 3 Prozent. Mit den richtigen Parametern schneidet das LSTM aber mit 39 Prozent am besten ab. Möglicherweise finden Sie noch bessere Parameter als unsere Vorgaben. Wenn Sie einen `c_score` von über 0,39 bei der Validierung erreichen, schreiben Sie uns, was Sie eingestellt haben. Wir passen dann die Voreinstellungen im Repository an und erwähnen, wer die Parameter gefunden hat. (pmk) c't

Literatur

[1] Andrea Trinkwalder, **Netzgespinste**, Die Mathematik neuronaler Netze: einfache Mechanismen, komplexe Konstruktion, c't 06/16, S. 130

[2] Pina Merkert, **Ziffernlerner**, Ein künstliches neuronales Netz selbst gebaut, c't 06/16, S. 142

Downloads und Repository:
www.ct.de/wu73

Den Stil eines Malers auf Fotos anwenden

Wie hätte wohl Claude Monet die Fotomotive des letzten Urlaubs gemalt? Ein KI-Algorithmus aus der Forschung erzeugt Bilder, die einer Antwort darauf erstaunlich nahekommen. In diesem Artikel erklären wir, wie mehrere neuronale Netze lernen, ein Bild so zu manipulieren, dass aus Ihren Urlaubsfotos Gemälde im Stil berühmter Künstler entstehen.

Von Bastian Wandt

Mit dem sogenannten „Stiltransfer“ beschäftigen sich immer mehr KI-Forscher. Der macht es möglich, tote Maler wie Claude Monet virtuell wiederauferstehen zu lassen. Der wiederbelebte Maler ist eigentlich ein Algorithmus, der mit einem Foto als Vorlage ein Bild berechnet, das den Stil eines Künstlers nachahmt. Beispielsweise erzeugt der KI-Maler aus einem im Yosemite-Nationalpark geschossenen Urlaubsfoto ein Bild, welches das gleiche Motiv zeigt, jedoch im Stil von Monets Abendstimmung in Venedig.

Diese anspruchsvolle Aufgabe meistert eine Technik namens „Convolutional Neural Networks“ (CNNs) (siehe Seiten 84, 104). Diese neuronalen Netze brauchen normalerweise zu jedem Eingabedatum die perfekt richtige Ausgabe. Produzieren sie während des Trainings eine andere Ausgabe, haut der Trainingsalgorithmus ihnen auf die Finger.

Der Maler Monet hat das Yosemite-Tal jedoch nie besucht, sodass es zum Urlaubsfoto keine Vorlage gibt, an der das neuronale Netz lernen könnte. Außerdem brauchen die Netze sehr viele Bei-

spiele zum Lernen und selbst Monets viele Bilder von der japanischen Brücke in seinem Garten würden als Trainingsdaten für ein einzelnes CNN nicht ausreichen. Mit der CNN-Variante der CycleGANs lernen CNNs auch ohne direkt einander entsprechende Trainingsdaten den Stil eines Malers zu imitieren. Einige hundert Fotos und Gemälde muss man als Trainingsdaten aber dennoch sammeln – sie dürfen aber unterschiedliche Motive zeigen.

CycleGANs verwenden fürs Training einen Trick: Ähnlich wie DeepFakes [1] setzen sie auf sogenannte „Generative Adversarial Networks“ (kurz: GAN). Ein GAN besteht aus zwei neuronalen Netzen, einem Generator und einem Diskriminator. Der Generator ist ein professioneller Kunstfälscher, während der Diskriminator ein Kunstkritiker ist, der Fälschungen von Originalen unterscheidet. Die beiden Netze arbeiten gegeneinander: Zuerst erzeugt der Generator aus einem Foto ein Gemälde. Anschließend bewertet der Diskriminator mit dem Blick des geschulten Kritikers das Werk. Fällt ihm die Unterscheidung leicht, hat der Fälscher schlechte Arbeit abgeliefert. Hat er dagegen Probleme, Fälschungen von Originalen zu unterscheiden, zeugt das von meisterhaften Fälschungen. Da der Trainingsalgorithmus stets weiß, ob er dem Diskriminator eine Fälschung oder ein Original zur Begutachtung vorlegt, fällt es ihm leicht, dem Diskriminator bei Fehlern im Training auf die Finger zu hauen.

Den Generator zu trainieren ist jedoch schwieriger. Er soll lernen, den Diskriminator immer besser zu täuschen; zu Beginn sind seine Werke jedoch wenig überzeugend. Der Trainingsalgorithmus weiß aber: Der Fälscher hat gerade dann gute Arbeit geleistet, wenn der Diskriminator keine Ahnung hat, ob ihm Original oder Fälschung vorliegt. Der Algorithmus haut dem Diskriminator also so lange auf die Finger, bis er Bilder erzeugt, die den Kritiker täuschen. Das gelingt ihm nur, wenn seine Werke den Vorlagen des Künstlers immer mehr ähneln. Die beiden Netze trainieren sich so gegenseitig und lernen, ihre jeweiligen Aufgaben immer besser zu erfüllen. Am Ende des Trainings erzeugt der Generator Bilder, die sowohl der Diskriminator als auch ein Mensch kaum mehr von echten Gemälden unterscheiden kann.

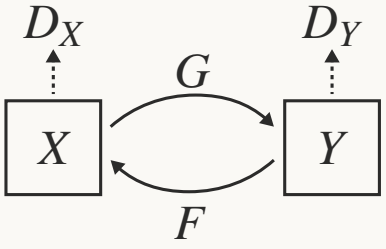
Gepaartes Doppel

Durch das Spiel aus Täuschen und Entlarven wird ein einzelnes GAN gut darin, den Stil zu treffen. Der Generator hat aber wenig Anreiz, den Inhalt des Fotos im Gemälde zu erhalten. Das könnte man wie eingangs erwähnt mit Trainingsdaten erzwingen, die zu einem Foto immer ein Gemälde mit dem gleichen Motiv aus der gleichen Perspektive enthalten, gemalt mit der gleichen Lichtstimmung. Im Normalfall gibt es zu einem Gemälde aber kein passendes Foto und zu einem Foto kein passendes Gemälde. Deswegen besteht ein CycleGAN aus zwei solcher GANs.

Ein zweiter Generator erzeugt aus einem Gemälde ein Foto, welches wiederum ein zweiter Diskriminator bewertet. Der zweite Diskriminator trainiert dafür, echte Fotos von Fakes zu unterscheiden. Berechnet das Netz mit dem ersten Generator ein Gemälde und mit dem zweiten Generator wieder ein Foto, kann der Algorithmus einfach vergleichen, wie ähnlich das so erzeugte Bild dem ursprünglichen Bild ist. Im Idealfall entsteht nach dem zweiten Generator wieder das Ursprungsbild. Diesen „Kreislauf“ nennen die Forscher „cycle consistency“, woraus der Name CycleGAN entstanden ist. Durch die „cycle consistency“ müssen die erzeugten Bilder nicht nur stilistisch zum Maler, sondern auch inhaltlich zur Vorlage passen.

Cycle GAN

In einem CycleGAN erstellt der Generator G aus einem Bild X (z. B. ein Foto) ein Bild Y (z. B. ein Gemälde). Ein weiterer Generator F berechnet aus dem Y wieder ein Bild, das X möglichst ähnlich sieht. Die jeweiligen Diskriminatoren D_x und D_y bewerten dabei, ob die Bilder realistisch aussehen und dem Stil entsprechen.



Beim Training kann man den Kreis sowohl mit einer Fotografie als auch mit einem Gemälde starten. Somit ist es nicht erforderlich, dass zu jedem Gemälde ein zugehöriges Foto und umgekehrt zu jedem Foto ein Gemälde existiert. Mit einer großen Menge an Bildern (beim Monet-Datensatz 1337 Gemälde und 2944 Fotos) lernt dieses zyklische Netzwerk bestehend aus den vier zusammengeschalteten neuronalen Netzen nun, wie Monet zu malen.

Dass die beiden Diskriminatoren dabei richtig gute Detektive fürs Erkennen der Fakes werden, spielt für die Anwendung nach dem Training keine Rolle. Die Aufgabe der Diskriminatoren besteht nur darin, das Training der beiden Generatoren in die richtige Richtung zu lenken. Will man ein Urlaubsfoto in ein Monet-Gemälde umwandeln, reicht der erste Generator. Das Ergebnis sieht überzeugend realistisch aus. Die Berge wirken wie per Pinsel gemalt und auch die Büsche und das Wasser sind leicht verschwommen gestrichelt, wie bei einem Ölgemälde.

Eigene Gemälde erzeugen

So viel zur Theorie, nun folgt die Praxis: Sie möchten ein Urlaubsfoto in ein Gemälde verwandeln? Kein Problem! Richten Sie zunächst eine KI-Arbeitsumgebung ein. Wir haben mit Anaconda gute Erfahrungen gemacht. Das Projekt bietet unter `www.anaconda.com` Installer für alle Desktop-Betriebssysteme zum Download an. Die Installation ist je nach System menügeführt oder grafisch. Lassen Sie Anaconda eine Default-Umgebung einrichten. Die trägt alle nötigen Pfade in Umgebungsvariablen ein. Anaconda aktiviert die Umgebung auch gleich automatisch in allen neu gestarteten Konsolenfenstern. Wenn Sie das stört, schalten Sie es einfach nach der Installation mit folgendem Befehl ab:

```
conda config --set ↵
            ↳auto_activate_base false
```

Richten Sie anschließend eine neue Anaconda-Arbeitsumgebung mit dem Namen „cyclegan“ ein:

```
conda create -n cyclegan python=3.7
```

Wichtig ist, die aktuelle Python-Version 3.7 und nicht die veraltete 2.7 einzustellen. Falls Ihre Shell den Befehl `conda` nicht findet, geben Sie den ganzen Pfad an, beispielsweise:

```
~/anaconda3/bin/conda create ↵
               ↳-n cyclegan python=3.7
```

Folgender Befehl lädt unter Linux die Anaconda-Arbeitsumgebung (unter Windows aktivieren Sie die Umgebung über das Menü):

```
conda activate cyclegan
```

Laden Sie als nächsten Schritt den Code für die CycleGANs herunter. Mit `git` geht das sehr einfach:

```
git clone https://github.com/junyanz/↵
        ↳pytorch-CycleGAN-and-pix2pix
```

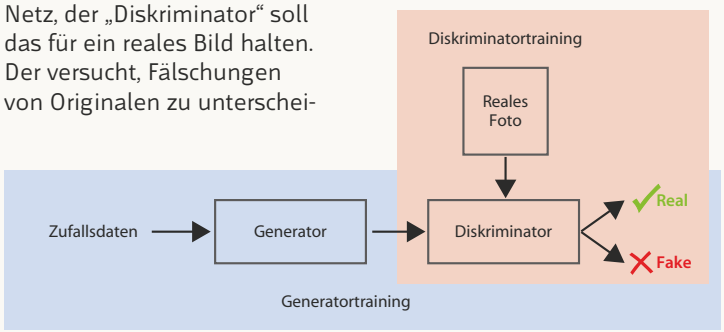
Sollten Sie kein Git installiert haben, laden Sie stattdessen einfach das Zip-Archiv von der Webseite des Repositories bei GitHub herunter und entpacken es (siehe ct.de/w3q3).

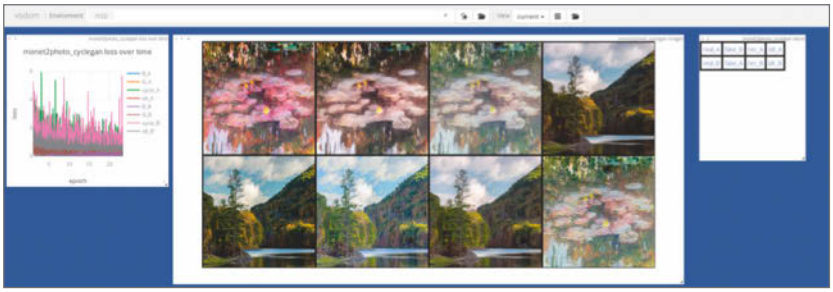
Danach befinden sich im Ordner `pytorch-CycleGAN-and-pix2pix` die Programmdateien. Damit die laufen, fehlen allerdings noch einige Pakete:

```
conda install numpy pyyaml setuptools
conda install mkl mkl-include cmake
conda install cffi typing pytorch
conda install torchvision -c pytorch
conda install visdom -c conda-forge
conda install dominate -c conda-forge
conda install jsonpatch -c conda-forge
```

GAN: Generative Adversarial Network

In einem GAN erzeugt ein neuronales Netz, der „Generator“, aus beliebigen Eingangsdaten (beispielsweise ein Eingabebild oder auch nur Zufallszahlen) ein Bild. Ein weiteres neuronales Netz, der „Diskriminator“ soll das für ein reales Bild halten. Der versucht, Fälschungen von Originalen zu unterscheiden. Man trainiert Generator und Diskriminator abwechselnd, sodass beide Netze lernen müssen, in ihrer Aufgabe immer besser zu werden.





Der Visdom-Server stellt eine lokale Webseite bereit, auf der Sie den Trainingsfortschritt verfolgen können. Das Diagramm links visualisiert die einzelnen Komponenten der Loss-Funktion. Die obere Reihe Beispielbilder zeigt das Monet-Original, das daraus imaginierte Foto und das aus dem imaginierten Foto rekonstruierte Original. Die Reihe darunter zeigt den Kreis für ein Foto: Original, Fake-Monet, Fake-Foto vom Fake-Monet.

Unter Linux führen Sie fürs Installieren dieser Pakete ganz bequem folgendes Skript aus dem Repository aus:

```
bash ./scripts/conda_deps.sh
```

Für eine Installation auf Systemen ohne CUDA-Grafikkarte nutzen Sie folgenden Befehl für die Installation von pytorch und torchvision:

```
conda install pytorch torchvision ↵
                                ↳cpuonly -c pytorch
```

Danach fehlt nur noch ein Datensatz. Zum Testen laden Sie am besten einen der fertigen Datensätze. Die haben sprechende Namen wie `maps` oder `horse2zebra`. Unser Beispiel verwendet `monet2photo`. Den Download erledigt ebenfalls ein Skript:

```
bash ./datasets/download_cyclegan_↵
                ↳dataset.sh monet2photo
```

Falls das Skript bei Ihnen nicht läuft, können Sie die Datensätze auch von `https://people.eecs.berkeley.edu/~taesung_park/CycleGAN/datasets/` herunterladen und in einen passend benannten Ordner unterhalb von `datasets/` entpacken. Mehr macht das Skript auch nicht.

Die Trainings- und Testbilder befinden sich danach im Ordner `./datasets/monet2photo` in den Unterordnern `trainA`, `trainB`, `testA` und `testB`.

Das Training des neuronalen Netzes starten Sie mit folgendem Befehl:

```
python train.py ↵
  ↳--dataroot ./datasets/monet2photo ↵
  ↳--name monet2photo_cyclegan ↵
  ↳--model cycle_gan
```

Für ein Training ohne CUDA ergänzen Sie die Option `--gpu_ids -1`. Das Repository enthält auch eine Webanwendung, mit der Sie den Trainingsfortschritt visuell überwachen können. Um die zu sehen, starten Sie zuerst den Visualisierungsserver mit

```
python -m visdom.server &
```

Unter Windows sollten Sie dafür im Anaconda-Navigator ein neues Konsolenfenster des `cyclegan`-Environment starten, da das `&` den Prozess dort nicht im Hintergrund öffnet.

Anschließend sehen Sie im Webbrowser unter der Adresse `http://localhost:8097` den Trainingsfortschritt als Diagramm und jeweils zwei Beispiele aus den Trainingsdaten A und B (die ersten Bilder erscheinen erst, nachdem das Netz eine Weile trainiert hat).

Wir haben das Netz auf einer Geforce GTX 1080 Ti von Nvidia trainiert. Nach ungefähr 24 Stunden hatte es ausreichend lang gelernt und erzeugte in unserem Test schöne Ergebnisse.

Inferenz

Nach dem Training können Sie das Netz mit eigenen Fotos testen. Legen Sie dafür Ihre Urlaubsfotos in den Ordner `./datasets/monet2photo/testA/` und führen Sie folgenden Befehl aus:

```
python test.py ↵
  ↳--dataroot ./datasets/monet2photo ↵
  ↳--name monet2photo_cyclegan ↵
  ↳--model cycle_gan ↵
  ↳--preprocess none
```

Der Code geht davon aus, dass Sie die Berechnungen mit CUDA und einer Nvidia-GPU beschleunigen. Wenn Sie keine haben, können Sie aber auch den Parameter `--gpu_ids -1` hinzufügen, um nur die CPU zu nutzen. Mit mehreren positiven `--gpu_ids` nutzt der Code auch mehrere Grafikkarten.

Fertig trainierte Netze

Falls Sie keine Lust auf zeitaufwendiges Training haben, stehen unter der URL `http://efrosgans.eecs.berkeley.edu/cyclegan/pretrained_models/` einige vortrainierte neuronale Netze zur Verfügung (Dateiendung `.pth`). Um sie zu nutzen, legen Sie einen Ordner `checkpoints/<model_name>_pretrained/` an. In

diesen Ordner kopieren Sie die heruntergeladene `<model_name>.pth`-Datei und benennen sie in `latest_net_G.pth` um. Für Bash-Nutzer gibt es wie üblich ein Download-Skript im Repository, das diese Schritte übernimmt:

```
bash ./scripts/download_cyclegan_↵
                ↳model.sh <model_name>
```

Statt `<model_name>` setzen Sie auch beim Skript den Namen des Modells ein. Zur Verfügung stehen: `apple2orange` (und umgekehrt, also `orange2apple`), `cityscapes_label2photo` (und umgekehrt), `facades_label2photo` (und umgekehrt), `horse2zebra` (und umgekehrt), `iphone2dslr_flower`, `map2sat`, `sat2map`, `monet2photo`, `style_cezanne`, `style_monet`, `style_ukiyoe`, `style_vangogh`, `summer2winter_yosemite` und `winter2summer_yosemite`.

Damit der Code die passende Verzeichnisstruktur findet, laden Sie am besten wie oben beschrieben den entsprechenden Datensatz. Der Befehl fürs Inferencing sieht dann beispielsweise so aus (Datensatz `summer2winter_yosemite`, ohne CUDA auf der CPU):

```
python test.py --dataroot ./datasets/↵
       ↳summer2winter_yosemite/testA ↵
             ↳ --name summer2winter_↵
                ↳yosemite_pretrained ↵
             ↳ --model test ↵
             ↳ --no_dropout ↵
             ↳ --gpu_ids -1
```

Die Ergebnisse landen danach in einem Unterordner von `results/` mit dem Namen des Modells. Im Dateinamen steht jeweils, ob das Bild das Original oder das Fake ist.

Eigene Datensätze

Forscher haben CycleGANs bereits auf eigene Probleme angewendet. Darunter befinden sich sehr sinnvolle, wie beispielsweise „Smartphone2Profifotografie“ aber auch sehr seltsame wie „Gesicht2Ramen“ (Ramen ist eine japanische Nudelsuppe). Auch Sie können ein eigenes neuronales Netz mit individuellem Stiltransfer trainieren. Wie wäre es zum Beispiel mit einem Transfer von Porträtaufnahme zu Comiczeichnung? Dazu brauchen sie nur einen ausreichend großen Datensatz. Er sollte mindestens 1000 Bilder pro Klasse enthalten. Je mehr Bilder Sie verwenden, desto besser wird anschließend das Ergebnis.

Legen Sie dafür einen neuen Ordner `mein_datensatz` im Verzeichnis `datasets` an. Dieser Ordner muss die Unterordner `trainA` und `trainB` enthalten. In `trainA` kommen die Porträtfotos, in `trainB` die Comiczeichnungen.

Beim Training sollten alle Bilder gleich groß sein. Um einen ganzen Ordner an Bildern unter Linux mit Imagemagick auf 256 × 256 Pixel zu skalieren und zu beschneiden, können Sie folgendes `bash`-Skript verwenden:

```
cd ${1}_raw
list=`ls`
for img in $list; do
  name=`convert "$img" ↵
              ↳-format "%t" info:`
  convert "$img" -resize "256x256^" ↵
  ↳-gravity center -crop 256x256+0+0 ↵
          ↳../${1}/${name}_256x256.jpg
done
```

Das Skript rufen Sie mit einem Ordnernamen auf, beispielsweise `bash resize.sh trainA`. In diesem Fall sucht es die Quellbilder im Ordner `trainA_raw`. Diese beiden Ordner müssen Sie vorher per Hand anlegen.

Anschließend starten Sie das Training mit folgendem Befehl:

```
python train.py ↵
↳--dataroot ./datasets/mein_datensatz↵
↳ --name mein_datensatz_cyclegan↵
↳ --model cycle_gan
```

Nach dem Training nutzen Sie wie beschrieben das Test-Skript, um eigene Bilder in den Ordnern `testA` und `testB` umzuwandeln.

CycleGANs in der Praxis

Stiltransfer dient nicht nur algorithmischer Kunst. Nahe liegt die Nutzung in der Fotografie, die von einer automatischen Bildverbesserung in vielen Bereichen profitiert. Falls Sie sich ärgern, dass Sie Ihr Urlaubsfoto nicht in HDR aufgenommen hatten, können Sie mit der Technik nachträglich einen HDR-Effekt für das Bild erzeugen.

Auch die Schönheitsindustrie hat den Stiltransfer für sich entdeckt: Haben Sie in einem Schnappschuss nett gelächelt, aber vergessen sich zu schminken? Mit Stiltransfer können Sie virtuell ein Make-up samt Lippenstift auftragen lassen. Es gibt mittlerweile eine ganze Reihe von Beauty-Apps, die Sie sogar in Echtzeit auf Ihrem Smartphone

oder Smart-Mirror per Augmented Reality schminken, frisieren oder umkleiden.

Aber auch in der Strafverfolgung, etwa bei der Erstellung von Phantombildern, können Fotos von Tätern oder Opfern verändert werden, um Personen virtuell zu rasieren, sie mit verschiedenen Frisuren zu betrachten oder sie altern zu lassen. Und damit sind die Möglichkeiten der Technik längst nicht ausgeschöpft.

Sollten Sie noch Fragen zu CycleGANs und maschinellem Lernen haben, können Sie diese gern per Mail an wandt@heuwa.de stellen. (pmk) c't

Halluzinierte Trainingsdaten

Per Stiltransfer erzeugte Bilder lassen sich auch fürs Trainieren anderer neuronaler Netze nutzen. Oft besteht eine der größten Herausforderungen fürs Training von neuronalen Netzen nämlich im Sammeln von ausreichend vielen Daten. Große neuronale Netze, beispielsweise zur Objekterkennung, nutzen beim Training oft Millionen von Bildern. Die manuelle Erstellung der zu Fotos passenden Labels sprengt viele Kosten- und Zeitbudgets.

Ein aktuelles Forschungsthema ist daher die Datenaugmentierung. Bei der erstellen Computer automatisch Trainingsdaten für neuronale Netze. Algorithmen generieren dabei aus einem kleinen Satz an bereits vorhandenen Daten viele neue Daten. Stiltransfer-Algorithmen haben dafür bereits bewiesen, dass sie in der Lage sind, sehr gute Trainingsdaten zu erzeugen. Die Fachliteratur spricht oft davon, dass ein Algorithmus (meist ein neuronales Netz) diese neuen Daten „halluziniert". Beispielsweise erzeugt am Institut für Informationsverarbeitung der Leibniz-Universität Hannover im Rahmen eines Forschungsprojekts zum autonomen Fahren ein Stiltransfer-Algorithmus künstliche Straßenschäden in Bildern. Ein Klassifikator lernt damit, später diese Schäden zu erkennen. Der Stiltransfer fügt dafür in eigentlich unbeschädigte Straßen sehr realistisch aussehende Risse und Löcher ein.

Neue Algorithmen zur Erkennung von Straßenschäden kann man auf den künstlich erstellten, aber trotzdem realistisch aussehenden Daten testen. Selbst Videos aus Computerspielen wie GTA V wurden bereits mittels Stiltransfer in realistisch aussehende Videos verwandelt. Die verwenden dann Algorithmen zum autonomen Fahren zum Training. In Spielen und Simulatoren kann man damit Situationen simulieren, die in echten Aufnahmen nur selten geschehen. Beispielsweise erstellt man so beliebig viele Daten zu Unfällen, in denen Menschen unvorhersehbar auf die Straße laufen und mit dem Fahrzeug kollidieren. In der Simulation erzeugt man solche für ein sicheres System wichtigen Beispiele, ohne dass jemand in Gefahr gebracht wird.

Fügt man per CycleGAN in Bilder von intakten Straßen virtuelle Schäden ein, kann der entstehende Datensatz für andere neuronale Netze zum Training dienen.

Literatur

[1] Pina Merkert, Gesichtertausch, Wie man Video-Fakes erkennt und selbst berechnet, c't 8/2018, S. 104

Quellcode bei GitHub, Trainingsdaten:
www.ct.de/w3q3

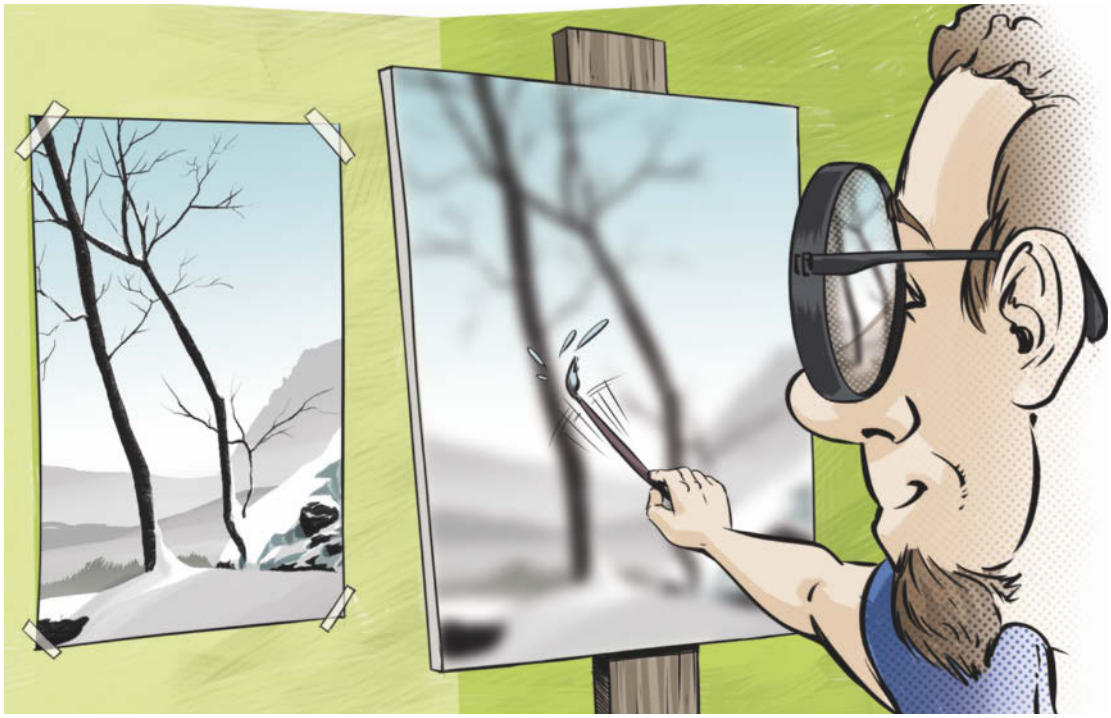


Bild: Albert Hulm

TensorFlow erkennt schlechte Bilder

Das Sammeln der Datensätze fürs Training von KI-Algorithmen kann ganz schön aufwendig sein. Berechnet man stattdessen die Trainingsdaten, spart das Arbeit und Speicher. Das KI-Framework Keras hilft bei der Implementierung in Python mit einer praktischen Basisklasse und parallelisiert die Berechnung für viele Prozessorkerne.

Von Pina Merkert

Der Urlaub ist vorbei und von der Speicherkarte wandern 4000 neue Urlaubsbilder auf die Festplatte. Der Finger am Auslöser war mal wieder nervös. Bei dieser Menge sind einige misslungene Schnappschüsse dabei: unscharf, verwackelt, verrauscht oder der Autofokus hat den falschen Bildteil ins Visier genommen. Die auszusortieren würde Stunden kosten – das muss besser gehen!

Wie schön wäre es, wenn der Rechner die unscharfen Bilder von alleine erkennen könnte. Mit einem einfachen Kantendetektor ist es aber nicht

getan: Der würde auch bei verrauschten Bildern anspringen. Stattdessen sollte der Rechner auf Kanten achten, die regelmäßige Strukturen in bestimmten Bereichen bilden. Solch grobe Vorgaben in einen Algorithmus zu gießen würde monatelanges Feintuning erfordern – keine attraktive Option. Worauf es beim Bildersortieren ankommt, soll der Rechner lieber an Beispielen selbst lernen.

Leider löscht man unscharfe Fotos vergangener Urlaube gleich, statt einen Trainingsdatensatz daraus zu machen. Zu den Tausenden scharfer Urlaubsfotos fehlen also die Negativbeispiele, an denen ein neuronales Netz die Unterschiede lernen könnte. Aber glücklicherweise ist es leichter, scharfe Bilder kaputt zu machen, als Misslungene zu retten. SciPy und ein paar Zeilen Python-Code zeichnen Bilder weich, verwackeln nachträglich und fügen Rauschen hinzu. Was dabei herauskommt, ist nicht schön und damit ideales Trainingsmaterial für das neuronales Netz.

Bilder sind allerdings groß und die Filter rechenaufwendig. Damit das Vermiesen der Bilder nicht das Training des Netzwerks ausbremst, sollte das parallel auf allen Kernen der CPU laufen – mit reinem Python wegen des Great Interpreter Lock (GIL) keine leichte Aufgabe. Das GIL sorgt nämlich dafür, dass Python-Threads sich zwar munter abwechseln, aber nie parallel laufen. Außerdem sollte der Code mit dem RAM haushalten und nicht die halbe Festplatte in den Arbeitsspeicher laden.

Glücklicherweise muss man für die Lösung dieses Problems kaum eigenen Code schreiben. Das Keras-API der Machine-Learning-Bibliothek TensorFlow enthält nämlich die `Sequence`-Klasse, mit der man mit minimalem Aufwand Datengeneratoren fürs Training der eigenen KI-Algorithmen schreibt. Die Generatoren klinken sich nicht nur effizient ins Training ein, sondern umgehen auch elegant das GIL, da TensorFlow die nötigen Threads aus C-Code startet. Die Keras-Dokumentation erklärt etwas zu knapp, wie `Sequence` funktioniert, aber den Makel behebt dieser Artikel.

Erst mal RAM sparen …

Damit TensorFlow den Datengenerator verwaltet, muss man lediglich eine Klasse erstellen, die von `Sequence` aus dem Modul `tensorflow.keras.utils` erbt. Die muss mindestens die Methoden `__len__()` und `__getitem__()` implementieren. Die Idee ist, dass der Datengenerator immer ganze Datenpakete (Batches) erzeugt. `__len__()` gibt die Anzahl dieser Batches zurück. `__getitem__()` ruft man mit einem Index zwischen 0 und dieser Anzahl auf, um einen bestimmten Batch zu erhalten. Der Batch besteht üblicherweise aus einem Tupel, zusammengesetzt aus einem Numpy-Array mit den Eingaben des KI-Algorithmus und einem zweiten Numpy-Array mit den gewünschten Ausgaben.

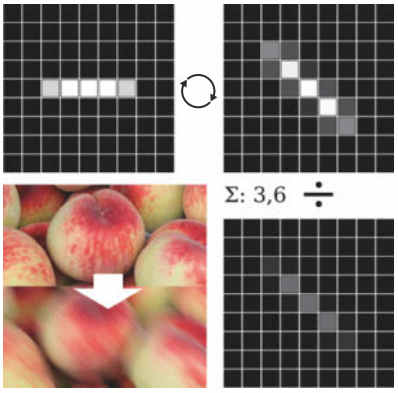
Ein ganz einfacher Datengenerator könnte beispielsweise im Konstruktor mit `os.listdir()` die Dateinamen von Beispielbildern auflisten und als Dictionary (Schlüssel `"filename"`) zusammen mit der Klasse (Schlüssel `"label"`) in `self.data` speichern. Mit der gewünschten Batchgröße lässt sich damit schon mal die Anzahl der Batches berechnen:

```
def __len__(self):
  return int(np.floor(len(self.data) /
                      self.batch_size))
```

Die Bilder selbst sollte der Datengenerator erst laden, wenn `__getitem__()` sie anfordert, da sie viel Speicher belegen. Die Funktion selektiert also zuerst die für den Batch benötigten Dateinamen:

Verwackeln mit Faltung

Beim Verwackeln vermischt eine Faltung die Farbwerte entlang einer Linie. Für den nötigen Filter zeichnet das Programm in ein 9 × 9 großes Bild eine Linie, dreht diese um einen zufälligen Winkel und teilt alle Farbwerte durch die Gesamthelligkeit. Dadurch bleibt die durchschnittliche Helligkeit des Bilds gleich.



```
def __getitem__(self, index):
  indexes = self.indexes[
       index * self.batch_size
       :(index + 1) * self.batch_size]
  filename_selection = [self.data[k]
                        for k in indexes]
  batch_x = []
  batch_y = []
  for d in filename_selection:
    img = imread(d["filename"])
    if d["label"] == 1:
      batch_y.append(np.array([0, 1],
                     dtype=np.float32))
    else:
      batch_y.append(np.array([1, 0],
                     dtype=np.float32))
  return (np.array(batch_x),
          np.array(batch_y))
```

Danach lädt die Funktion nur die zum Batch gehörenden Bilder, was so wenig Speicher wie möglich belegt.

Die zu den Bildern gehörenden Label `batch_y` erzeugt der Generator als One-Hot-Vektoren. Solche Vektoren enthalten eine 1 und sonst nur Nullen. Das entspricht dem Format einer Wahrscheinlichkeitsver-

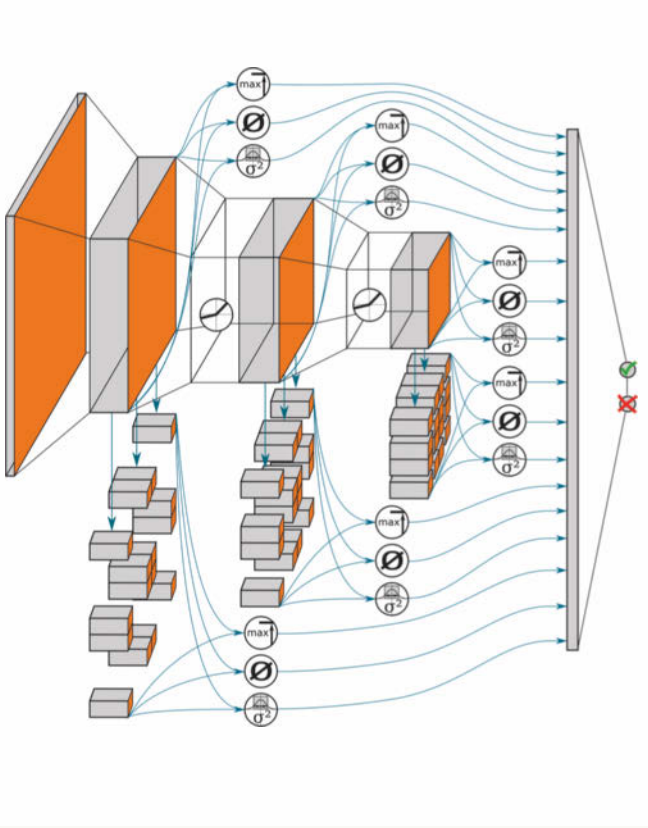
KI erkennt unscharf

Um unscharfe und verwackelte Fotos von scharfen Aufnahmen zu unterscheiden, verwenden handgeschriebene Algorithmen oft einen Laplace-Filter. Diesen Filter implementiert man üblicherweise als Faltung mit einer bestimmten 3x3-Matrix. Das Ergebnis dieser Operation ist ein Bild, das Kanten hell hervorhebt und in kontrastarmen Bereichen dunkel bleibt.

Berechnet man die Varianz dieses Bilds, also die mittlere quadrierte Differenz zwischen den Pixeln und der mittleren Helligkeit, ist dieser Wert bei scharfen Bildern höher als bei unscharfen Bildern. Enthält das Bild viel Rauschen, kommt allerdings ein noch höherer Wert heraus.

Convolutional-Layer eines neuronalen Netzes berechnen ohnehin Faltungen, können also Kantendetektoren wie den Laplace-Filter lernen. Sie können allerdings keine Varianz berechnen. Wir haben deswegen zwei eigene Layer mit der Keras-API implementiert: Einer berechnet die Varianz des gesamten Bilds, ein anderer die Varianz von Blöcken im Bild. Damit berechnet das neuronale Netz Werte, mit denen auch ein handgeschriebener Algorithmus Unschärfen erkennen könnte. Zusätzlich kann es wie Bilderkennungsnetzwerke lernen, Muster zu erkennen und die Varianzen dieser Muster berechnen.

Unsere Netzwerkarchitektur nutzt drei Schichten aus Convolutional-Layern mit durchlässigen Rectified-Linear-Units als Aktivierungsfunktion. Die ersten beiden Schichten überspringen bei jedem Schritt ein Pixel (strides=2), sodass sie jeweils



teilung: Die korrekte Antwort hat eine Wahrscheinlichkeit von 100 Prozent, falsche Antworten haben eine Wahrscheinlichkeit von Null. Diese Darstellung harmoniert gut mit Softmax-Ausgabefunktionen von neuronalen Netzen, da diese ebenfalls eine Wahrscheinlichkeitsverteilung ausgeben. Jede Abweichung zählt der Trainer als Fehler des Netzwerks.

Das Array mit Indexen (`self.indexes=np.arange(len(self.data))`) nutzt der Datengenerator, um die Reihenfolge der Bilder für jede Epoche mischen zu können. Durch die zufällige Reihenfolge nimmt der Gradientenabstieg in jedem Batch einen anderen Weg:

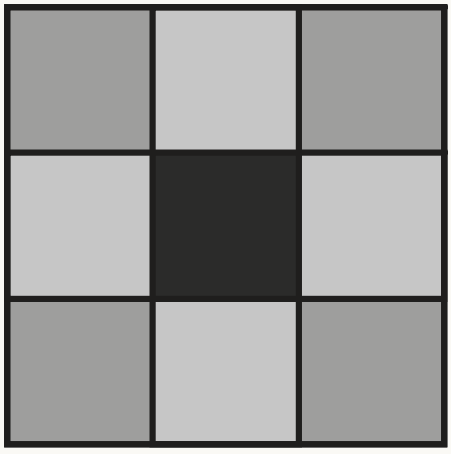
```
def on_epoch_end(self):
  self.indexes = np.arange(
                      len(self.data))
  np.random.shuffle(self.indexes)
```

Das Beispiel nutzt einen solchen einfachen Generator zum Validieren des Trainings (`ValidationDataProvider.py` im Repository zu finden über ct.de/wmh4).

… dann Bilder kaputt machen

Ein Datengenerator, der Bilder verunstaltet, benutzt die gleiche Grundstruktur, filtert die Bilder die Auflösung halbieren. Die Varianz, das Maximum und der Durchschnitt jedes dieser Layer darf die Ausgabeschicht als Feature benutzen.

Ein weiterer selbst geschriebener Layer extrahiert die Ecken, Seiten und die Mitte des Bilds, sodass das Netzwerk mit beliebigen Auflösungen der Eingabeschicht arbeiten kann. Für diese Ausschnitte berechnet das Netz ebenfalls Varianzen, Maxima und Durchschnitte, die alle der Ausgabeschicht als Features zur Verfügung stehen.



Der Laplace-Filter als Bild: Mittelgraue Felder bedeuten 0, Hellgrau 1 und Schwarz -4.

Der resultierende Feature-Vektor hat eine feste Breite, die nicht von den Abmessungen des Eingabebilds abhängt. Eine voll verbundene Ausgabeschicht mit Softmax-Aktivierungsfunktion berechnet die Wahrscheinlichkeit, dass ein Bild gut oder schlecht ist.

Trainiert haben wir mit Gradientenabstieg (Adam-Optimizer) und Kreuzentropie als Loss-Funktion. Das sorgt nach etwa zwei Epochen für Erkennungsraten um 80 Prozent. Danach lernt das Netz Besonderheiten der generierten Trainingsdaten, was die Erkennungsrate auf den Trainingsdaten auf über 90 Prozent steigert. Leider wirkt sich das leicht negativ auf die Erkennung echter Fotos aus, sodass das Netzwerk mit echten Daten unter 80 Prozent bleibt.

An ein vollautomatisches Löschen schlechter Fotos ist bei solchen Erkennungsraten nicht zu denken. Wir nutzen die KI stattdessen in einem grafischen Programm, um eine Vorauswahl vorzuschlagen. Das Programm nutzt Python und Qt, sodass es unter allen Desktopbetriebssystemen funktioniert. Um es zu nutzen, müssen Sie Qt5 und Python über ihr jeweiliges Setup installieren und anschließend mit pip die Abhängigkeiten in requirements.txt installieren (Tensorflow, pyQt5, Scipy etc.). Liegt das Netzwerk falsch, kann man die KI-Entscheidung im Programm einfach überschreiben. Das gebrauchsfertige Programm samt vortrainiertem neuronalem Netz finden Sie über ct.de/wmh4.

Der Datengenerator verschlechtert Bilder zufällig durch Verwackeln (1, 2), Weichzeichnen und Rauschen (3), Verwackeln und Weichzeichnen (4), sowie Weichzeichnen ohne (5) und mit Maske (6).

aber in `__getitem__()`. Fürs Filtern stellen SciPy und das dazu gehörende Modul Scikit-Image (`skimage`) die nötigen Funktionen zur Verfügung:

```
from skimage.transform import resize
from skimage.transform import rotate
from skimage.filters import gaussian
from scipy.ndimage.filters import convolve
```

Damit aus einer endlichen Anzahl verschiedener Bilder nahezu unbegrenzt viele unterschiedliche Bildausschnitte entstehen, skaliert der Datengenerator zuerst das Bild um einen zufälligen Faktor und wählt einen zufälligen quadratischen Ausschnitt:

```
min_scale_factor = max(
   self.target_size[0] / img.shape[0],
   self.target_size[1] / img.shape[1])
sized_img = resize(img,
       (int(img.shape[0] * sf),
        int(img.shape[1] * sf),
        img.shape[2]),
        mode='reflect')
crop_start_x=randrange(0, sized_img.␣
  ↳shape[1] - self.target_size[1] + 1)
crop_start_y=randrange(0, sized_img.␣
  ↳shape[0] - self.target_size[0] + 1)
img = sized_img[
   crop_start_y:
   crop_start_y + self.target_size[0],
   crop_start_x:
   crop_start_x + self.target_size[1],
   :].astype(np.float32)
```

Der Befehl zum Zuschneiden nutzt die Syntax für Listenausschnitte von Numpy. Die erlaubt Ausschnitte `[von:bis]` in mehreren Dimensionen gleichzeitig. Der Doppelpunkt bei der dritten Dimension übernimmt die Farbkanäle jedes Pixels unverändert.

Danach entscheidet der Generator zufällig, ob er das Bild scharf belässt oder es mit Filtern verschlechtert. Beim Verschlechtern wählt er zunächst zwischen den Optionen Weichzeichnen, Verwackeln oder beides. Das Weichzeichnen braucht nur einen Befehl:

```
gaussian(img, sigma=0.5+5.5*random(),
         multichannel=True)
```

Das Maß der Unschärfe legt `sigma=` fest. Das Minimum von 0,5 bewirkt eine leichte Unschärfe, die erst beim zweiten Blick auffällt, während das Maximum von 6 für ein stark verwaschenes Bild sorgt. Der Generator wählt einen zufälligen Wert aus diesem Bereich.

Verwackeln

Bei verwackelten Bildern fällt Licht, das bei einem scharfen Bild nur ein Pixel treffen würde, auf eine Linie von benachbarten Pixeln. Die Linie kann beliebig gedreht sein, ist aber meistens gerade. Je stärker der Fotograf gewackelt hat, desto länger ist die Linie und desto schlimmer die Bewegungsunschärfe.

Um diesen Effekt nachzuahmen, eignet sich eine Faltung. Interpretiert man die Kernel-Matrix als

kleines Graustufenbild, zeigt er eine Linie um den Mittelpunkt, deren Drehung und Länge der Bewegungsunschärfe entspricht. Damit sich durch die Faltung an der Helligkeit des Bildes nichts ändert, müssen sich alle Einzelwerte im Kernel zu 1 addieren.

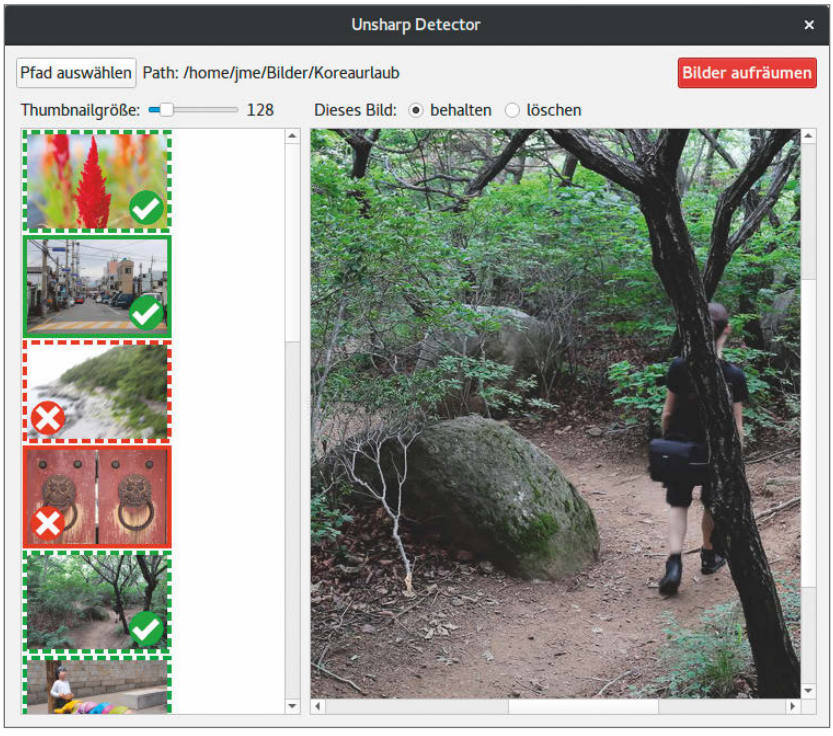
Der Generator wählt dafür zuerst eine zufällige Länge zwischen 2,5 und 9 Pixeln. Dann initialisiert er einen 9x9-Kernel mit Nullen und setzt mittig in diesen eine aus Einsen bestehende horizontale Linie in der gewählten Länge. Anschließend dupliziert er diesen Kernel dreimal, um später gleichzeitig alle drei Farbkanäle falten zu können. Danach dreht er den Kernel mit rotate() um einen zufälligen Winkel. Nachfolgend teilt er den Kernel durch seine Summe, um die ursprüngliche Helligkeit wiederherzustellen. Im letzten Schritt faltet er das Bild mit dem Kernel (convolve()):

```
kernel = np.zeros((9, 9),
                  dtype=img.dtype)
shake_len = random()*6.5+2.5
kernel[4, 4] = 1.0
for i in range(1, 5):
  x = (shake_len - i * 2 + 1) / 2
  kernel[4+i, 4] = x
  kernel[4-i, 4] = x
kernel = np.clip(filter_matrix, 0, 1)
kernel = np.repeat(
   kernel.reshape(kernel.shape[0],
                  kernel.shape[1], 1),
   3, axis=2)
kernel = rotate(kernel,
           random() * 360,
           mode='constant', cval=0.0)
kernel = kernel / kernel.sum()
img = convolve(img, kernel,
              mode='reflect')
```

Maskieren

Ein häufiges Problem bei schlechten Urlaubsfotos ist der Autofokus, der bei Porträts auf den Hintergrund statt aufs Gesicht fokussiert. Dann ist ein Teil des Bildes zwar perfekt scharf, in der Mitte lächelt aber nur ein unscharfer Blob. Ein solches Bild kann man nicht einfach aus einem scharfen Porträt berechnen. Man kann die KI aber trainieren, Bilder mit unscharfen Bereichen in der Mitte abzulehnen.

Dafür erzeugt der Generator eine kleine Maske als zweidimensionales Numpy-Array. Die enthält in



Die Einschätzung der KI, wie scharf ein Bild ist (gestrichelte Umrandung), kann man per Hand überschreiben (durchgezogene Linie), falls das neuronale Netz einen Fehler gemacht hat.

der Mitte Einsen und in den Ecken Nullen. Beim Skalieren auf die Auflösung der Trainingsbilder werden die kontrastreichen Kanten zu weichen Übergängen. Ähnlich wie beim Verwackeln dupliziert der Generator die Maske für alle drei Farbkanäle. Im letzten Schritt multipliziert er das unscharfe Bild mit der Maske und das scharfe Bild mit 1 minus der Maske. Das Beispiel wendet die Maske zufällig in einem von fünf Fällen an.

Rauschen

Bei ebenfalls 20 Prozent der verschlechterten Bilder fügt der Generator Rauschen hinzu. Im Prinzip muss er dafür nur mit `np.random.randn()` normalverteiltes Rauschen erzeugen und zu den Farbwerten im Bild addieren. Das Rauschmuster ist aber für Menschen leicht vom Rauschen echter Bildsensoren zu unterscheiden. Deswegen zeichnet der Generator das Muster mit `gaussian()` noch weich. Wie stark bestimmt der Zufall (Sigma=0,1 bis 1,2):

```
def add_noise(img):
  noise = np.random.randn(*img.shape)
          ↳ * (0.05 + 0.1 * random())
  noise = gaussian(noise,
           sigma=0.1 + 1.1 * random(),
           multichannel=True)
  return np.clip(img+noise, 0, 1)
```

Datengeneratoren einsetzen

Den fertigen Datengenerator (siehe `TrainingDataGenerator.py` im Repository) zu nutzen ist leicht. Man übergibt dem neuen Objekt den Pfad zu den Bildern, die Batch- und die Bildgröße:

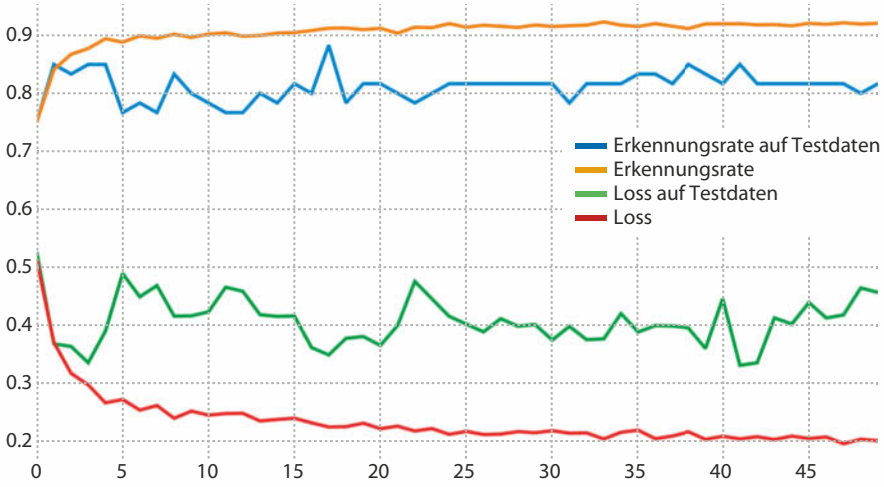
```
data_generator =
       ↳UnsharpTrainingDataGenerator(
              image_folders,
              batch_size=12,
              target_size=(256, 256))
```

Ein Mischen der Bildreihenfolge erzwingt man danach mit `data_generator.on_epoch_end()`, falls man das möchte.

Danach trainiert man lediglich mit `fit()` und übergibt den Generator als `x`. Ein `y` darf man dann

Kein Overfitting

Wenn Loss und Erkennungsrate nur auf den Trainingsdaten und nicht beim Testen besser werden, ist das normalerweise ein Zeichen für Overfitting. Hier lernt die KI aber nicht Datensätze auswendig, sondern erkennt die Unterschiede zwischen generierten Daten und echten Fotos.



nicht mehr angeben, da der Generator ja Eingaben und Labels gemeinsam erzeugt:

```
model.fit(
            x=data_generator,
            epochs=50,
            use_multiprocessing=True,
            workers=8,
            max_queue_size=30)
```

Um mehrere Kerne auszulasten, erwartet TensorFlow die Option `use_multiprocessing=True`, eine Anzahl an Threads und eine Größenangabe für die Warteschlange. Acht Arbeiter und 30 Schlangenplätze lasten einen Vierkerner mit Hyper-Threading aus. Unser GPU-Testserver mit zwei Skylake-CPUs hat 64 Prozessorkerne, sodass wir dort 128 Threads starten und die Warteschlange auf 256 Einträge verlängern.

Die vielen Threads sind nötig, da die SciPy-Filter sehr viel Rechenzeit verschlingen. Trainiert man auf einer GPU, langweilt die sich, wenn der Prozessor nicht schnell genug Trainingsbeispiele beischafft.

Repository, Dokumentation:
www.ct.de/wmh4

Lohnende Sparsamkeit

Durch die speichersparende Implementierung empfehlen sich Datengeneratoren für große Datensätze. Bei der Bilderkennung ist diese Sparsamkeit oft der einzige Grund, warum das Training überhaupt funktioniert. Je weniger Speicher die Daten brauchen, desto mehr RAM bleibt für die Parameter der neuronalen Netze übrig. Experimente mit großen Netzen klappen daher auf vielen Rechnern nur dank Generator.

Je mehr Variation ein Datengenerator in die Datensätze bringt, desto weniger kann ein neuronales Netz Daten auswendig lernen. Damit verhindert man effektiv Overfitting. Es entsteht allerdings ein neues Problem: Die generierten Daten könnten sich zu stark von echten Daten unterscheiden. In den Lernkurven sieht das ähnlich aus, hat aber andere Ursachen. In vielen Fällen verfügt man aber ohne den Generator ohnehin über keinen ausreichend großen Datensatz. (pmk) c't

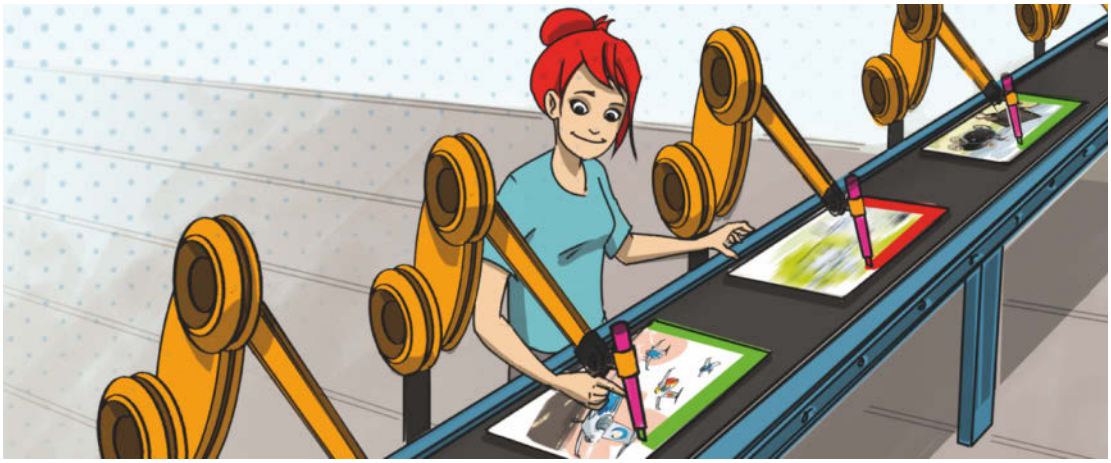


Bild: Thorsten Hübner, illustrator

Listen in der PyQt5-GUI zur KI

Im Handumdrehen entstehen mit PyQt grafische Anwendungen, die auf allen Desktopbetriebssystemen laufen. Doch wenn sich die Länge von Listen mit Widgets zur Laufzeit ändern soll, sollte man auf das in Qt enthaltene Model-View-Framework zurückgreifen. Das gibt die Struktur für eine performante Implementierung vor.

Von Pina Merkert

Wir haben ein neuronales Netz trainiert, um automatisch unscharfe Urlaubsbilder zu löschen (siehe Seite 104). Da diese KI manchmal Fehler macht, haben wir mithilfe von PyQt5 ein grafisches Interface programmiert, das bei jedem Urlaubsbild anzeigt, ob die KI es löschen möchte oder nicht. Die Entscheidung der KI kann man mit einem Mausklick überschreiben.

Das Programm besteht aus einem Button, um einen Ordner voller Bilder zu laden, einem Vorschaufenster, um einzelne Bilder in voller Auflösung sehen zu können, und einer Liste mit Thumbnails zu jedem Bild. Diese Liste enthält alle Bilder im Ordner. Wie viele das sind, entscheidet sich erst, wenn der Benutzer mit dem Laden-Dialog einen Ordner ausgewählt hat. Daher kann man nicht einfach ein paar Thumbnail-Widgets ins Layout ziehen.

Abhilfe schafft die `QListView`. Sie gehört zum seit Qt4 integrierten Model-View-Framework und zeigt Daten an, die ein `QAbstractListModel` liefert. Wie die Daten dargestellt werden, entscheidet `QListView` nicht selbst, sondern delegiert die Aufgabe an eine von `QAbstractItemDelegate` abgeleitete Klasse. Die ist selbst kein `QWidget`, hat aber eine `paint()`-Methode, um das Aussehen der Daten in der Liste zu bestimmen.

Die etwas verwirrende Struktur nutzt Qt aus Performancegründen. Listen und Tabellen enthalten nämlich oft mehr Daten, als das GUI-Framework

gerade darstellen muss. Renderte es intern alle unsichtbaren Daten, würde das viel Rechenzeit und Arbeitsspeicher verschwenden. Stattdessen sorgt Qt automatisch dafür, dass nur die Delegaten zum Zug kommen, die auch sichtbar sind.

Daten ins Model

Das Model-View-Framework orientiert sich am MVC-Konzept und versucht die Daten daher unabhängig von der Anzeige zu halten. Die Model-Klasse, die die Daten aufbewahrt, erbt dafür von der Qt-Klasse `QAbstractListModel`. Implementieren muss man mindestens die Methoden `rowCount()` und `data()`. Wir haben uns dafür einen Wrapper um eine Python-Liste geschrieben, die im Prinzip jeden Datentyp aufnimmt. Die Methode `rowCount()` gibt in unserem Fall schlicht die Länge der Liste zurück.

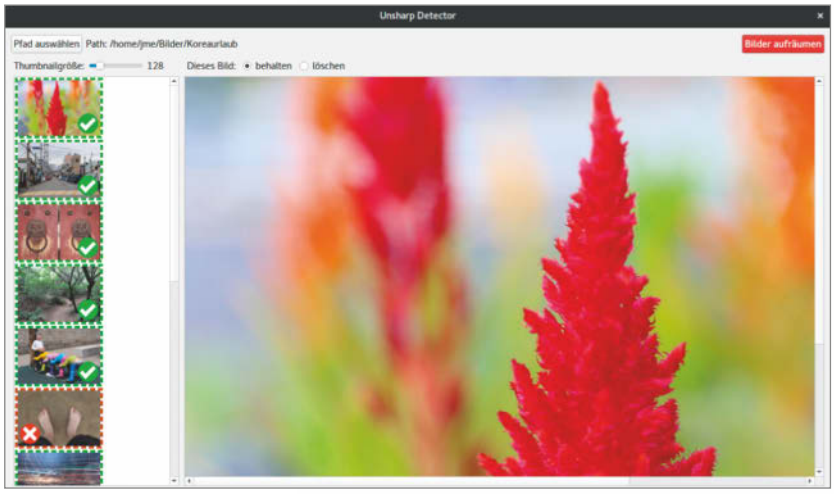
Der Methode `data()` übergibt Qt einen `index` und sie liefert den zum Index passenden Eintrag. Der übergebene `index` ist allerdings keine Ganzzahl, sondern ein `QModelIndex`-Objekt. Das enthält in `row()` aber eine Ganzzahl, die man stattdessen benutzen kann:

```
def data(self, index, role=None):
 return self.list[index.row()]
```

Falls Daten in der Liste überschrieben werden, möchte man die `QListView` über die Änderung informieren. Das `QAbstractListModel` bringt dafür das Signal `dataChanged` mit. Das transportiert `QModelIndex`-Objekte für den Beginn und das Ende des geänderten Bereichs. Aktualisiert man nur einen Eintrag der Liste, ist das zweimal derselbe Index. Das `QModelIndex`-Objekt erzeugt man am leichtesten mit der Factory-Methode `createIndex()`:

```
def data_changed(self, item):
  model_index = self.createIndex(
       self.list.index(item), 0, item)
  self.setData(model_index, item)
def setData(self, model_index, data,
                    role=Qt.EditRole):
  super().setData(model_index,
                  data, role=role)
  self.dataChanged.emit(
    model_index, model_index, [role])
```

Die eigene Funktion `data_changed` verknüpfen wir später mit einem gleichnamigen Signal aus den Datensätzen. Sie illustriert hier, wie man den für `setData()` nötigen `QModelIndex` erzeugt. `setData()` erweitert die Methode der Elternklasse, indem sie zusätzlich das `dataChanged`-Signal aussendet.



Unser KI-Bildsortierer zeigt in der linken Leiste eine Liste mit Thumbnails an. Das Aussehen der Listeneinträge bestimmt eine Python-Klasse. Die Länge der Liste hängt daran, wie viele Bilder das Programm unter dem gewählten Pfad findet.

Standarddelegierte

Da `QListView` die Listeneinträge nicht selbst zeichnet, braucht sie immer einen Delegaten, der das übernimmt. Das ist eine einzelne Klasse, die man mit `setItemDelegate()` setzt. Man übergibt der Funktion dafür nicht die Klasse selbst, sondern ein bereits instanziiertes Objekt. Qt kümmert sich darum, dieses mit den Daten der einzelnen Listeneinträge zu versorgen.

Stecken in einer Liste verschiedene Datentypen, muss derselbe Delegat mit allen davon umgehen können. Für übliche Datentypen wie Zahlen und Strings bringt Qt eine fertige Klasse mit, die alle Typen anzeigen kann. Da sie ihr Aussehen an die aktuell gewählte Qt-Stilvorlage anpasst, heißt die Klasse `QStyledItemDelegate`. Die Klasse ist keine abstrakte Klasse. Das heißt, Sie können sie direkt instanziieren, ohne weitere Methoden zu implementieren. Das reicht, falls Sie nur Standarddatentypen in der Liste anzeigen und editieren möchten.

Eigene Datentypen

Für unseren Bildersortierer haben uns Standarddatentypen nicht ausgereicht. Die Listeneinträge bestehen nämlich aus einem Bild und einem

Thumbnail (beides `QImage`), den Eingabedaten für das neuronale Netz (Numpy-Array), dem Dateinamen, der Entscheidung des Netzes oder des Nutzers und Statuswerten für die Anzeige. Wir haben diese Sammlung `ClassifiedImageBundle` genannt und von der Qt-Basisklasse `QObject` abgeleitet.

Solche Objekte fügen sich nahtlos ins Typsystem von Qt ein. Um Änderungen an den Daten zu überwachen, können Sie mit `pyqtSignal()` Signale definieren, zu denen sich Widgets oder Delegaten verbinden können. Falls eine solches Objekt Berechnungen beim Abruf von Properties ausführen soll, definiert man die Getter und Setter als `pyqtProperty()`. Wir haben das für `QPropertyAnimation` genutzt (dazu später mehr).

Delegaten erweitern

Mit dem selbst definierten Datentyp kann `QStyledItemDelegate` natürlich nichts anfangen. Aber statt das Rad neu zu erfinden, kann man ja einfach von der Klasse erben. Überschreiben sollte man zumindest die Methoden `paint()` und `sizeHint()`.

Beim Überschreiben der Methoden muss man nur herausfinden, um welchen Datentyp es sich bei dem Listeneintrag handelt. Dafür extrahiert man mit der `data()`-Methode aus dem `QModelIndex` das Datenobjekt und prüft mit `type()`, zu welcher Klasse es gehört. Für eigene Datentypen berechnet man die Antwort direkt in der Methode, für alle anderen Datentypen lässt man mit `super()` die Basisklasse entscheiden und reicht das Ergebnis durch.

Am `sizeHint()` erkennt man das Vorgehen recht gut. Falls der Delegat die Größe zur Darstellung eines `ClassifiedImageBundle` zurückgeben soll, gibt die Funktion die Größe des Thumbnails plus 8 Pixel Rahmen zurück. Bei allen anderen Datentypen übernimmt sie die Größenangabe der Basisklasse:

```
def sizeHint(self,
 style_option_view_item, model_index):
  mid = model_index.data()
  midtype = type(mid)
  if midtype is ClassifiedImageBundle:
    return QSize(
         mid.get_thumb().width() + 8,
         mid.get_thumb().height() + 8)
  else:
    return super().sizeHint(
              style_option_view_item,
             model_index)
```

Delegaten zeichnen jeweils einen Teil der Liste. Dabei muss man auf alle Zeichenoperationen ein Offset addieren, da sich die Delegaten sonst gegenseitig übermalen.

In der `paint()`-Methode läuft das im Prinzip genauso, nur ohne `return`.

Zeichenregeln

Zeichenoperationen nutzen in Qt Objekte vom Typ `QPainter`. Die bieten Funktionen für Linien, Kreise, Text, und Pixmaps und kümmern sich intern darum, die Anweisungen effizient bis zum Display durchzureichen. Aus Performance-Gründen ergibt es Sinn, nicht für jeden Strich einen eigenen `QPainter` zu erzeugen. Die `QListView` übergibt daher ihren `QPainter`, mit dem sie Umrandungen oder Scrollbalken zeichnet, an jeden der Delegaten. Die fügen mit dem einfach ihre Zeichenanweisungen hinzu.

Damit dabei kein Chaos mit Farben und Strichstilen entsteht, enthält jeder `QPainter` einen Stack. Mit `save()` speichert man die aktuellen Einstellungen auf diesem Stack und mit `restore()` setzt man die Einstellungen auf den zuletzt gespeicherten Stand zurück. Daher sollte der Delegat mit dem übergebenen `QPainter`-Objekt `qp` vor dem ersten

Zeichenbefehl `qp.save()` und nach dem letzten Zeichenbefehl `qp.restore()` aufrufen:

```
def paint(self, qp,
 style_option_view_item, model_index):
  mid = model_index.data()
  midtype = type(mid)
  if midtype is ClassifiedImageBundle:
    qp.save()
    svir = style_option_view_item.rect
    qp.drawImage(svir.left() + 4,
      svir.top() + 4, mid.get_thumb())
    # ... Weitere Zeichenbefehle ...
    qp.restore()
  else:
    super().paint(qp,
                  style_option_view_item,
                  model_index)
```

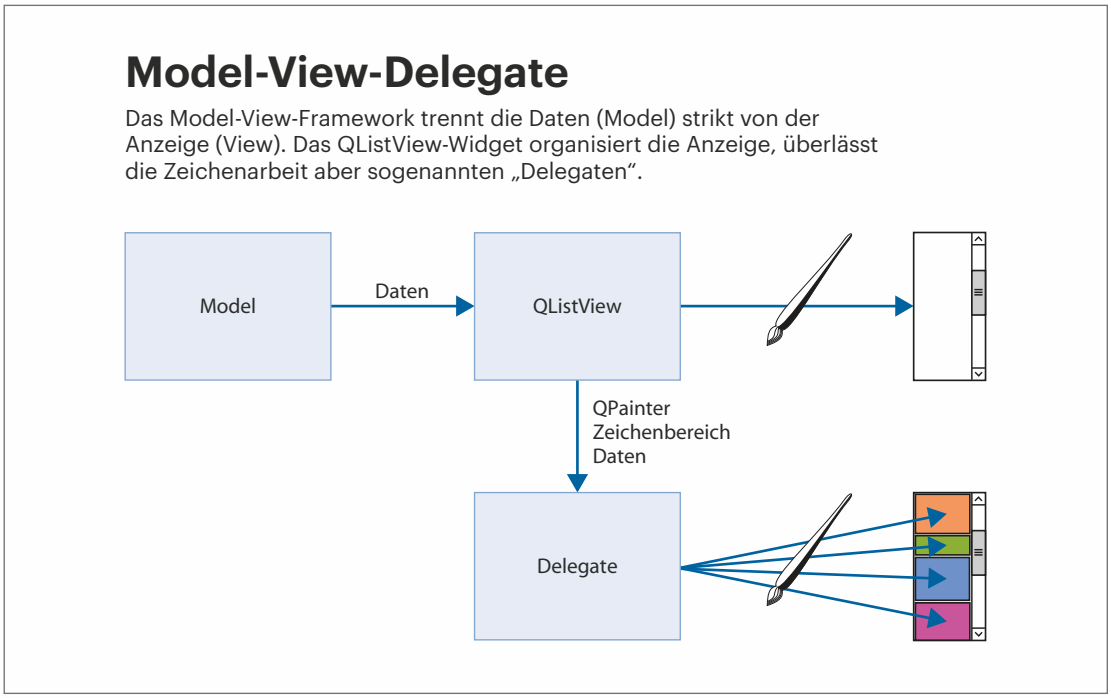
Da der Delegat einen bestehenden `QPainter` übernimmt, liegt der Nullpunkt des Koordinatensystems nicht in der linken oberen Ecke des Listeneintrags, sondern in der linken oberen Ecke des sichtbaren Teils der Liste. Den Zeichenbereich, in den der Delegat zeichnen sollte, erfährt er mit dem zweiten übergebenen Objekt `style_option_view_item`. Das speichert in der Eigenschaft `rect` ein `QRect`, das den Zeichenbereich definiert. Zeichnet der Delegat darüber hinaus, übermalen die Listeneinträge sich gegenseitig.

Mausereignisse

Damit Anwender bereits in der Vorschau Bilder zum Löschen oder Behalten auswählen können, reagieren die Delegaten auch auf Mausereignisse. Normalerweise empfangen Delegaten nur Klicks. Ruft man aber bei `QListView` die Methode `setMouseTracking(True)` auf, erfährt der Delegat auch, wenn die Maus nur darüber schwebt.

Die Ereignisbehandlung übernimmt die Methode `editorEvent()`, die vier Parameter erhält: zuerst ein Event-Objekt, von dem man erfährt, welche Maustasten gedrückt wurden. Danach das Model-Objekt, mit dem man beispielsweise Werte für die ganze Liste setzen kann. Anschließend ein `style_option_view_item`, mit dem man wie in der `paint()`-Methode lokale Koordinaten ausrechnen kann. Zuletzt ein `QModelIndex`, über den man mit `data()` an einen einzelnen Eintrag der Liste kommt.

Schwebt die Maus über dem Listeneintrag, hat `event.type()` den Wert `QEvent.MouseMove`. Wird ein



Mausbutton gedrückt, gibt die Funktion stattdessen `QEvent.MouseButtonPressed` zurück. Um nur auf den linken Mausbutton zu reagieren, prüft man zusätzlich, dass `event.button()` den Wert `Qt.LeftButton` hat:

```
if event.type() == ↵
    ↳QEvent.MouseButtonPress and ↵
    ↳event.button() == Qt.LeftButton:
pass
```

Urlaubsbilder sortieren

Um Ihre Urlaubsbilder mit unserer KI-getriebenen Qt-Anwendung zu sortieren, brauchen Sie Python, PyQt und TensorFlow.

Laden Sie zuerst eine 64-Bit-Version von Python 3 von Python.org herunter und installieren es. Linuxer können diesen Schritt überspringen, da Python dort üblicherweise vorinstalliert ist.

Den Code können Sie als ZIP-Datei herunterladen (siehe ct.de/wvkh) oder das Repository mit Git auschecken. Windows- und Mac-Nutzer laden Git von git-scm.com, unter Linux kümmert sich die Paketverwaltung um die Installation.

Sobald Sie den Code entpackt oder ausgecheckt haben, sollten sie ein Virtualenv anlegen, damit die weiteren Schritte keine bestehenden Python-Programme stören können. Das Virtualenv erzeugen Sie unter macOS und Linux mit folgendem Befehl, den Sie im Verzeichnis des Codes ausführen:

```
python3 -m venv env
```

Leider unterstützt TensorFlow Python bisher nur bis Version 3.7. Die Unterstützung für Python 3.8 ist aber in Nightly-Builds bereits vorhanden. Windows findet `python.exe` nicht automatisch, sodass Sie in der Eingabeaufforderung den Pfad angeben müssen:

```
..\..\AppData\Local\Programs\Python\↵
    ↳Python36\python.exe -m venv env
```

Starten Sie das Virtualenv unter Windows mit `env\Scripts\activate.bat` und unter Linux und macOS mit `source env/bin/activate`.

Um alle weiteren Abhängigkeiten kümmert sich Pythons Paketmanager `pip`:

```
pip install -r requirements.txt
```

Falls Sie eine Nvidia-Grafikkarte einsetzen und CUDA installiert haben, können Sie stattdessen `requirements-gpu.txt` verwenden. Dann nutzt TensorFlow die GPU für das neuronale Netz.

Nutzen

Um das Programm zu nutzen, müssen Sie zuerst wie bei der Installation das Virtualenv aktivieren. Danach führen Sie `python inference_gui.py` aus.

Das Programm startet mit einer leeren Liste an Bildern. Die füllen Sie, indem sie mit dem Button „Pfad auswählen“ in der linken oberen Ecke einen Ordner auswählen. Das Programm lädt dann alle Bilder im Ordner, was bei vielen und großen Bildern einige Sekunden dauern kann. Die Analyse der Bilder beginnt sofort im Hintergrund. Sie können aber auch sogleich mit der Maus Bilder zum Behalten oder Verwerfen markieren. Alle, die Sie nicht per Hand markieren, analysiert das neuronale Netz. Dessen Einschätzung erkennen Sie am gestrichelten Rahmen um das Bild.

Falls Sie am Thumbnail nicht erkennen können, ob ein Bild gut oder schlecht ist, klicken Sie einfach auf den Thumbnail. Das Programm zeigt dieses Bild dann in voller Auflösung im Vorschaubereich an.

Wenn Sie mit der Einordnung aller Bilder zufrieden sind, drücken Sie den roten „Bilder aufräumen“-Button in der oberen rechten Ecke. Vorsicht: Die zum Löschen vorgemerkten Bilder werden dann ohne weitere Nachfrage von der Festplatte entfernt.

Instabilitäten

Hin und wieder ist unser Programm mit einem Segmentation-Fault abgestürzt. Der Fehler hängt entweder mit Qt oder TensorFlow zusammen. Er scheint zufällig aufzutreten. In diesem Fall starten Sie das Programm einfach noch mal und versuchen es erneut. Wenn Sie eine Idee haben, woran das liegt, schreiben Sie uns bitte eine Mail an pmk@ct.de.

Danach braucht man nur noch die X- und Y-Koordinate des Mauszeigers innerhalb des Eintrags:

```
x_in_delegate = event.pos().x() - ↵
   ↳style_option_view_item.rect.left()
y_in_delegate = event.pos().y() - ↵
   ↳style_option_view_item.rect.top()
```

Den gesamten Code des Beispiels finden Sie über ct.de/wvkh im Repository in der Datei `extended_qt_delegate.py`.

Animierte Delegierte

Während die KI über der Qualität der Bilder sinniert, soll die Anwendung den Nutzer mit einer Animation unterhalten. Da der Delegat aber nur Aufgaben für den `QListView` übernimmt, kann er sich selbst nicht neu zeichnen. Qt sieht aber einen Mechanismus vor, der Teile der Liste neu zeichnet, falls sich Daten ändern. Darum kümmert sich statt des Delegaten selbst das Model, indem es `dataChanged`-Signale versendet.

Uns war es lieber, wenn die Objekte im Model sich selbst animieren. Dafür definierten wir den Fortschritt der Animation (Gleitkommazahl zwischen –1 und 1) als Property:

```
animation_progress = pyqtProperty(
        float, get_animation_progress,
               set_animation_progress)
```

Die lässt sich dann per `QPropertyAnimation` animieren. Sobald man bei diesem Animations-Objekt die `start()`-Methode aufruft, setzt es die Eigenschaft `animation_progress` ständig neu. Das ruft auch die Methode `set_animation_progress()` auf, zeichnet aber nichts neu.

Als Nächstes muss das Model von der Änderung erfahren. Das Datenobjekt definiert dafür das Signal `data_changed`, das es bei jeder Änderung am Animationsfortschritt aussendet. Das Modell empfängt dieses Signal mit der Methode `data_changed()`. Die kennen Sie bereits. Sie erzeugt den `QModelIndex` zum Eintrag und versendet ein `dataChanged`-Signal, das endlich ein erneutes Zeichnen des Eintrags auslöst.

Mit dieser Struktur reicht es dann, `animation_progress` in der `paint()`-Methode des Delegaten auszuwerten. Die Beispielanwendung berechnet damit den Start- und Endpunkt einer gestrichelten Umrandungslinie. Die wandert dank der Property um den Thumbnail, während das neuronale Netz rechnet.

Gute Schwuppdizität

Im ersten Moment erscheint das Model-View-Framework unnötig kompliziert. Beim genaueren Hinsehen entpuppt sich die Gängelei des Frameworks aber als sinnvolle Struktur, um performante Anwendungen zu schreiben. Die brechen dann auch nicht zusammen, wenn eine Liste mal etwas länger wird, weil Qt stets nur das Nötigste berechnet. Die Logik dafür nicht programmieren zu müssen, wiegt die Einarbeitungszeit ins Model-View-Framework mehr als auf. (pmk) ct

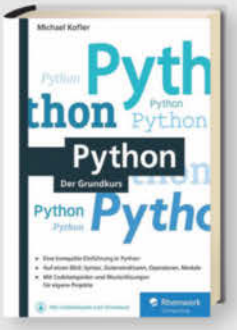
Repository:

ct.de/wvkh

Für Wissenshungrige

Ausgewählte Fachliteratur

shop.heise.de/buecher



Michael Kofler

Python

Diese Python-Einführung konzentriert sich auf das Wesentliche und zeigt Ihnen, wie Sie die Sprache in eigenen Projekten einsetzen. Erfahren Sie praxisgerecht, wie Sie mit Python Daten verarbeiten, den Raspberry Pi ansteuern, wiederkehrende Aufgaben automatisieren und vieles mehr.

ISBN 9783836266796
shop.heise.de/python-buch

14,90 € ›

BEST-SELLER

Christian Solmecke, Sibel Kocatepe

DSGVO für Website-Betreiber

Ihr Leitfaden für die sichere Umsetzung der EU-Datenschutz-Grundverordnung. Experten erklären Schritt für Schritt, wie Sie Ihren Webauftritt vollständig rechtskonform gestalten – gut verständlich auch für Nichtjuristen.

ISBN 9783836267120
shop.heise.de/dsgvo-websites

39,90 € ›

BEST-SELLER

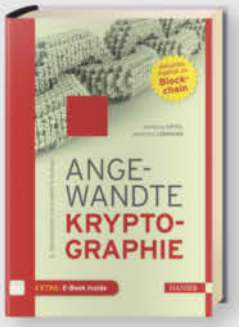
Jörg Frochte

Maschinelles Lernen (2. Auflg.)

Maschinelles Lernen ist ein interdisziplinäres Fach, das die Bereiche Informatik, Mathematik und das jeweilige Anwendungsgebiet zusammenführt.
In diesem Buch werden alle drei Teilgebiete gleichermaßen berücksichtigt.

ISBN 9783446459960
shop.heise.de/maschinelles-lernen

38,00 € ›



Wolfgang Ertel, Ekkehard Löhmann

Angewandte Kryptographie

Ziel des Buches ist es, Grundwissen über Algorithmen und Protokolle zu vermitteln und kryptographische Anwendungen aufzuzeigen. Mit so wenig Mathematik wie nötig, aber vielen Beispielen, Übungsaufgaben und Musterlösungen.

ISBN 9783446454682
shop.heise.de/kryptographie

32,00 € ›

NEU

Michael Kofler, Charly Kühnast, Christoph Scherbeck

Raspberry Pi, 6. Auflage

Das umfassende Handbuch mit über 1.000 Seiten komplettem Raspberry-Wissen, um richtig durchstarten zu können. Randvoll mit Grundlagen und Kniffen zu Linux, Hardware, Elektronik und Programmierung.
Aktuell für alle Versionen, inkl. Raspberry Pi 4!

ISBN 9783836269339
shop.heise.de/raspberry-6

44,90 € ›

Hans-Georg Schumann

Calliope mini für Kids

Die wichtigsten Bestandteile des Calliope mini mit allen Sensoren kennenlernen und ausprobieren. Mit vielen kleinen Calliope-Projekten für die Schule und zu Hause wie Würfelspiele, Farbthermometer, Alarmanlage, Wasserwaage, Funkgerät uvm.

ISBN 9783958458598
shop.heise.de/calliope-kids

19,99 € ›

PORTOFREI
AB 15 €
BESTELLWERT

› Generell portofreie Lieferung für Heise Medien- oder Maker Media Zeitschriften-Abonnenten oder ab einem Einkaufswert von 15 €. Nur solange der Vorrat reicht. Preisänderungen vorbehalten.

und Maker!

Zubehör und Gadgets

shop.heise.de/gadgets

Waveshare Game HAT für Raspberry Pi

Ein Muss für jeden Retro Gamer! Verwandeln Sie Ihren Raspberry Pi in kürzester Zeit in eine Handheld-Konsole. Mit Onboard-Speakern, 60 Frames/s, Auflösung von 480x320 und kompatibel mit allen gängigen Raspberrys.

shop.heise.de/game-hat **41,90 € ›**

Raspberry Pi-Kameras

Aufsteckbare Kameras, optimiert für verschiedene Raspberry Pi-Modelle mit 5 Megapixel und verschiedenen Aufsätzen wie z. B. Weitwinkel für scharfe Bilder und Videoaufnahmen.

shop.heise.de/raspi-kameras **ab 18,50 € ›**

NVIDIA Jetson nano

Das Kraftpaket bietet mit 4 A57-Kernen und einem Grafikprozessor mit 128 Kernen ideale Voraussetzungen für die Programmierung neuronaler Netze, die ähnlich wie Gehirnzellen arbeiten.
Inklusive Netzteil!

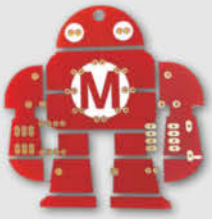
shop.heise.de/jetson **134,90 € ›**

ArduiTouch-Set

Setzen Sie den ESP8266 oder ESP32 jetzt ganz einfach im Bereich der Hausautomation, Metering, Überwachung, Steuerung und anderen typischen IoT-Applikationen ein!

~~69,90 €~~

shop.heise.de/arduitouch **36,90 € ›**



Makey Lötbausatz

Hingucker und idealer Löt-Einstieg: das Maskottchen der Maker Faire kommt als konturgefräste Platine mitsamt Leuchtdiodendie, die den Eindruck eines pulsierenden Herzens erwecken.

Jetzt neu mit Schalter!

shop.heise.de/makey-bausatz **ab 4,90 € ›**

Komplettset Argon ONE Case mit Raspberry Pi 4

Das Argon One Case ist eines der ergonomischsten und ästhetischsten Gehäuse aus Aluminiumlegierung für den Raspberry Pi.

~~117,60 €~~

shop.heise.de/argon-set **99,90 € ›**

„No Signal“ Smartphone-Hülle

Passend für Smartphones aller Größen bis 23cm Länge blockt diese zusammenrollbare Hülle alle Signale von GPS, WLAN, 3G, LTE, 5G und Bluetooth, sowie jegliche Handy-Strahlung. Versilbertes Gewebe im Inneren der Tasche aus recycelter Fallschirmseide bildet nach dem Schließen einen faradayschen Käfig und blockiert so alles Signale.

shop.heise.de/no-signal-sleeve **29,90 € ›**

Stockschirm protec'ted

Innen ist Außen und umgekehrt.
Dieser etwas andere Regenschirm sorgt für interessierte Blicke auch bei grauem und nassem Wetter. Als Highlight kommt noch das stilvolle und dezente Design in Schwarz und Blau mit der mehr als passenden Aufschrift “Always protec'ted“ daher.

shop.heise.de/ct-schirm **22,90 € ›**

› Bestellen Sie ganz einfach online unter **shop.heise.de** oder per E-Mail: **service@shop.heise.de**

heise shop

shop.heise.de ›

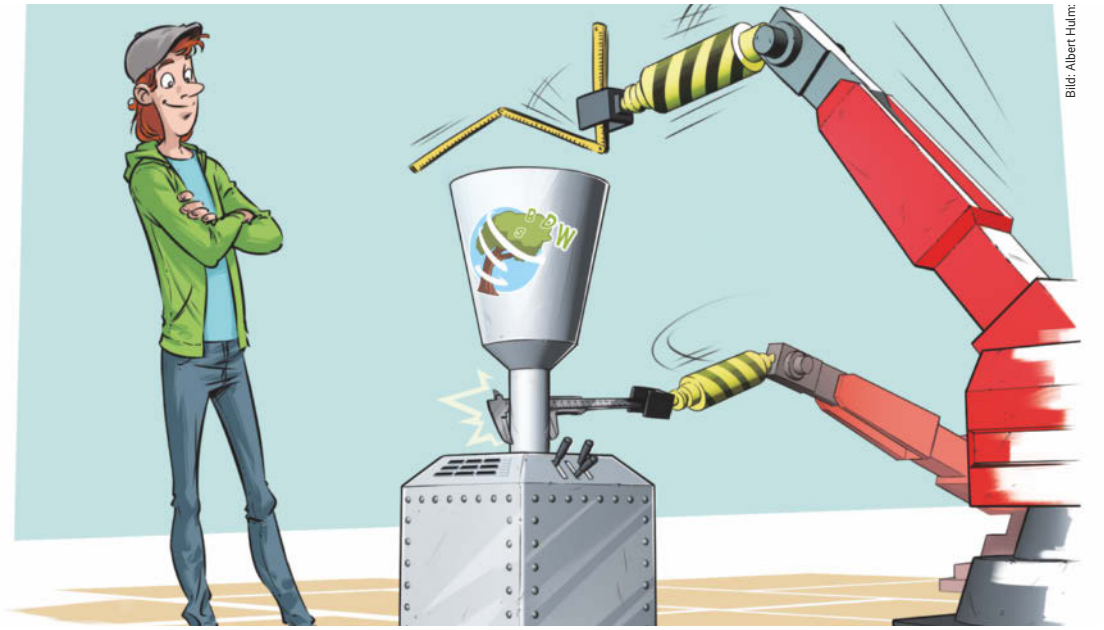


Bild: Albert Hulm

Automatisches Testen mit unittest

Warum sollte man Software per Hand testen, wenn Unit-Tests die Arbeit automatisch erledigen? Der Zusatzaufwand zum Programmieren der Tests lohnt sich schon nach wenigen Releases und die Tests helfen, Fehler zu vermeiden und sauber zu strukturieren.

Von Pina Merkert

Programmieren wäre einfach, wenn der Code nicht funktionieren müsste. Stattdessen versuchen Programmierer, so gut es geht, ihren Code zu verstehen, damit der in allen Fällen ohne Fehler genau das tut, was die Auftraggeber wünschen. Die Prüfung nach der Auslieferung frisst eine Menge Zeit. Meist wollen die Prüfer auch noch Zeit sparen und übersehen dann Fehler.

Die Infamie manueller Tests offenbaren die weiteren Releases: Falls der Test zeigt, dass der Programmierer etwas ändern muss, wiederholt sich der gesamte Zyklus. Am Ende der zweiten Iteration testen die Prüfer dann dieselben Features noch mal, was mit zunehmender Zahl an Iterationen zu einer ziemlich stupiden Arbeit verkommt. Und immer, wenn ein Mensch am Rechner stupide Arbeiten verrichtet, keimt die Idee, ob man diese denn nicht automatisieren könnte.

Genau dazu dienen automatische Tests. Das sind kleine Programme, die anderen Programmen

auf die Finger schauen und prüfen, ob deren Ausgaben den Erwartungen entsprechen. Wir haben diverse Tests für unseren Flask-Microservice (siehe Seite 172) geschrieben und nutzen den dort vorgestellten Markdown-Konverter als Beispiel, wie automatische Tests einer Python-Anwendung aussehen. Den Code finden Sie im Git-Repository über ct.de/wvau.

Randfällig

Jeder automatische Test prüft nur einen Aspekt einer Funktion. Dadurch muss man beim Programmieren der Tests nicht lange nachdenken: Zu je einem Beispiel gehört eine bestimmte erwartete Ausgabe. Um eine Funktion jedoch vollständig zu testen, sind oft Dutzende von Beispielen nötig. Schließlich sollen die Tests jeder Verzweigung folgen und jeden Randfall abdecken.

Dass die Tests dabei jeweils ähnlich aussehen und Code duplizieren, muss den Tester nicht kümmern. Jeder Test soll für sich einfach zu schreiben und leicht zu verstehen sein. Performance und Ästhetik spielen eine untergeordnete Rolle.

Viel wichtiger ist dagegen, dass die Tests alle Fehler finden, die sich im Code verstecken könnten. Für eine Funktion, die eine Liste mit Zahlen überprüft, ob sie größer als 5 sind, bedeutet das beispielsweise folgende Tests:

- Die Funktion wirft einen Fehler, wenn man keine Liste übergibt.
- Die Funktion wirft einen Fehler, wenn die Liste nicht nur Zahlen enthält.
- Die Funktion gibt `False` zurück, wenn man eine Liste mit Werten übergibt, bei der einer der Werte exakt 5 ist.
- Die Funktion gibt `True` zurück, wenn man eine Liste mit genau einer Zahl übergibt, die minimal größer als 5 ist, beispielsweise 5,001.

Je nach Programmiersprache ist es auch sinnvoll, negative Zahlen und verschieden lange Datentypen zu prüfen. Im Zweifel sollte man lieber einen Test mehr als zu wenig schreiben.

Häppchenweise

Da jemand die Tests programmieren muss, fällt beim testgetriebenen Entwickeln zwangsläufig mehr Arbeit für den Programmierer an. Damit die Tests nicht zu viel der kostbaren Zeit fressen, versucht man deswegen, die Tests so klein und einfach wie möglich zu schreiben. Testet ein automatischer Test nur eine einzelne Funktion (im Tester-Jargon eine „Unit“), handelt es sich um einen Unit-Test.

Damit die Tests einzelne Funktionen aufrufen können, liegen sie im gleichen Repository wie der produktive Code, meist in einem Unterordner `tests/`. Jeder Test prüft nur ein einzelnes Feature mit einem bestimmten Satz an Eingaben. Um dafür möglichst wenig Code zu brauchen, bringt Python das Modul `unittest` mit. Für andere Programmiersprachen gibt es stets mindestens ein Test-Framework (oft sogar mehrere), das nach der gleichen Idee funktioniert wie das hier vorgestellte `unittest` für Python.

Ein Unit-Test sieht beispielsweise so aus:

```
import unittest
class TestPandocStringConverter(
                    unittest.TestCase):
  def test_single_string(self):
    span_list = convert_list([
        {'t': 'Str', 'c': 'Foo.'}
    ], [])
    self.assertEqual([{
        "type": "span-regular",
        "text": "Foo."
    }], span_list)
```

Nach dem Import von `unittest` definiert der Test eine Klasse, die von `unittest.TestCase` erbt. Alle in solch einer Klasse definierten Tests führt das Framework aus, wenn der Code in der gleichen Datei `unittest.main()` aufruft:

```
if __name__ == '__main__':
  unittest.main()
```

In der Klasse kann man nach Herzenslust Methoden definieren, die im `TestCase`-Objekt Eigenschaften setzen. Automatisch ausgeführt werden solche Funktionen aber nicht. Erst wenn der Name einer Methode mit `test` beginnt, führt `unittest` sie automatisch als eines der Test-Programme aus.

Der `TestCase` bringt zusätzlich eine ganze Reihe von Prüf-Funktionen mit, die mit `assert` anfangen und ganz ähnlich wie Pythons Schlüsselwort `assert` funktionieren. Das Verb „to assert“ bedeutet bei automatischen Tests „zusichern“. Eine Assertion sichert zu, dass an dieser Stelle im Programm die angegebene Bedingung erfüllt ist. Bleibt eine Zusicherung unerfüllt, wirft der Python-Interpreter einen Fehler, der das gebrochene Versprechen benennt.

Im Beispiel prüft `self.assertEqual()`, ob in der Variable `span_list` (zweiter Parameter) die im ersten

Parameter erwartete Datenstruktur steckt (eine Liste mit einem Dictionary mit zwei Schlüsseln).

Statt wie hier die gesamte Struktur auf einmal zu prüfen, könnte die Funktion auch mit mehreren Assertions einzelne Werte prüfen:

```
self.assertIsInstance(tree, list)
self.assertEqual(1, len(tree))
self.assertIsInstance(tree[0], dict)
self.assertIn("type", tree[0])
self.assertEqual("span-regular",
                 tree[0]["type"])
self.assertIn("text", tree[0])
self.assertEqual("Foo.",
                 tree[0]["text"])
self.assertEqual(2,
                 len(tree[0].keys()))
```

Neben diesen Assertions gibt es auch Gegenteile wie `self.assertNotIn()`, für Gleitkommazahlen `self.assertAlmostEqual()`, wo man die Zahl der Nachkommastellen angibt, auf die zwei Werte übereinstimmen müssen, und für Fehler `self.assertRaises()`. Bei Letzterer übergibt man die zu testende Funktion der Assertion, statt sie vorher selbst aufzurufen, damit die Assertion die `Exception` abfangen kann.

Testgetrieben

Interessant wird es, wenn Entwickler zuerst die Tests schreiben und danach erst den Code, den diese prüfen. Dieses „Test Driven Development“ hilft dabei, den Code zu strukturieren. Es ist nämlich relativ leicht, für die Tests zunächst ein paar Beispiele zu erfinden, die der Code später erfüllen soll. Beim Schreiben des Codes steht man dann nämlich nicht mehr unter Druck, jeden Randfall im Kopf zu haben. Stattdessen folgt man einfach der intuitivsten Idee für die Implementierung. Mit den Tests prüft man in Millisekunden, ob die erste Idee schon alle Anforderungen erfüllt. Tut sie das nicht, geben die fehlschlagenden Tests Auskunft darüber, wo man noch nachbessern muss.

Testgetriebenes Entwickeln schafft nicht nur Gemütsruhe, es erlaubt auch, entgegen des üblichen Vorgehens (vom Groben ins Feine), vom Feinen ins

Die reine Lehre der Unit-Tests

Ein Unit-Test prüft im Idealfall einen einzelnen Aspekt einer einzelnen Funktion. Ruft diese Funktion intern weitere Funktionen auf, sollte deren korrektes Funktionieren eigentlich nicht Gegenstand dieses Tests sein. Für sie sollte es nämlich eigene Unit-Tests geben.

Um das zu erreichen, verwendet man sogenannte „Mocks“. Das Wort entstammt dem Englischen „to mock someting“ – „etwas vortäuschen“. Ein Mock sieht für die zu testende Funktion wie die aufgerufene Funktion aus (gleiche Signatur, gleicher Typ der Rückgabe), gibt aber lediglich statisch den für den Test nötigen Wert zurück, ohne tatsächlich etwas zu berechnen.

Damit der Test die echte und die vorgetäuschte Funktion gegeneinander austauschen kann, kommen Techniken wie Dependency Injection zum Einsatz. Das erfordert wiederum eine bestimmte Struktur des ursprünglichen Codes und nicht nur zusätzlichen Code für die Mocks auf der Seite der Tests.

Ob man Tests als Unit-Tests nach reiner Lehre schreibt (sie laufen dann besonders schnell, da sie jeweils nur ein Minimum an Code ausführen) oder den Aufwand scheut und die Funktionen mehrfach aufruft statt sie durch Mocks zu ersetzen, bleibt jedem selbst überlassen. Das Beispiel im Artikel ruft die Funktionen einfach auf, weil die Tests dadurch einfacher sind und die Laufzeit trotzdem sehr kurz bleibt. Dadurch sind aber viele der Tests genau genommen keine Unit-Tests, weil sie mehr testen als nur eine Unit.

Grobe zu arbeiten. Bei diesem Workflow entstehen einzelne Funktionen ohne einen Zusammenhang. Die Tests stellen ihre korrekte Funktion sicher, bevor andere Programmteile sie nutzen. Statt ein Feature in seiner ganzen Komplexität in einem Schritt erdenken und umsetzen zu müssen, stürzt man sich so zuerst auf überschaubare Probleme. Einzelne Funktionen vorab auszuentwickeln, lohnt sich jedoch immer nur dann, wenn man abschätzen kann, dass das Programm eine Funktion später auch braucht.

Integrationstests

Prüft ein Test nicht nur eine Unit, sondern den ganzen Programmfluss von der Benutzereingabe bis zur sichtbaren Ausgabe, spricht man von „Integrationstests“. Ungeachtet des Namens unterstützt das Python-Modul `unittest` auch diese Art des automatischen Tests. In der Praxis müssen aber meist andere Frameworks mithelfen, damit die Tests ein realistisches Szenario für den Aufruf des Programms simulieren können.

Das Beispiel nutzt `app.test_client()` aus dem Flask-Framework, um einen HTTP-Request zu simulieren. Der Aufruf der Methode `.post()` bei diesem Client verhält sich genau wie ein echter HTTP-POST-Request, ohne dafür jedoch Port 80 zu belegen und über diesen TCP-Pakete zu verschicken. Der Mock-Request ist wesentlich schneller und belegt weniger Ressourcen des Systems. Das zurückgelieferte Response-Objekt entspricht trotzdem der echten Antwort:

```
markdown = "Paragraph 1.\n\nPara 2..."
with app.test_client() as test_client:
  response = test_client.post('/',
                              data=markdown)
  tree = response.get_json()
self.assertEqual({"type": "block" #...
                  }, tree)
```

Damit Flask dabei auch nichts Überflüssiges tut, versetzt der Code das Web-Framework noch mit `app.testing=True` in den Testmodus. Da das für jeden im `TestCase` definierten Test sinnvoll ist, lagert man das in die `setUp()`-Methode aus, die `unittest` automatisch vor jedem Test ausführt:

```
def setUp(self) -> None:
  app.testing = True
```

Was die Klasse in `setUp()` aufbaut, kann sie später in `tearDown()` wieder einreißen. Mit diesen beiden Methoden sorgen die Tests bei jedem Lauf für gleiche Bedingungen (Datenbank zurücksetzen etc.).

Test-Runner

Mit dem Aufruf von `unittest.main()` führt `unittest` nur die Tests aus, die die `TestCase`-Klassen in dieser Datei definieren. Hat man die Tests für eine bessere Übersicht aber auf mehrere Dateien verteilt, müsste man jede einzeln aufrufen. Statt selbst ein Skript dafür zu schreiben, installiert man lieber einen Test-Runner:

```
pip install nose2
```

Nose2 schnüffelt selbsttätig nach Dateien mit `unittest`-Tests. Es durchsucht dafür rekursiv sämtliche Python-Module (Ordner, die eine Datei namens `__init__.py` enthalten), die Ordner `src` und `lib` und alle anderen Ordner, die `test` im Namen haben (ob in Groß- oder Kleinschreibung ist Nose2 dabei egal). In den Ordnern sucht es nach Dateien, die mit `test` anfangen. Die Datei `missglueckter_test.py` lässt Nose2 also links liegen. In den Dateien ruft Nose2 alle von `unittest.TestCase` erbenden Klassen auf und zusätzlich Funktionen, deren Name mit `test` beginnt.

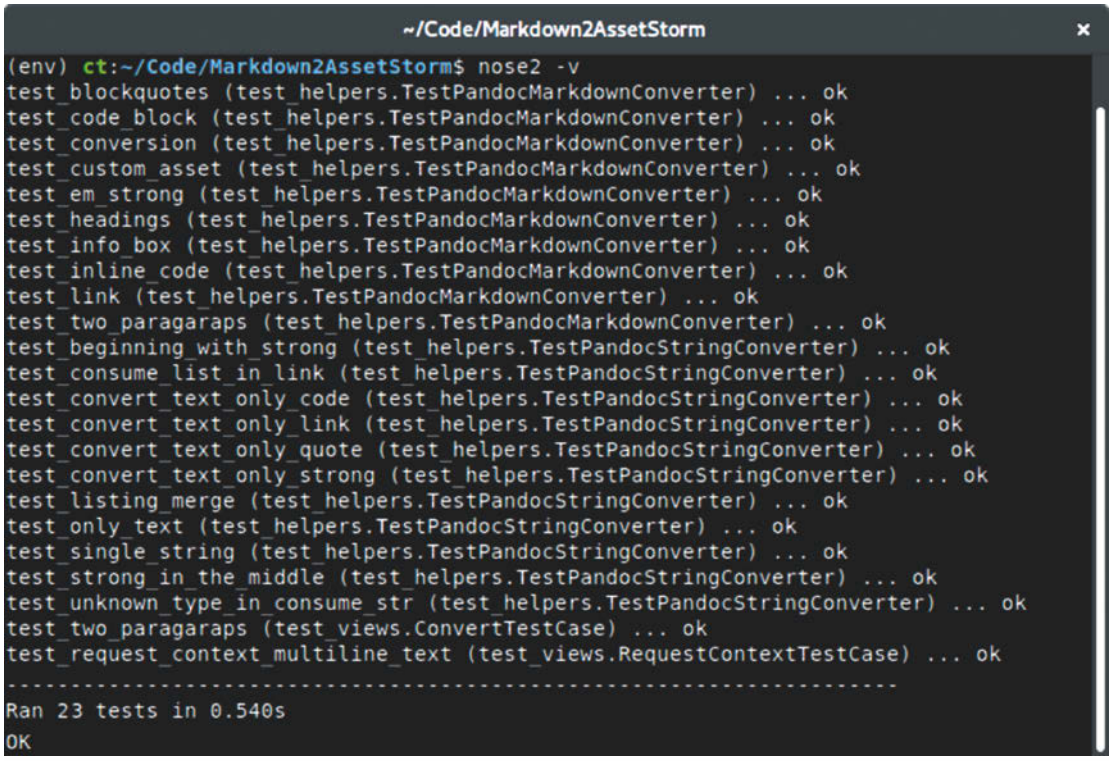
Den Schnüffler setzt man mit dem Aufruf `nose2` im Projektverzeichnis auf die Fährte (die Option `--start-dir` setzt ein anderes Verzeichnis). Nose2 protokolliert dann jeden erfolgreichen Test mit einem Punkt und jeden Fehlschlag mit einem `F`. Diese Ansicht verschleiert, welcher Test genau fehlschlug. Mit der Option `-v` erteilt Nose2 brav Auskunft:

```
nose2 -v
```

Nose2 führt auch nur einzelne Tests aus, wenn man diese als Parameter angibt:

```
nose2 -v tests.test_helpers.␣
        ↳TestPandocMarkdownConverter.␣
        ↳test_headings
```

Die Notation entspricht der von `import`: `tests` bezeichnet das Verzeichnis mit diesem Namen. `test_helpers` die Datei `test_helpers.py` in diesem Verzeichnis. `TestPandocMarkdownConverter` heißt die dort definierte Klasse, die eine Methode `test_headings()` enthält. Nose2 führt mit obigem Aufruf nur diesen einzelnen Test aus – perfekt fürs testgetriebene Entwickeln. Lässt man die Methode weg, führt der Test-Runner alle Tests dieser Klasse aus, entfernt man auch die Klasse, sämtliche Tests in der Datei und ohne Datei sämtliche Tests im Ordner.



Nose2 durchsucht die Ordner, in denen Entwickler üblicherweise Tests ablegen, und führt alle Tests in Test-Case-Objekten aus.

Coverage

Nose2 zusammen mit `unittest` führt alle existierenden Tests in einem Rutsch aus. Doch testen die Tests auch den kompletten Code? Der Frage geht Coverage.py nach:

```
pip install coverage
```

Das Tool protokolliert, welche Zeilen des Quellcodes ein Programm durchläuft. Die Tests sollten im Idealfall alle Zeilen des Codes durchlaufen. Coverage.py ruft also Nose2 auf, das alle Tests aufruft:

```
coverage run -m nose2 -v
```

Riefe `coverage` direkt ein Python-Programm auf, könnte die Option `-m` wegfallen. `nose2` ist aber keine Datei im Projektordner, sondern ein Python-Modul. Egal ob Modul oder Datei, hinter dem Namen dürfen die üblichen Parameter folgen. Hier gehört die Option `-v` also zum Aufruf von `nose2` und nicht zu `coverage`.

Der Befehl sammelt die Informationen, welche Zeilen durchlaufen wurden, still und heimlich in der versteckten Datei `.coverage`. An der Ausgabe erkennt man also zunächst nicht, dass der Aufruf mit Coverage-Analyse stattgefunden hat.

Eine Zusammenfassung, wie viel Code durchlaufen wurde, zeigt erst `coverage report`. Da diese Zusammenfassung aber auch die Abdeckung der Bibliotheken in der virtuellen Umgebung unter `env/` und der Tests selbst enthält, ist es sinnvoll, diese Ordner für eine bessere Übersicht aus der Ausgabe auszuschließen:

```
$ coverage report --omit=env/*,src/tests/*
Name           Stmts   Miss  Cover
--------------------------------------
converter.py      12      1    92%
helpers.py       139     16    88%
--------------------------------------
TOTAL            151     17    89%
```

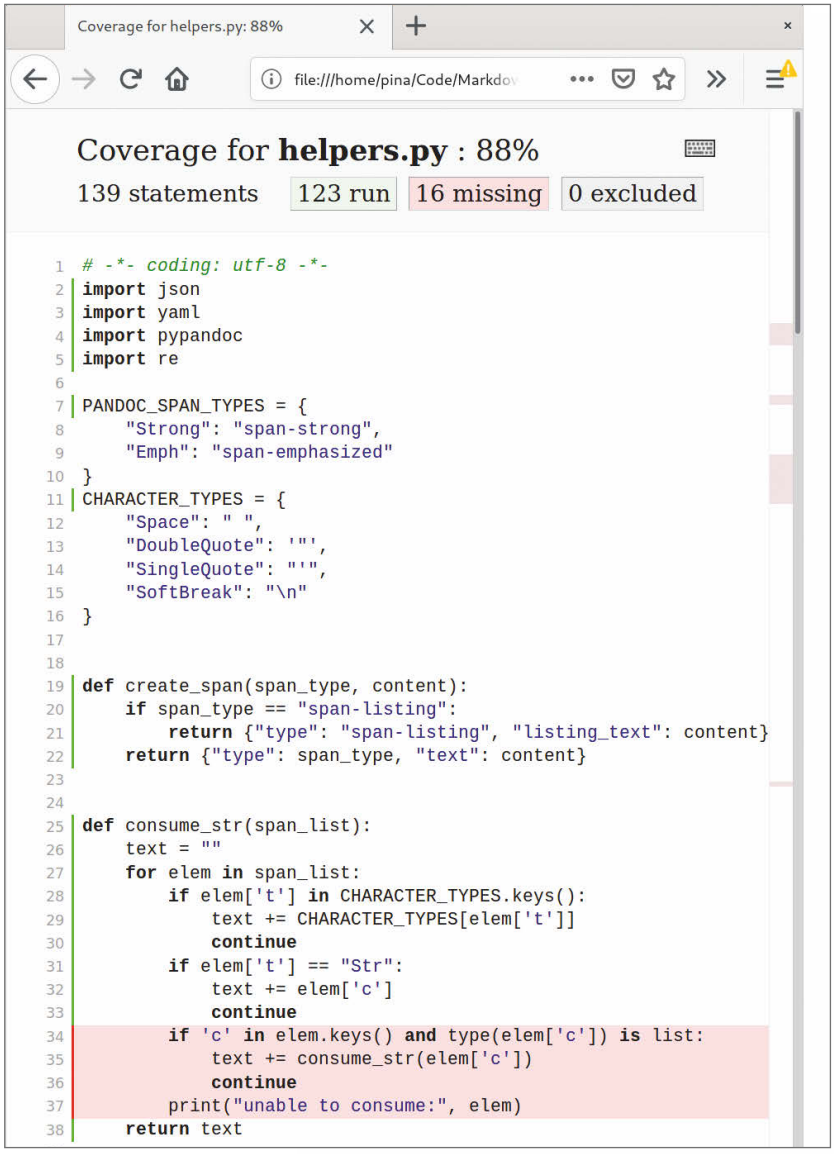
Der Bericht belegt, dass das Beispielprojekt hier nur 123 von 139 Zeilen mit Python-Befehlen in der Datei `helpers.py` testet. Fehler in den ungetesteten Zeilen können die Tests unmöglich finden – wir müssen hier also noch mal ran.

Literatur

[1] Merlin Schumacher, Und Actions!, Erste Schritte mit GitHubs CI/CD-Werkzeug Actions, c't 25/2019, S. 164

Code bei GitHub:

www.ct.de/wvau



Die Tests durchlaufen bei dieser Datei nicht alle Zeilen des Quellcodes. In den von Coverage.py rot markierten Zeilen verstecken sich leicht Fehler.

Dafür wünscht man sich eine gut lesbare Übersicht der getesteten Zeilen. Genau die erzeugt die HTML-Ausgabe von Coverage.py:

```
coverage html --omit=env/*,src/tests/*
```

Der Befehl erzeugt den Ordner `htmlcov/` mit einer `index.html` für die Anzeige mit einem Browser. Dort navigiert man zur fraglichen Datei und bekommt den Quelltext mit grüner oder roter Markierung, je nachdem, ob die Tests eine Zeile durchlaufen haben oder nicht.

Eine Testabdeckung von 100 Prozent bei allen Code-Dateien bedeutet nicht, dass die Tests jeden Fehler finden würden. Gerade ungetestete Randfälle machen häufig Probleme, obwohl ein Test die Zeile berührt. Ungetestete Zeilen bedeuten aber im Umkehrschluss, dass sich dort Bugs verstecken könnten, die die Tests sicher nicht finden. Eine vollständige Testabdeckung sollte also grundsätzlich ein Ziel sein, wenn auch nicht das einzige.

CI-Alltag

Hat man ein Projekt mit Tests ausgestattet und eine gute Coverage erreicht, macht das Entwickeln neuer Features richtig Spaß: Alle Funktionen haben bereits für sich unter Beweis gestellt, dass sie funktionieren. Ob die Erweiterung bestehende Funktionen stört, zeigen die Tests in Millisekunden. Und die Handvoll zusätzlicher Tests für das neue Feature kosten auch nur wenige Minuten Programmierarbeit. Refactoring ist sogar noch schöner, weil man da gar keine neuen Tests schreiben muss und sofort weiß, ob die Änderung korrekt funktioniert.

Viele Firmen nutzen Continuus-Integration-Server (CI), um Tests automatisch bei jedem Commit im Repository aufzurufen (beispielsweise mit GitHub-Actions [1]). Schlagen die Tests fehl, oder sinkt die Coverage unter einen eingestellten Wert, weist das Repository den Commit automatisch zurück und erstellt dem Entwickler ein Issue, um den Missstand zu beheben. Solche Automatiken helfen enorm, eine exzellente Codequalität aufrechtzuhalten.

Im ersten Moment sieht das Programmieren von Tests meist wie zusätzliche Arbeit aus. Je länger ein Softwareprojekt genutzt und entwickelt wird, desto mehr zahlt sich diese aber aus: Die Tester sparen bei jedem Release Zeit, Refactoring geht leichter von der Hand und die Arbeit mit getesteten Units sorgt für Gemütsruhe beim Entwickeln neuer Features. Außerdem weisen Projekte mit automatischen Tests im Schnitt weniger Fehler auf als per Hand getestete Software. Schleicht sich dennoch ein Bug ins Release ein, schreibt man als allererstes einen Test, der den Fehler reproduziert. Das stellt nämlich sicher, dass man niemals zweimal den gleichen Fehler in die Software einbaut. (pmk) c't

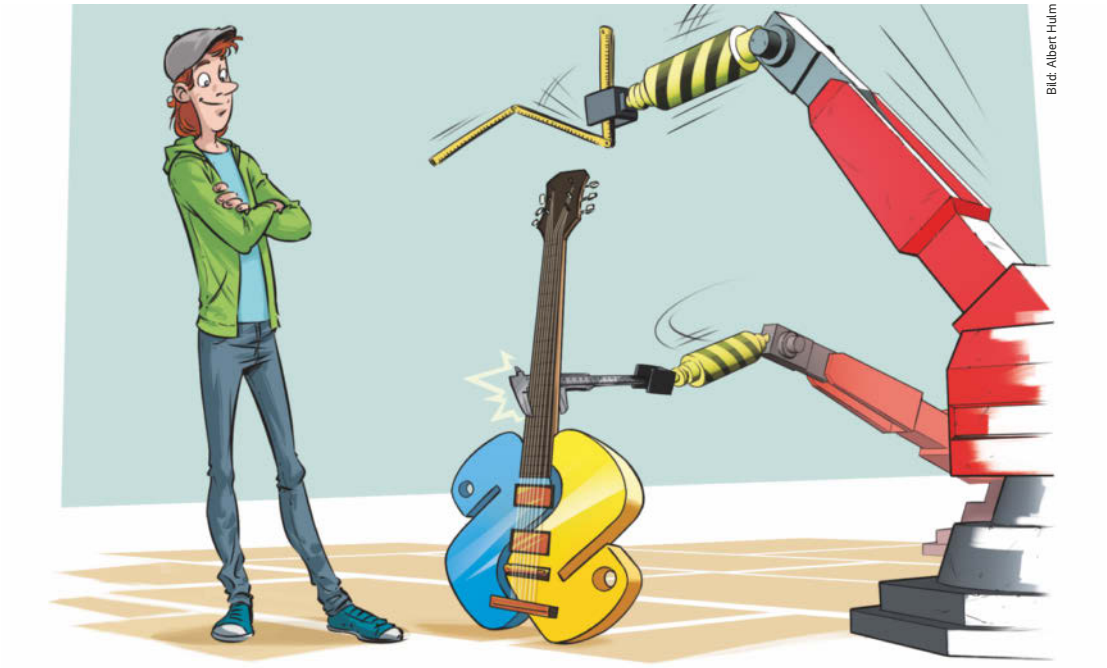


Bild: Albert Hulm

Django testen

Automatische Tests sollten immer unter den gleichen Voraussetzungen laufen. Datenbanken dienen aber dazu, zu speichern ohne zu vergessen – jede SQL-Anweisung im Test einer Datenbankanwendung verändert also den Status für andere Tests. Irgendwie müssen diese widerstrebenden Prinzipien zusammenfinden. Django löst das scheinbare Dilemma mit begnadeter Eleganz.

Von Pina Merkert

Ein neuer TestCase prüfte, ob die Anwendung Daten korrekt in der Datenbank gespeichert hatte. Nach ein paar Durchläufen passte die Testdatenbank nicht mehr auf die SSD, weil keiner der Tests die Daten wieder löschte. Eine Lösung bestand aus einem Winkelzug in Form eines Pseudo-Tests, der alle Datensätze eines Typs entfernte. Kurz darauf fluchte ein Kollege, dass seine Testdaten verschwunden seien.

Diese Szene spielte sich vor Jahren mal bei einem Projekt ab, bei dem alle Entwickler mit einer einzigen Test-Datenbank arbeiteten. Sie veranschaulicht, warum eine Datenbank für alle, besonders zusammen mit automatischen Tests, eine dumme Idee ist.

Dass automatische Tests von Python-Anwendungen jedoch die Qualität des Codes verbessern, strukturierteres Programmieren erlauben und dabei gar nicht so viel zusätzliche Arbeit verursachen, erklärt unser Einstieg in Pythons `unittest` (siehe Seite 120). Das Beispiel war aber recht überschaubar und klammerte einige Stolperfallen wie eine Datenbank beim Schreiben automatischer Tests aus. Denn vor jedem Test sollte die Datenbank

stets auf dem gleichen Stand sein. Diese Anforderung ans Testen von Datenbankanwendungen erfüllt Django elegant und mit wenigen Zeilen Code. Weil das so einfach ist, sollte kein Django-Projekt ungetestet bleiben.

Django bringt für seine eleganten Tests eigene Funktionen mit, die das Standardmodul `unittest` erweitern und daher genauso funktionieren. Auch einen Test-Runner, der sich fast wie Nose2 bedienen lässt, packt Django automatisch in das Projekt-Verwaltungsskript `manage.py`.

django.test.TestCase

Djangos `TestCase` erbt von der gleichnamigen Klasse aus dem `unittest`-Modul. Sie erweitert aber unsichtbar die `setUp()`- und `tearDown()`-Methoden (genau genommen `setUpClass()` und `tearDownClass()`, was aber aufs Gleiche hinausläuft) um die Logik, die Tests einer Datenbankanwendung benötigen.

Vor dem Ausführen des ersten Tests legt Django eine Test-Datenbank an, um die Datenbank des Systems nicht zu stören. Den Namen der Test-Datenbank erzeugt Django, indem es dem Namen der Datenbank in `settings.py` ein `test_` voranstellt. Da Django diese Datenbanken stets neu erzeugt, laufen Django-Tests nur, wenn die Anwendung auf dem Datenbankserver auch das Recht zum Anlegen neuer Datenbanken besitzt (`CREATE ROLE djangouser WITH CREATEDB` vergisst man leicht beim Anlegen des Datenbank-Benutzers).

Vor jedem Test führt Django ein Rollback der Test-Datenbank aus, sodass jeder Test mit einer vergleichbaren jungfräulichen Datenbank startet. Unterstützt das Datenbank-Backend kein Rollback, löscht Django alle Inhalte. Die leere Datenbank füllt Django danach mit „Fixtures“. Das sind Inhalte für die Datenbank, die man bei Django-Projekten üblicherweise als YAML-Dateien im `fixtures/`-Ordner der zu testenden App ablegt. Django-Apps sind Unterordner mit eigenen Datenbankmodellen, Views und URL-Konfigurationen. Welche Fixtures ein Test nutzt, gibt man über eine Klassenvariable im `TestCase` namens `fixtures` an. Die Liste enthält volle Dateinamen ohne Ordnerangabe. Neben YAML-Dateien erlaubt Django auch Fixtures im JSON-Format:

```
from django.test import TestCase
class TestLoadAsset(TestCase):
  fixtures = [
    'span_assets.yaml',
    'block_assets.yaml']
```

Fixtures kann man per Hand schreiben, muss man aber nicht. Der Befehl `./manage.py dumpdata` konvertiert den aktuellen Inhalt der Datenbank in YAML. Hinter der Option `-o` gibt man den Namen einer Datei an, in die der Befehl die Daten schreibt.

Mit YAML-befüllter Test-Datenbank führt der `TestCase` die Tests aus. Das funktioniert genau wie bei `unittest`; Django ergänzt lediglich einige praktische Assertions wie `assertURLEqual()` oder `assertQuerysetEqual()`. Die Liste aller Assertions nennt die Dokumentation (ct.de/wwug).

Zuletzt räumt Django automatisch hinter jedem Test auf, damit der nächste Test wieder mit gleichen Bedingungen und definierter Umgebung starten kann. Nach dem letzten Test löscht Django die Test-Datenbank, um nicht unnötig Platz auf dem Datenbankserver zu belegen.

./manage.py test

Der Aufruf von `./manage.py test` erspart einen Test-Runner wie Nose2. Django geht beim Ausführen von Tests aber ähnlich wie Nose2 vor. Die Option `-v` erwartet nun aber eine zusätzliche Geschwätzigkeitsangabe in Form einer Zahl zwischen 0 und 3. Der Parameter 1 entspricht der Ausgabe von Nose2 ohne `-v` und stellt Djangos Standardwert dar, falls man die Option weglässt. Ab `-v 2` listet Django auch die Namen aller Tests auf, ähnlich wie Nose2 das mit `-v` niederschreibt. Djangos Ausgabe gibt vor der Liste zusätzlich noch an, wie es die Datenbank vorbereitet und welche Migrationen es ausführt.

Gibt man weitere Parameter an, spezifizieren diese einzelne Tests, TestCases, Dateien oder Apps, die Djangos Test-Runner dann ausführt:

```
./manage.py test -v 2 assets.tests.↵
      ↳test_views.TestSaveAsset.↵
      ↳test_no_type assets.tests.↵
      ↳test_models.AssetBasicTestCase
```

Mit der Option `--parallel` klont Django die Test-Datenbank einmal für jeden logischen Kern des Prozessors und führt die Tests anschließend parallel aus. Das beschleunigt die Tests selbst zwar enorm, die Klone der Datenbank verursachen aber einen erheblichen Overhead. Die Tests für das Beispielprojekt AssetStorm (siehe ct.de/wwug) brauchen deswegen auf einem i5 mit vier Kernen und Hyperthreading 11,3 statt 3,6 Sekunden. `--parallel` lohnt sich also erst, wenn das Projekt viele Tests mit langer Laufzeit enthält. Laufen die Tests auf Ihrem

System nicht an, fehlt vermutlich die PostgreSQL-Datenbank. Wir haben einen Docker-Container vorbereitet:

```
docker-compose up
```

Coverage.py

Coverage.py aus dem Python Package Index (PyPI) prüft wie üblich die Testabdeckung:

```
pip install coverage
```

Coverage.py sammelt zunächst beim Ausführen eines Python-Programms die Informationen, welche Zeilen der Code durchlaufen hat. Zum Protokollieren der Testabdeckung muss man nur `manage.py test` per `coverage run` ausführen:

```
coverage run manage.py test -v 2
```

`coverage report` listet danach die Testabdeckung aller Zeilen in allen Dateien auf – viel zu geschwätzig. Aus der Liste möchte man die Dateien der Tests, `manage.py`, Django und alle anderen Bibliotheken der virtuellen Umgebung ausschließen:

```
coverage report --omit=manage.py,↴
              ↳assets/tests/*,↴
              ↳../env/*,*__init__.py
```

Mit den gleichen Parametern schreibt `coverage html` eine im Browser navigierbare Übersicht in den Ordner `htmlcov`. Dort steht, welche Zeilen der Test-Code ausführt oder nicht.

self.client=Client()

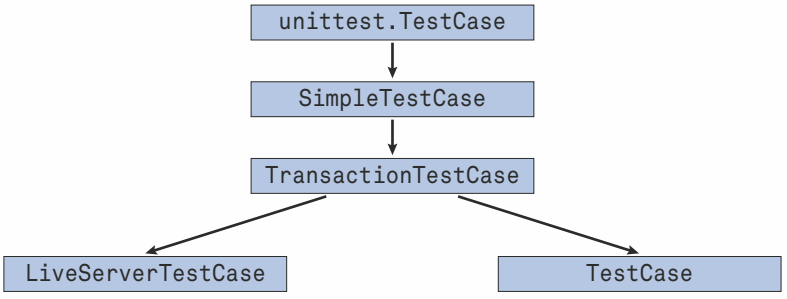
Um die View-Funktionen zu testen, mit denen Django auf Requests antwortet, muss das Framework keinen kompletten Webserver starten. Stattdessen liefert Django einen Test-Client für simulierte Requests mit (`django.test.Client`). Mit diesem gelingen Integrationstests für die Serverseite von Django-Anwendungen, ohne mit einem tatsächlichen Webserver Port 80 belegen zu müssen.

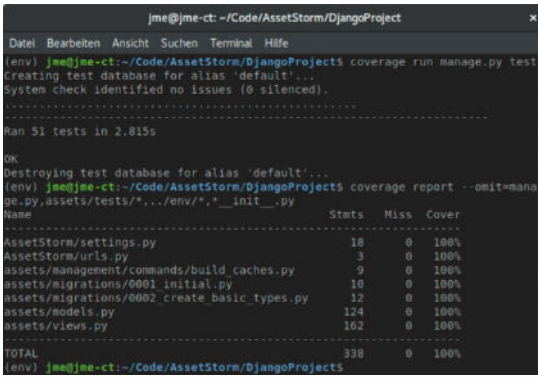
Die Aufrufe mit so einem Client geben Pfade an, die zu Djangos Regex-basierter URL-Konfiguration in `urls.py` passen müssen. Wer die Pfade dort gern mal umstellt, kann in den Tests die `reverse()`-Funktion aus dem Modul `django.urls` verwenden. Sie nimmt die View-Funktion und URL-Parameter an und erzeugt daraus den Pfad, der zur zugehörigen Definition in `urls.py` passt. Ein Test damit sieht dann so aus:

```
from django.test import TestCase
from django.test import Client
from django.urls import reverse
class TestLoadAsset(TestCase):
  fixtures = []
  def setUp(self) -> None:
    self.client = Client()
  def test_no_params(self):
    rsp = self.client.get(
                reverse('load_asset'))
    self.assertEqual(rsp.status_code,
                     400)
```

Djangos TestCase-Klassen

Django erweitert die `TestCase`-Klasse aus dem `unittest`-Modul mehrfach über Vererbung: `SimpleTestCase` enthält Erweiterungen für Django-Projekte ohne Datenbank. `TransactionTestCase` kümmert sich um Rollbacks der Datenbank und Fixtures. `TestCase` ergänzt Datenmigrationen. `LiveServerTestCase` startet einen Webserver für Selenium und Co.





Coverage.py beweist: Die Tests durchlaufen den gesamten Code des Beispielprojekts. Dafür sind 51 Tests nötig.

```
        self.assertJSONEqual(rsp.content,
            {'Error': "Please supply an " +
                      "'id' as a GET param."})
```

Bei den Assertions prüft der Test zunächst, ob die Antwort des Servers wie erwartet einen Fehlercode liefert. Die Fehlermeldung kodiert der Server als JSON. Die Assertion `self.assertJSONEqual()` erspart den JSON-Parser per Hand aufzurufen, weil die Funktion Strings frisst.

Tests ohne Client

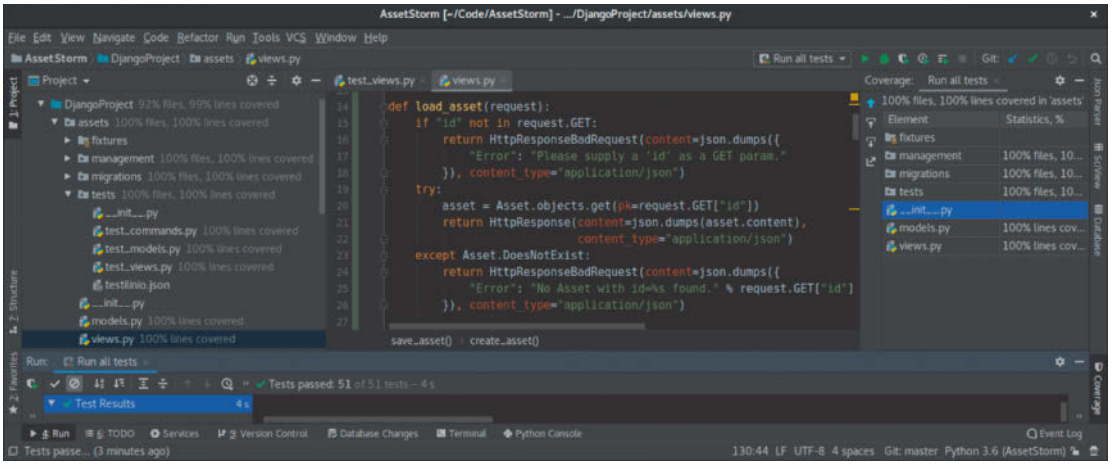
Statt mit dem `django.test.Client` einen HTTP-Request zu simulieren, kann man die View-Funktionen auch direkt in einem Unit-Test aufrufen. Da View-Funktionen ein `Request`-Objekt erwarten, bringt Django eine `RequestFactory` mit, um solche Objekte zu produzieren:

```
from django.test import RequestFactory
from django.urls import reverse
from assets.views import load_asset
class TestLoadAssetUnittest(TestCase):
  def test_no_params(self):
    rf = RequestFactory()
    path = reverse('load_asset')
    req = rf.get(path)
    req.user = AnonymousUser()
    resp = load_asset(req)
    self.assertEqual(resp.status_code,
                     400)
```

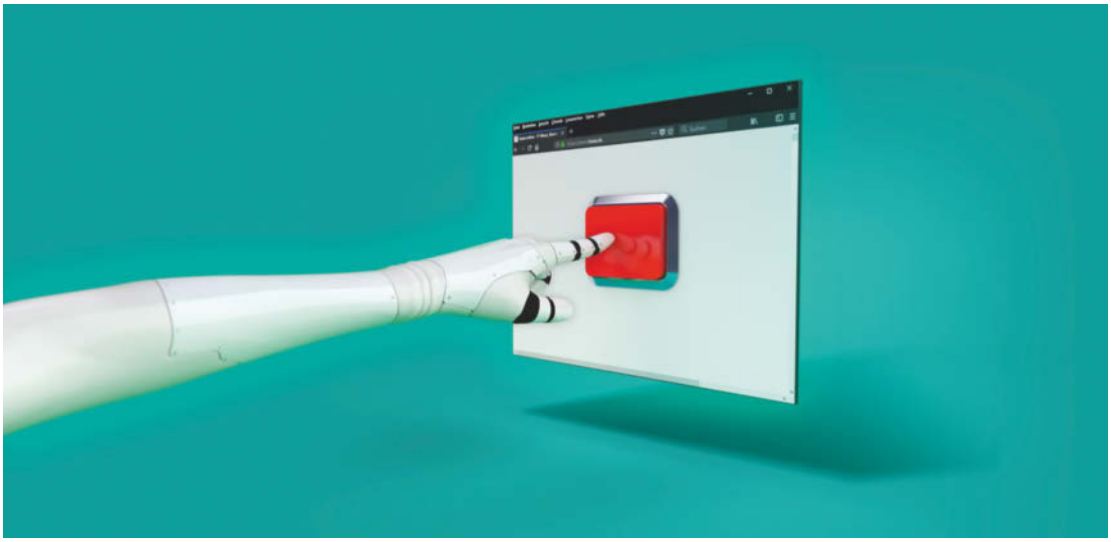
Dieses Vorgehen bietet sich für Anhänger einer reinen Lehre von Unit-Tests an, bei der ein Test nur die Funktion und nichts drum herum testet.

Keine Ausreden!

Django erweitert `unittest.TestCase` so geschickt, dass keine zusätzliche Zeile Code nötig wird, um eine komplexe Datenbankanwendung automatisch zu testen. Dank Rollback setzen sich die genutzten Datenbanken vor jedem Test so schnell zurück, dass dutzende Tests in wenigen Sekunden durchlaufen. Fixtures befüllen die Test-Datenbank in einem Wimpernschlag mit Daten, sodass sie bei jedem Durchlauf auf einem beliebigen, aber definierten Stand ist. Zusammen mit Djangos ergänzenden Assertions, dem Test-`Client` für HTTP-Requests und Bequemlichkeitsfunktionen wie `reverse()` geraten automatische Tests mit Django zu einem wonnevollen Spaziergang. (pmk) c't



PyCharm Professional hilft beim Testen, indem die IDE die Ergebnisse der Coverage-Analyse direkt im Editor anzeigt. Getestete Zeilen haben links einen grünen Balken.



Web-GUIs testen mit Selenium

Automatisches Testen kann die Fehlerrate von Programmen drastisch reduzieren. Um die Oberflächen von Webanwendungen automatisch zu testen, muss der Rechner wie ein Mensch klicken und tippen. Mit dem Automatisierungsframework Selenium gelingt das mit wenigen Zeilen Code.

Von Nikolaus Schüler

Webseite aufrufen, Formular ausfüllen, Button anklicken, prüfen, ob der richtige Text angezeigt wird. Webanwendungen testen ist einfach, aber stupide. Genau diese drei Schritte muss ein Entwickler nämlich nach jeder Änderung am Quellcode erneut ausführen, genau wie zahlreiche andere ebenso stupide Abläufe. Selbst bei winzigen Bugfixes sollte er sich diese Arbeit auf keinen Fall sparen, da jede Änderung Seiteneffekte haben könnte, die er übersieht. Damit der Entwickler noch Zeit zum Entwickeln hat, statt nur zu testen, führt an automatischen Tests kein Weg vorbei.

Selenium erlaubt genau solche automatischen Tests von Web-GUIs. Das Framework steuert den Browser, wie ein Mensch das tun würde. Das heißt, es klickt auf Links und Buttons, füllt Textfelder aus oder wählt Einträge aus Listen aus. Zeigt der Browser dabei seine Oberfläche an, kann man dem Rechner sogar dabei zusehen, wie er wie von Geisterhand die Webseite bedient.

Selenium wird seit 2004 als freie Software entwickelt (siehe ct.de/w67p) und steht unter der Apache License. Das Framework bringt eine Reihe von Bindings mit, sodass man es aus Java, C#, Ruby,

JavaScript (mit Node.js) und Python nutzen kann. Wir zeigen Seleniums magische Fähigkeiten anhand einer winzigen Demo-App in Python. Die mit dem Python-Framework Flask programmierte Webanwendung zeigt lustige Kombinationen aus Vor- und Nachnamen an und erklärt bei Bedarf den Witz. Daher verwenden wir auch das Python-Binding von Selenium für die Tests. Selenium führt die Tests nicht selbst aus und prüft sie auch nicht. Darum kümmert sich das bei Python mitgelieferte `unittest`-Modul (siehe Seite 120).

Selenium-Tests funktionieren immer gleich: Man sucht in der Baumstruktur einer HTML-Seite (DOM) nach Elementen und interagiert danach mit ihnen. Die HTML-Elemente wählt man per Name, ID, Klasse oder XPath. Da eine Web-GUI aus nichts anderem als HTML besteht (eventuell mit JavaScript garniert), kann man damit alles auslösen, was ein Nutzer auch mit seinem Browser anstellen kann. Ob das HTML per Hand geschrieben oder mit JavaScript in den DOM eingefügt wurde, spielt dafür keine Rolle.

Kopflos durch die Nacht

Damit Selenium wie von Geisterhand Knöpfe drücken und Eingabefelder ausfüllen kann, braucht das Framework einen zum Browser passenden WebDriver. Bei Firefox ist das der `geckodriver`, bei Chrome der `chromedriver`. Beides sind kleine Hilfsprogramme (Download siehe ct.de/w67p), die Sie irgendwo in den `PATH` kopieren müssen, beispielsweise in ein Virtualenv unter `venv/bin/`. Außer den WebDrivern brauchen Sie nur noch das Framework selbst, das Sie ganz leicht mit `pip install selenium` installieren. Im Code erzeugen Sie dann einen `webdriver.Firefox()` oder einen `webdriver.Chrome()` und steuern mit diesem Objekt das Browserfenster.

Lässt man die Tests auf einem Continuous-Integration-Server (CI-Server) laufen, nutzt man die Browser lieber mit der `-headless`-Option, sodass sie kein Fenster öffnen. Im Code übergibt man die Option als Objekt:

```
def set_firefox(self):
  options = webdriver.FirefoxOptions()
  options.add_argument('-headless')
  self.browser = webdriver.Firefox(
                      options=options)
```

Chrome lässt den Strich vor der Option weg:

```
def set_chrome(self):
  options = webdriver.ChromeOptions()
  options.add_argument('headless')
  self.browser = webdriver.Chrome(
                      options=options)
```

Suchen und finden

Der erste Schritt jedes Tests besteht darin, die Webseite abzurufen. In der sucht man danach per ID, Name, Klasse oder XPath nach dem Element, mit dem man zuerst interagieren möchte. Folgerichtig nennt Selenium die Methoden dafür „Locators“. Eine Suche kann beispielsweise bei einer Klasse mehrere Elemente liefern. Enthält die Seite beispielsweise eine lange Liste, liefert ein Locator auf der Suche nach `<li>`-Elementen für jeden Listenpunkt einen Treffer. Selenium hat daher Locators in zwei Ausfertigungen: Eine im Singular formulierte wie beispielsweise `find_element_by_name()`. Hier liefert Selenium das erste gefundene Element zurück. Wo es sinnvoll ist, gibt es noch eine Methode im Plural, beispielsweise `find_elements_by_name()`, die eine Liste mit allen Treffern liefert. Eine Plural-Methode muss es nicht in allen Fällen geben. Da IDs eindeutig sein müssen, gibt es nur `find_element_by_id()`.

Nachdem Selenium das HTML-Element gefunden hat, ist die Arbeit des Frameworks erledigt. Für das Prüfen, ob Inhalt und Attribute des Elements den Erwartungen entsprechen, nutzt man nämlich das Testframework. Das stellt Assertions (auf Deutsch „Zusicherungen“) bereit, mit denen man die Erwartungen formuliert. Schlägt eine dieser Prüfungen fehl, protokolliert das Testframework den Fehler. Eine typische Assertion in Pythons `unittest`-Framework sieht so aus:

```
self.assertTrue(found)
```

Hier ist `found` eine Boolean-Variable. Die Assertion testet, ob sie wahr ist. Daneben gibt es noch Assertions, die auf Gleichheit testen, ob Elemente Teil einer Liste sind, ob eine Exception geworfen wird und jeweils welche für das Gegenteil.

`unittest` kümmert sich darum, die Anzahl der erfolgreichen und der fehlgeschlagenen Tests zu zählen und auszugeben.

Die App

Um den Umgang mit Selenium an einem realen Beispiel zeigen zu können, haben wir mit dem leichtgewichtigen Python-Framework Flask eine Web-Anwendung programmiert (siehe ct.de/w67p).

Unsere Anwendung präsentiert lustige Namen, die sie zufällig aus einer Liste auswählt. Die Namen muss man im Kopf umdrehen, um den Gag zu verstehen. Beispielsweise: Knito, Ingo → Ingo Knito → Inkognito, oder: Silie, Peter → Peter Silie → Petersilie.

Das GUI besteht aus drei Seiten: Die erste Seite zeigt einen Namen an. Ruft man die App auf, ohne einen Pfad anzugeben, wählt die App einen zufälligen Namen aus. Da Flask beim Entwickeln standardmäßig Port 5000 verwendet, erscheint der zufällige Name bei der URL `http://localhost:5000`. Gibt man dahinter `/name` und die Parameter `firstname` und `lastname` an, zeigt die App diesen Namen an. Beispielsweise `http://localhost:5000/name?firstname=Lore&lastname=Mipsum` für Frau Mipsum. Ein mit „Zeigs mir“ beschrifteter Button blendet den Namen in umgedrehter Form ein. Wer den Witz dann immer noch nicht versteht, kann sich mit dem neu erschienenen Button „Erklärs mir“ den Witz erklären lassen.

Auf einer weiteren Seite kann man nach einem Vor- oder Nachnamen suchen (Pfad: `/search/`). Drückt man den Suchen-Schalter, ohne wenigstens einen Vor- oder Nachnamen einzugeben, schlägt sie fehl. Ebenso, falls es keinen passenden Namen in der Datenbank gibt. Ein Test sollte besonders solche Grenz- und Fehlerfälle abfangen.

Die dritte Seite ist eine Liste aller Namen (Pfad: `/all/`). Die Liste zeigt neben jedem Namen gleich die Auflösung.

Die Namen kommen der Einfachheit halber nicht aus einer Datenbank. Die App liest sie stattdessen direkt aus einer CSV-Datei. Bei automatischen Tests ist das Verwenden einer „Fixture“ genannten Datei mit Dummy-Daten ein übliches Vorgehen: Die Tests sollen ja bei jedem Durchlauf gleich ablaufen. Würden sie Datenbankeinträge anlegen, könnte das andere Tests oder den nächsten Durchlauf beeinflussen. Fixtures fallen meist viel kleiner aus als reale Datenbanken, sodass die Tests schneller ablaufen. Außerdem enthalten sie keine datenschutzrelevanten Einträge, sodass die Tester keine Datenfreigabe brauchen. Niemand will beispielsweise reale Patientendaten in den Tests einer Krankenhaussoftware. Um unser Beispiel auszuprobieren, checken Sie es von GitHub aus und wechseln in sein Verzeichnis:

Automatische Testverfahren

Es gibt verschiedene Arten von automatisierten Tests: Unit-Tests, Integrationstests auf der Kommandozeile, mit GUI und im Browser.

Unit-Tests

Am bekanntesten sind wohl Unit-Tests. Sie sind Teil des Codes und testen hauptsächlich den „Kontrakt“ von Funktionen und Methoden. Beispielsweise testet ein Unit-Test eine Funktion, die eine Zahl quadriert. Der Test würde zunächst ein paar grundlegende Beispiele testen, also zum Beispiel `quadrat(4)==16` und `quadrat(5)==25`. Außerdem behandelt er Grenzfälle wie `quadrat(0)==0` (border conditions). Schließlich testet er noch, dass die Funktion die Vorzeichen richtig behandelt, also z. B. `quadrat(-5)==25`. Unit-Tests laufen typischerweise schnell ab, sodass man sie bei jeder Änderung laufen lassen kann, sei es neuer Code, sei es Refactoring.

Integrationstests

Das Verhalten eines ganzen Kommandozeilenprogramms testet man, indem man ein Programm schreibt, das das Programm aufruft und dann die Ausgabe prüft. Tests grafischer Oberflächen sind besonders schwer, weil die GUI-Elemente „bewegliche Ziele“ sind. Ihr Aussehen und ihre Position ändert sich. GUIs gibt es in zwei Varianten: native GUIs (Programme wie Libre Office oder Gimp) sowie Web-GUIs. Für Tests von Desktop-GUIs gibt es beispielsweise unter Java die Frameworks Marathon und uispec4j. Für Web-GUIs ist Selenium das Mittel der Wahl.

Test-Runner

Statt des in Python integrierten Moduls unittest kann man auch Nose oder Pytest verwenden. Diese Test-Runner muss man zwar separat installieren, dafür führen sie viele Tests auf einmal aus und bringen zahlreiche Komfortfunktionen mit, die eine ganze Menge Tipparbeit sparen.

```
git clone https://github.com/nikolaus: ↵
                        ↳.schueler/sillynames.git
cd sillynames
```

Mit `make venv` legen Sie dort ein Virtualenv an, damit die Pakete der App nicht anderen Python-Paketen in die Quere kommen. Mit `source venv/bin/activate` beziehungsweise `venv\bin\activate.bat` unter Windows aktivieren Sie das Virtualenv in der aktuellen Konsole. Die Pakete installieren Sie anschließend einfach mit `pip`:

```
pip install Flask Flask-WTF selenium
```

Stilfragen

Schön muss unsere Demo nicht sein, ein CSS-Stylesheet sollte sie aber trotzdem haben. Das Template base.html lädt es mithilfe der Funktion `url_for()`:

```
<link rel="stylesheet"
      type= "text/css"
      href="{{ url_for('static', ↵
                  ↳filename='css/style.css') }}"/>
```

Das Stylesheet setzt lediglich einen serifenlosen Font und eine (geschmacklich durchaus diskussionswürdige) Hintergrundfarbe.

Da Selenium direkt auf das DOM zugreift, findet es auch problemlos Elemente, die der Browser beispielsweise wegen eines zu großen Paddings nicht anzeigt. Ob der User ein Element tatsächlich sieht, kann man mit Selenium nicht testen.

Die Tests

Pythons Unittest-Framework kümmert sich ums Ausführen aller Testfunktionen und Einsammeln der Ergebnisse. Um das zu nutzen, schreibt man für die Tests eine Klasse, die von `unittest.TestCase` erbt. Enthält die Methoden, deren Name mit `test` beginnt, führt Unittest sie als jeweils einzelnen Test aus. Damit das beim Ausführen der Datei passiert, reichen folgende zwei Zeilen:

```
if __name__ == '__main__':
  unittest.main()
```

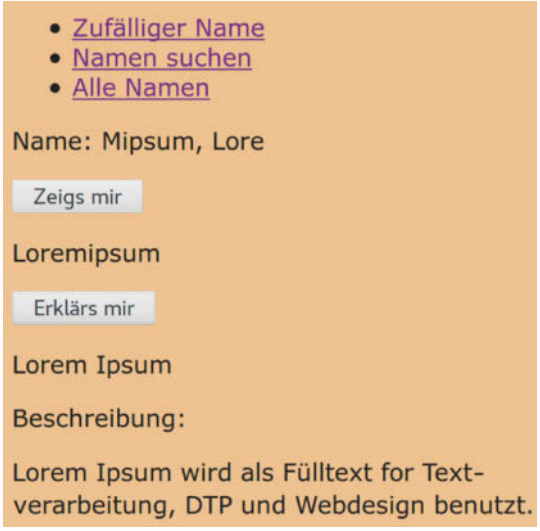
Implementiert man die Methode `setUp()`, führt Unittest sie vor jedem Test aus. Die Methode ist daher die ideale Ort, um den Selenium-WebDriver zu initialisieren und den Testmethoden das Fixture zur Verfügung zu stellen:

```
def setUp(self):
  self.set_firefox()
  self.browser.implicitly_wait(3)
  with open(names.CSV_FILE) as f:
    self.names=names.Name.from_csv(f)
```

Ganz ähnlich funktioniert `tearDown()`, womit man nach jedem Test wieder aufräumen kann:

```
def tearDown(self):
  self.browser.quit()
```

In den `test`-Methoden stehen außer den Membervariablen der Testklasse noch die `assert`-Methoden



Selenium kann auf die Buttons klicken, um die Erklärung einzublenden und das HTML-Element auszulesen. Ob der Text stimmt, überprüft jedoch das Unittest-Framework.

von `TestCase` zur Verfügung. Mit diesen prüft man boolesche Ausdrücke, Variablen auf Gleichheit, Einträge in Listen und ob Funktionen bestimmte Exceptions werfen. Was so überprüft wurde, taucht in der Zusammenfassung auf, die Unittest erstellt, nachdem es alle Tests ausgeführt hat.

Der erste Test soll prüfen, ob die App einen der Namen aus der CSV-Datei anzeigt, wenn man `http://localhost:5000` aufruft. Den Abruf der URL übernimmt der vom WebDriver ferngesteuerte Browser:

```
self.browser.get('http://127.0.0.1:5000')
```

Nun gilt es das HTML-Element mit dem Namen zu finden. Dafür eignet sich der „Inspektor“, den Firefox und Chrome mit F12 öffnen (auf dem Mac mit Alt+Cmd+I). Damit kann man Elemente in der Seite anklicken und bekommt alle relevanten Informationen über das Element. Bei dem `<p>` mit dem Namen beispielsweise, dass es die `id` „name“ hat. Mit der findet Selenium das Element und extrahiert seinen Inhalt:

```
t = self.browser.find_element_by_id(
                          'name').text
```

In `self.names` stehen die Namen in der Reihenfolge Vorname, Nachname, sodass der Test nicht einfach prüfen kann, ob `t` in `self.names` enthalten ist. Die Funktion `get_puzzle_name()` dreht den Namen um, sodass der Code ihn überprüfen kann. Da die App dieselbe Funktion verwendet, prüft der Test nicht, ob sie fehlerfrei ist, sondern lediglich, ob ihre Ausgabe im HTML landet. Ob die Funktion richtig arbeitet, sollte ohnehin ein getrennter Unittest prüfen, da sich das unabhängig von der Weboberfläche testen lässt.

Falls t vorn oder hinten Whitespace enthält, soll der Test nicht fehlschlagen. Deswegen entfernt der Test mit `t=t.strip()` alle Leerzeichen, Tabs und so weiter. Danach kann er prüfen, ob einer der Namen exakt übereinstimmt:

```
self.assertIn(t.strip(),
          [name.get_puzzle_name() for
           name in self.names])
```

Der nächste Test prüft, ob die Seite auch den unvertauschten Namen anzeigt. Dafür sucht der Test den Button „Zeigs mir“ und klickt ihn an:

```
self.browser.find_element_by_xpath(
        '//input[@value="Zeigs mir"]'
                         ).click()
```

Der Locator hangelt sich hier mit einem XPath durch das DOM, da der Button keine eindeutige `id` hat. Die Prüfung des Inhalts von `<p  id="show">` funktioniert genau wie zuvor nur mit der Funktion `get_funny_name()` statt `get_puzzle_name()`.

Ein wenig komplizierter wird es beim Test für die ausführliche Erklärung. Der Button, der die Erklärung einblendet, erscheint nämlich nur bei den Namen, bei denen auch eine ausführliche Erklärung in der CSV-Datei hinterlegt ist. Fehlt sie, zeigt die App gar keinen Button an. Ein Test, der den Button drückt, soll aber nicht fehlschlagen, falls er fehlt.

Findet Selenium ein Element nicht, wirft das Framework eine `NoSuchElementException`. Die kann der Test abfangen, um ohne Fehler abzubrechen, falls es den Button nicht gibt:

```
try:
  be=self.browser.find_element_by_id(
                        'explainbutton')
except NoSuchElementException:
  return
```

Ein Test, der kein `assert` ausführt, gilt als erfolgreich.

Such mich

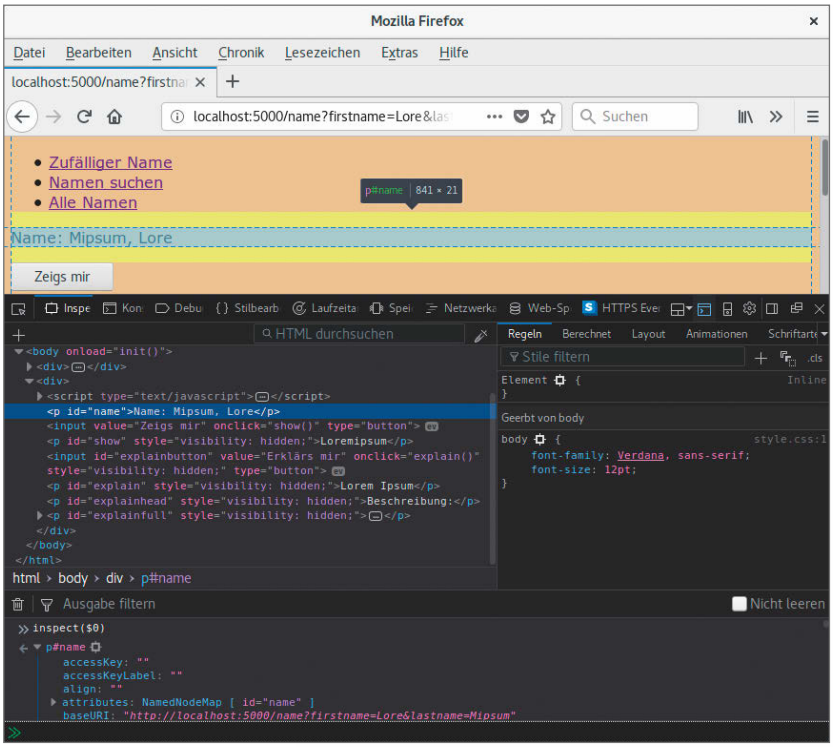
Der nächste Test gilt der Suchseite. Wenn die Suche erfolgreich ist, landet man auf der Seite für den gesuchten Namen. Das ist dieselbe Seite, die im vorherigen Test einen zufälligen Namen serviert hat. Diesmal weiß der Test aber, welchen Namen er erwartet, und prüft direkt das gewünschte Ergebnis.

Beim Suchen kann man aber auch viel falsch machen. Die Tests sollten nicht nur den Erfolg, sondern auch gerade die Fehlerbehandlung prüfen. Ein Nutzer könnte beispielsweise in die Suchfelder gar nichts eingeben und dann trotzdem den Submit-Button drücken. Oder er sucht nach einem Namen, den es nicht gibt. Beides sollte zu verschiedenen Fehlermeldungen führen.

Das Textfeld zur Eingabe eines Vornamens findet der Test mit einem XPath-Ausdruck:

```
self.browser.find_element_by_xpath(
          '//input[@name="firstname"]'
                  ).send_keys('Ingo')
```

Da der Test mit dem Eingabefeld nichts anderes machen will, als ein paar Buchstaben einzutippen, muss er ihn keiner Variable zuweisen.



Mit dem per F12 geöffneten „Inspektor" wählt man im Browser per Maus das zu testende HTML-Element aus (Icon links oben in der Ecke). Der Browser zeigt dann die nötigen Infos an, um das Element auch mit Selenium zu finden.

Anschließend klickt der Test den Suchknopf:

```
self.browser.find_element_by_xpath(
   '//input[@value="Suchen"]').click()
```

Das lädt die Ergebnisseite, auf der nun der Name stehen muss:

```
self.assertEqual(
      self.browser.find_element_by_id(
                'name').text.strip(),
      'Name: Knito, Ingo')
```

Bei Fehlern gibt es ein Extraelement mit dem Attribut `flashes`, in dem die App die Fehler in roter Schrift angezeigt. Die Fehlerbeschreibung holt sich der Test wieder aus names.py, damit er nicht fehlschlägt, nur weil man die Formulierung geändert hat:

```
def test_search_fail_empty(self):
  self.browser.get(
        'http://127.0.0.1:5000/search')
  self.browser.find_element_by_xpath(
   '//input[@value="Suchen"]').click()
  self.assertEqual(
   self.browser.find_element_by_xpath(
           '//ul[@class="flashes"]/li'
                       ).text.strip(),
   names.ERROR_EMPTY_SEARCH)
```

Analog testet ein zweiter Test die richtige Fehlermeldung, falls man nach einem Namen sucht, den es nicht gibt.

Und jetzt alle

Der letzte Test des Beispiels prüft, ob die Seite für alle Namen auch wirklich alle Namen anzeigt:

```
def test_all_names(self):
  self.browser.get(
          'http://127.0.0.1:5000/all')
  names_without_heading=self.names[1:]
  list_items=self.browser.find_
                  elements_by_xpath(
          '//li[@class="nameentry"]')
  self.assertCountEqual(
      [name.get_puzzle_name() for
       name in names_without_heading],
      [item.text.strip() for
       item in list_items])
```

Der XPath `//li[@class="nameentry"]` findet hier die ganze Liste mit Namen, da die `<li>`-Elemente alle die Klasse „nameentry" haben. Die Funktion `assertCountEqual()` vergleicht diese Liste mit der Liste aller Namen. Die Reihenfolge ist dieser Assertion egal. Entgegen des verwirrenden Namens prüft sie aber nicht nur die Länge. Möchte man auch die Reihenfolge prüfen, bringt Unittest dafür `assertListEqual()` mit.

Wir haben unsere Beispiel-App sehr einfach gehalten, sodass sie sich mit nur wenigen Tests bereits gut testen lässt. Bei realen Anwendungen sind erheblich mehr Tests nötig, um wirklich alle möglichen Eingabefehler und Klickwege zu prüfen. Die einzelnen Tests werden aber auch bei echten Anwendungen nicht viel komplexer als die hier gezeigten. Wer den Aufwand dennoch scheut, sollte aber zumindest darüber nachdenken, jeden gefundenen Bug mit einem automatischen Test zu prüfen. Das verhindert nämlich Regressionen, also dass man später versehentlich den gleichen Fehler wieder einbaut. (pmk) c't

Quellcode und Dokumentation:
www.ct.de/w67p

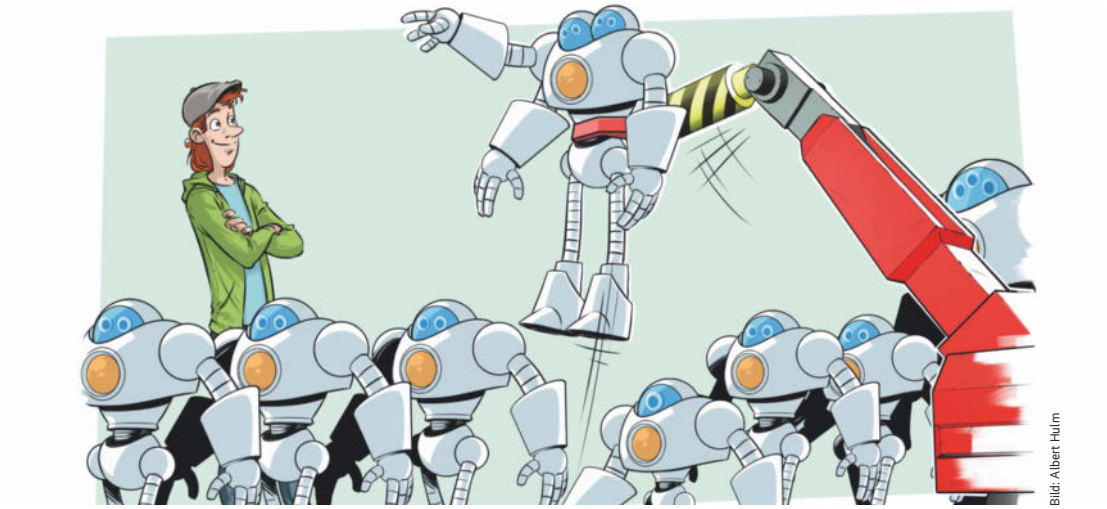


Bild: Albert Hulm

Mutationstests mit MutMut

Mit automatisiert verbuggtem Quelltext überprüfen Mutation-Tests die Qualität bestehender Tests. Das offenbart Lücken und ungeprüfte Randfälle, die einer simplen Coverage-Analyse entgehen.

Von Pina Merkert

Automatische Tests prüfen Funktionen oder auch ganze Programme mit wohlüberlegten Beispielen, ob sie korrekt arbeiten (siehe Seite 120). Im Idealfall durchlaufen sie dabei jede einzelne Zeile des Quelltexts. Doch auch in einem so getesteten Programm können sich Fehler verstecken. Ein Beispiel:

```
def elected(yes_ratio: float) -> bool:
  return yes_ratio > 0.5
```

Die Funktion liefert für eine Parlamentswahl die Entscheidung, ob die Zustimmung für eine einfache Mehrheit reicht. Ein Entwickler hat für diese Funktion folgende zwei Tests geschrieben, die 100 Prozent der Zeilen durchlaufen:

```
def test_elected():
  assert elected(0.6)
def test_not_elected():
  assert not elected(0.2)
```

Das Problem dabei: Keiner der Tests überprüft den Randfall `elected(0.5)`. Für 50 Prozent liefert sie nämlich `False`, obwohl `True` richtig wäre. Aber wie findet man ungenügende Tests wie diese?

Antwort: Indem man Fehler machen lässt! Fehler machen lässt sich nämlich automatisieren.

Die Idee dabei ist Folgende: Ein Programm sollte eigentlich einen einzigen Weg beschreiben, um ein korrektes Ergebnis zu berechnen. Ändert ein anderes Programm eine beliebige Stelle des Quell-

texts ab, sollte diese Änderung einen Bug produzieren. Laufen die Tests mit dieser fehlerhaften Version, sollten sie eigentlich fehlschlagen. Melden sie trotz verändertem Quelltext „OK“, testen sie das Programm nicht ausreichend.

Eine automatisch verbuggte Version des Programms bezeichnet man in diesem Zusammenhang als „Mutant“. Beim Mutation-Testing geht es nun darum, die Tests so umfassend zu programmieren, dass sie alle Mutanten „töten“, also die Fehler entlarven. Sollte ein Mutant „überleben“, ist das ein Grund, sich die Tests noch mal genauer anzusehen. Mutation-Testing testet also die Tests. Dieser Artikel zeigt, wie Sie die Tests einer Python-Anwendung mit dem Framework `mutmut` unter Beschuss nehmen und die überlebenden Mutanten jagen. Als Beispiel dient unser Markdown-zu-Asset-Storm-Konverter – Quellcode bei GitHub ct.de/wdhv (siehe Seite 172).

Abzählbar viele Mutanten

Um Mutanten zu erzeugen, darf das Mutation-Test-Framework nicht einfach zufällige Zeichen im Quelltext ändern: Solche Änderungen würden meist nur Syntaxfehler erzeugen. Stattdessen sucht das Framework Stellen, die es ändern kann, ohne die Syntax zu zerstören. Beim Beispiel oben könnte es die Konstante auf 1,5 ändern oder das > durch ein >= ersetzen. Der zweite Mutant würde zufälligerweise sogar den Fehler beheben. Aber darum geht es nicht: Die Tests töten diesen Mutanten nicht und daher kann das Framework diese Schwäche der Tests dokumentieren. Mit der Information, wie die Mutation aussieht, die die Tests nicht prüfen, lassen sie sich gezielt und schnell verbessern.

Der Mutantengenerator `mutmut` nutzt intern die Bibliothek Parso, um den Python3-Code zu parsen und alle Stellen zu finden, die es verändern kann. Da sich dafür nur bestimmte Sprachkonstrukte eignen (siehe Kasten) und ein Mutant stets nur eine modifizierte Stelle enthält, lassen sich die Mutanten leicht abzählen. Für obiges Beispiel gibt es zwei Mutanten, für das Beispielprojekt „Markdown2Asset-Storm“ 887.

Dank Parso verändert `mutmut` den Code flüchtig im Speicher, sodass sich die Rechenzeit in Grenzen hält. Dauert die Ausführung der Tests bei größeren Projekten etwas länger, multipliziert sich das aber mit der Zahl der Mutanten, sodass letztlich eine nicht unerhebliche Rechenzeit zusammenkommt. Mutation-Tests sind daher eine willkommene Aufgabe für CI-Server wie GitHub Actions [1].

Mutanten marsch!

MutMut selbst ist mit weniger als 1000 Zeilen Quellcode eine winzige Software. Am leichtesten installiert `pip` sie aus dem PyPi mit nur einem Befehl:

```
pip install mutmut
```

Zum Aufrufen genügt es anschließend, `mutmut` auf der Konsole einzugeben. Standardmäßig sucht `mutmut` die Tests im Verzeichnis `tests/` und versucht sie mit `pytest` aufzurufen. Den Quellcode sucht es

So mutiert MutMut den Code

Mutmut ersetzt Operatoren und modifiziert Konstanten, um syntaktisch korrekten, aber fehlerhaften Code zu erzeugen.

```
x=1      ➡  x=2          # type: int
x=1.3    ➡  x=2.3        # type: int
x="st"   ➡  x="XXstXX"   # type: str
lambda x: x+1  ➡  lambda x: None
x=1      ➡  x=None  # type: Any
deepcopy(x)    ➡  copy(x)
break          ⬌  continue
x is not False ➡  x is False
x is True      ➡  x is not True
1 in [1]       ➡  1 not in [1]
1 // 2  ➡  1 / 2
1 % 2   ➡  1 / 2
```

```
1 ** 2   ➡  1 * 2
1 <> 2   ➡  1 == 2
None     ➡  ""
True     ⬌  False
1 < 2    ⬌  1 <= 2
1 = 2    ⬌  1 != 2
1 * 2    ⬌  1 / 2
1 + 2    ⬌  1 - 2
1 << 2   ⬌  1 >> 2
1 & 2    ⬌  1 | 2
1 ^ 2    ⬌  1 & 2
a and b  ⬌  a or b
```

```
x += 2  ➡  x = 2
x -= 2  ➡  x = 2
x *= 2  ➡  x = 2
x /= 2  ➡  x = 2
x //= 2 ➡  x = 2
x %= 2  ➡  x = 2
x >>= 2 ➡  x = 2
x <<= 2 ➡  x = 2
x &= 2  ➡  x = 2
x |= 2  ➡  x = 2
x ^= 2  ➡  x = 2
```

```
(env) ~/Code/Markdown2AssetStorm$ mutmut run

- Mutation testing starting -

These are the steps:
1. A full test suite run will be made to make sure we
   can run the tests successfully and we know how long
   it takes (to detect infinite loops for example)
2. Mutants will be generated and checked

Results are stored in .mutmut-cache.
Print found mutants with `mutmut results`.

Legend for output:
🎉 Killed mutants.   The goal is for everything to end up in this bucket.
⏰ Timeout.          Test suite took 10 times as long as the baseline so were killed.
🤔 Suspicious.       Tests took a long time, but not long enough to be fatal.
🙁 Survived.         This means your tests needs to be expanded.

mutmut cache is out of date, clearing it...
1. Running tests without mutations
⠸ Running...Done

2. Checking mutants
⠸ 284/312  🎉 254  ⏰ 2  🤔 0  🙁 28
```

Nach einem Durchlauf zeigt mutmut results die IDs aller überlebenden Mutanten. Sie gehören auf die To-do-Liste des Testers.

beispielsweise unter `src/`. Für eine andere Verzeichnisstruktur oder einen anderen Test-Runner wie nose2 legt man am besten eine kleine Konfigurationsdatei namens `setup.cfg` an:

```
[mutmut]
paths_to_mutate=converter.py, helpers.py
backup=False
runner=nose2
tests_dir=tests/
dict_synonyms=Struct, NamedStruct
```

Der gesamte Code unseres Markdown-Konverters lebt in zwei Python-Dateien im Wurzelverzeichnis des Repositorys. Die Konfiguration in `setup.cfg` nimmt dafür eine kommaseparierte Liste von Dateinamen entgegen. Stattdessen darf dort aber auch ein ganzes Verzeichnis wie `sources/` stehen. MutMut erzeugt dann Mutanten für den gesamten Code in diesem Verzeichnis. Unsere Beispielkonfiguration gibt außerdem noch `nose2` als Test-Runner an.

Der Wettbewerb Mutanten vs. Tests startet mit:

```
mutmut run
```

Je nachdem, wie schnell die Tests laufen, dauert dieser Aufruf zwischen einigen Sekunden und vielen Minuten. `mutmut` informiert aber stets brav über den Fortschritt.

Mutantenjagd

Nach der Attacke auf den letzten Mutanten zeigt `mutmut results` eine Statistik, welche Mutanten überlebt haben. Jeder Mutant hat eine eigene ID, mit der man mit `mutmut show ID` Details zu dem Problem in der Testabdeckung abrufen kann:

```
$ mutmut show 11
-- converter.py
+++ converter.py
@@ -13,7 +13,7 @@
     response = app.response_class(
         response=json.dumps(data),
-        status=200,
+        status=201,
         mimetype='application/json'
     )
     return response
```

In diesem Fall gibt es keinen Test, der den Status-Code eines Requests in der Antwort der Webanwendung überprüft. Der Test zum Aufruf ist aber schnell erweitert:

```
self.assertEqual(200,
                 response.status_code)
```

Um Zeit zu sparen, sollte man den Mutanten nun einzeln testen:

```
(env) ~/Code/Markdown2AssetStorm$ mutmut results
To apply a mutant on disk:
    mutmut apply <id>

To show a mutant:
    mutmut show <id>


Timed out (2)

---- converter.py (1) ----

14

---- helpers.py (1) ----

220

Survived (31)

---- converter.py (2) ----

11, 15

---- helpers.py (29) ----

23, 24, 25, 26, 57, 60, 61, 74, 104, 111, 117, 124, 127, 128,
 129, 130, 131, 132, 133, 134, 135, 136, 137, 172, 180, 300,
303, 304, 311
```

MutMut erzeugt Mutanten und führt für jede dieser modifizierten Versionen des Codes alle automatischen Tests aus. Den Tests unseres Markdown-Konverters sind zunächst einige der Mutanten entwischt.

```
mutmut run 11
```

Wie gewünscht erwischt der erweiterte Test nun den Mutanten.

MutMut versucht, nur geänderte Tests erneut aufzurufen, und speichert dafür seinen Cache in der versteckten Datei `.mutmut-cache`. Im Test funktionierte die Änderungserkennung nicht immer zuverlässig, sodass `mutmut show` sporadisch kein Diff lieferte. Wir konnten das Problem aber stets lösen, indem wir den Cache löschten und die Mutanten neu erzeugten:

```
rm .mutmut-cache
mutmut run
```

Unsinnige Mutanten

In unserem Beispiel erkennt Mutant Nummer 15, dass die Tests über den direkten Aufruf der Datei den Start von Flasks Development-Server nicht testen. Die Zeile `if __name__ == __main__ :` hat aber keinen Einfluss darauf, ob die Anwendung korrekt funktioniert. Das Produktiv-Setup sollte den Development-Server ohnehin nicht starten. Um die Zeile aus den möglichen Mutationen auszuschließen, reicht ein Kommentar dahinter:

```
if (__name__ ==  # pragma: no mutate
    "__main__"):  # pragma: no mutate
    app.run()
```

Unendliche Mutanten

Das Beispiel zeigt außerdem bei Mutant 220 ein unlösbares Problem von Mutationstests. Der Mutant ändert dabei in einer `while`-Schleife die Zählvariable von `pos+=1` auf `pos=1`. Damit entfesselt er das berühmte Halteproblem: Die Änderung erzeugt eine Endlosschleife und die Tests liefern weder „OK“ noch „Fehler“, da sie schlicht nie enden. `mutmut` bringt für solche Fälle einen Timeout mit, sodass der Aufruf trotz Endlosschleife fertig wird. Mutant 220 zählt unter diesen Bedingungen allerdings nicht als getötet. `mutmut` sortiert ihn stattdessen in eine eigene Kategorie namens „Timeout“ ein.

Jäger und Tester

Mutation-Tests legen vollautomatisch den Finger in die Wunden der Tests: Was auf den ersten Blick nach zusätzlicher Arbeit aussieht, erweist sich in der Praxis als nützliches Tool, die neuralgischen Punkte in der Testabdeckung zu finden. Als Tester bekommt man mit Unterstützung durch den Mutanten-Generator viel schneller ein Gefühl dafür, wie umfassend die Tests dem Code auf den Zahn fühlen. Überleben keine oder nur wenige Mutanten, erzeugt das zu Recht das Gefühl, dass die Tests gut und (fast) vollständig prüfen. Das zeigt, dass man für späteres Refactoring und neue Features gut gerüstet ist.

Mit `mutmut` steht ein hervorragendes Modul für Python 3 zur Verfügung. Tools für Mutation-Testing gibt es aber für ganz viele moderne Programmiersprachen (beispielsweise PIT für Java, Humbug für PHP, Stryker für JavaScript, Stryker.NET für C#, Mull für C++ mit LLVM).

Es ist also fast egal, in welcher Sprache Sie programmieren: Die Mutantenjagd kann und sollte beginnen! (pmk) c't

Literatur

[1] Merlin Schumacher, **Und Actions!,** Erste Schritte mit GitHubs CI/CD-Werkzeug Actions, c?t 25/2019, S. 164

Beispielcode bei GitHub, Dokumentation:

www.ct.de/wdhv

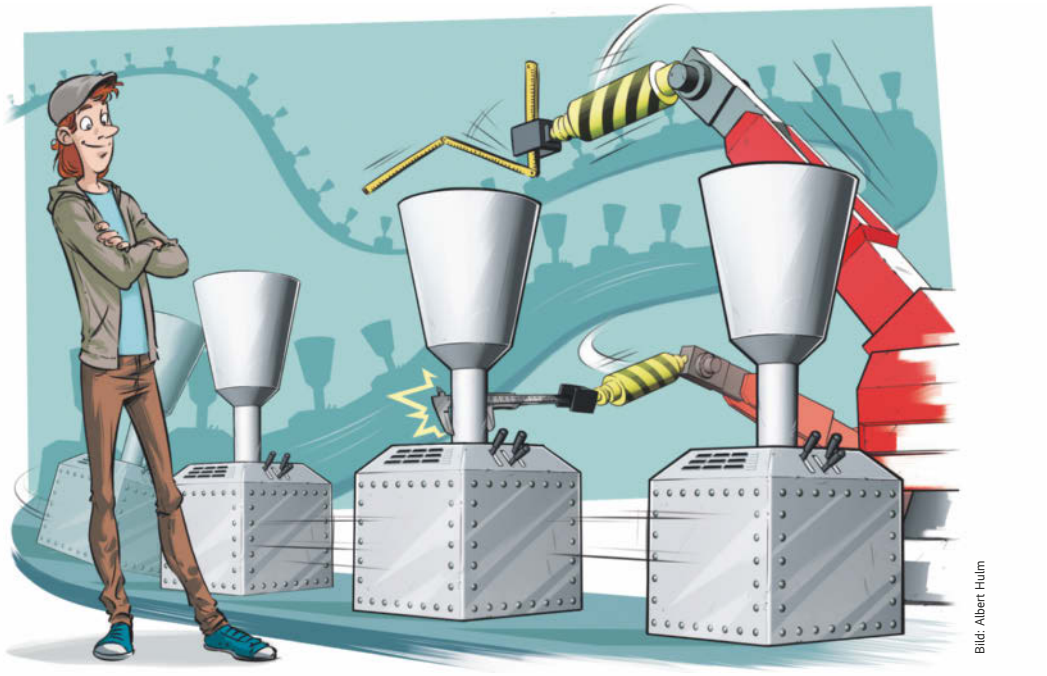


Bild: Albert Hulm

Hypothesis erfindet Testbeispiele

Das Test-Framework Hypothesis erzeugt automatisch hunderte von Beispielen für Integrations- und Unittests. Das spart Entwicklern duplizierten Code und hilft Fehler zu finden. Die Automatik funktioniert besonders effektiv, wenn die Tests dabei im Kreis rechnen.

Von Pina Merkert

Wer Unittests schreibt, muss kreativ sein: Jeder Test muss nämlich ein Beispiel mit möglichen Eingaben und der dazu passenden Ausgabe der getesteten Funktion definieren. Wie einfach das mit Pythons `unittest` geht, haben wir im Artikel „Programmierte Prüfer“ (siehe Seite 120) beschrieben. Normalerweise erfinden die Tester diese Beispiele und kodieren sie in Handarbeit in die Tests.

Dabei ist es eine Kunst, die richtigen Beispiele zu finden, die alle Randfälle und Ausnahmen abdecken. Mutation-Tests (siehe Seite 136) helfen zwar, ungenügende Tests zu identifizieren, der Fix besteht aber zumeist aus weiteren Tests mit mehr Beispielen, was den Test-Code schnell wachsen lässt.

Mit dem Python-Framework Hypothesis umgeht man elegant die Test-Code-Adipositas, indem man das Erzeugen von Beispielen an den fleißigen

Rechner delegiert. Der erschafft flugs hundert Beispiele statt nur eines und merkt sich brav, falls eines der zufälligen Beispiele fehlschlägt. Damit das funktioniert, muss man Hypothesis lediglich per Decorator mitteilen, aus welcher Menge es die Beispiele aussuchen darf. Als ganz normale Parameter der Test-Funktion fließen sie ohne Staustufen in den Programmfluss.

Mathe-Sprech

Hypothesis findet mit `pip install hypothesis` auf allen Systemen den Weg in die virtuelle Python-Umgebung. Danach reichen ein paar `import`-Zeilen und der `@given()` -Decorator, um Tests mit hunderten Beispielen laufen zu lassen. Wie das konkret aussieht, zeigt das folgende Beispiel (siehe `tests/test_playground.py` im Git-Repository unter ct.de/w3ma):

```
import unittest
import math
from hypothesis import given
from hypothesis.strategies import integers
def square(x: int) -> int:
  return x*x
def sqrt_rounded(x: int) -> int:
  return int(round(math.sqrt(x)))

class HypothesisPlaygroundTestCase(
                                unittest.TestCase):
  @given(integers())
  def test_square_and_sqrt(self, x):
    self.assertEqual(x,
          sqrt_rounded(square(x)))
```

Die erste Funktion quadriert Ganzzahlen, die Zweite zieht die Wurzel und rundet. Beide hintereinander sollten die gleiche Zahl ausgeben, die sie als Eingabe erhielten. Der Test prüft genau das. Anders als üblich nimmt die Test-Funktion aber den Parameter x entgegen. Den befüllt Hypothesis mit dem Decorator `@given()`. Der braucht allerdings die Angabe, aus welcher Menge er den Parameter befüllen darf. Hypothesis nennt diese Mengen „Strategie“ mit der Basisklasse `BasicStrategy`. Ganzzahlen liefert `hypothesis.strategies.integers()`.

Zum Ausführen dieses Tests eignen sich die gleichen Test-Runner, wie bei anderen automatischen Tests in der Python-Welt auch. Unser Beispiel bei GitHub nutzt `pytest`:

```
python -m pytest
```

Randbemerkung: Mathematiker merken sich den Namen der Funktion besonders leicht. Der Python-Code formuliert nämlich den üblichen Satzbau, mit dem Mathematiker einen Beweis anfangen: „Gegeben x aus der Menge der ganzen Zahlen, testet die Funktion …“ Als Formel ausgedrückt sieht das so aus:

$$\forall x \in \mathbb{Z}$$

Simplify

Obiger Code läuft zwar, Hypothesis findet aber sofort ein Gegenbeispiel: `x=-1`. Beim Quadrieren einer Zahl geht das Vorzeichen natürlich verloren, weshalb es sinnvollerweise zwei Tests geben sollte: Einer testet nur positive Ganzzahlen, der Zweite nur negative und multipliziert zusätzlich mit `-1`. Als Reparatur des Problems nimmt die Strategie `integers()` für Test 1 einfach den Parameter `min_value=0` entgegen. Test 2 verwendet entsprechend `max_value=0`.

Wer sich nun wundert, dass Hypothesis für sein Gegenbeispiel zufälligerweise gerade die größte negative Ganzzahl ausprobiert hat, wundert sich zu Recht. Hypothesis hat nämlich vermutlich zuerst eine viel kleinere „krumme“ Zahl ausprobiert. Im Code und in der Ausgabe sieht man von krummen Beispielen aber nichts, da Hypothesis sie versteckt. Mit der `simplify()`-Methode der Strategie versucht Hypothesis nämlich, automatisch ein möglichst einfaches Beispiel zu finden, das den Test ebenfalls fehlschlagen lässt. Als Anwender bekommt man nur das einfache Beispiel zu sehen. Das Schöne an `simplify()` ist, dass Hypothesis sich durch diesen Mechanismus automatisch an Randfälle herantastet.

Ungenaue Kommazahlen

Mit `min_value=0` probiert Hypothesis keine negativen Zahlen mehr. Der Test schlägt aber weiterhin fehl. Nun findet Hypothesis ein Gegenbeispiel mit einer sehr großen Zahl. Das liegt daran, dass in ein Register keine beliebig großen Zahlen passen. Auf einem üblichen Rechner mit 64-Bit Betriebssystem ist bei der Ganzzahl 2^{63}-1=9223372036854775807 schluss. Python liefert diesen Wert mit dem `sys-Modul:`

```
import sys
print(sys.maxsize)
```

Beim Quadrieren entstehen schnell Ganzzahlen, die Python mit mehr als 64 Bit speichern muss. Das

größte x, das quadriert noch in 64 Bit passt, ist 3037000499. Ausgestattet mit dieser oberen Grenze läuft der Test fehlerfrei durch:

```
@given(integers(min_value=0,
          max_value=3037000499))
def test_square_and_sqrt(self, x):
  self.assertEqual(x,
          sqrt_rounded(square(x)))
```

Beispiel-Erfinder

Dass Hypothesis automatisch Beispiele erfindet, ist enorm praktisch. Schließlich versprechen viele Beispiele eine gute Test-Abdeckung und fehlerfreien Code. Doch die Beispiele eines automatischen Tests (egal ob Unit- oder Integrationstest) bestehen immer aus einer Eingabe und der dazu passenden Ausgabe. Hypothesis erzeugt zwar mögliche Eingaben en masse, bleibt die zugehörige Ausgabe aber leider schuldig.

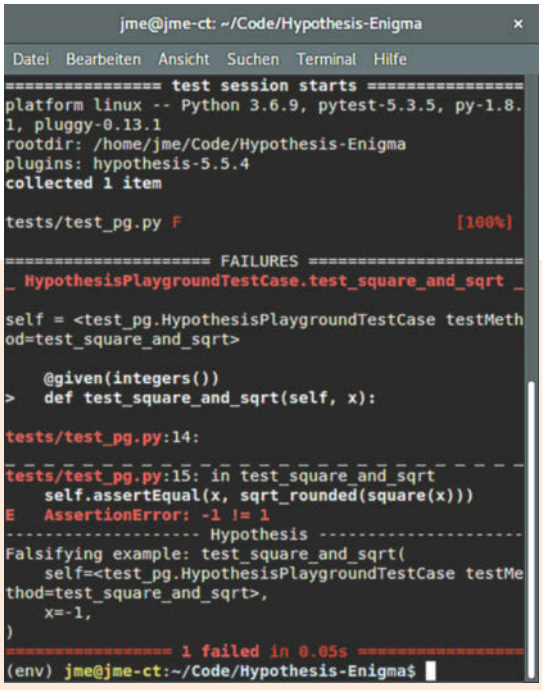
Eine Möglichkeit, damit umzugehen, besteht darin, dass der Test nur grobe Rahmenbedingungen testet. Ob die Funktion im Detail korrekt rechnet, prüft ein solcher Test zwar nicht erschöpfend, Abstürze und Exceptions triggert das aber schon, sodass mit Hypothesis diese Variante des Fuzzing möglich ist.

Erschöpfende Tests zu schreiben, fällt dagegen schwerer, da die Tests die passende Ausgabe zur Eingabe berechnen müssen. Die Macher stellen hier das Testen von Eigenschaften „property based testing“ dem Testen von Beispielen „example based testing“ gegenüber. Die „Eigenschaften“ (Propertys) definieren die Strategien und Hypothesis kümmert sich ums Erfinden der Beispiele.

Tests mit Bit-genauer Prüfung der Ergebnisse gelingen manchmal mit Reimplementierungen der gleichen Funktion beim Refactoring oder dem Aufruf fremder Bibliotheken, die man nachprogrammiert. Im Allgemeinen ist die Berechnung für das passende Beispiel zur Eingabe aber schwer zu finden.

Rechnen im Kreis

Eine besonders effiziente Lösung dieses Problems besteht darin, nicht nur eine Funktion zu testen, sondern gleich zwei. Die erste Funktion wandelt die Eingaben in eine Ausgabe, deren Korrektheit der Test nicht prüfen kann. Die Ausgabe der ersten Funktion verwendet die zweite Funktion als ihre Eingabe und kehrt die Berechnung um. Vorsicht: Das geht natürlich nicht mit jeder Funktion. Am Ende fallen, wenn es geht, aus der zweiten Funktion die gleichen Daten heraus, die in die erste Funktion hinein gepurzelt sind.



So sieht es aus, wenn Hypothesis ein Gegenbeispiel findet: Der Test schlägt fehl, allerdings mit einem vereinfachten Beispiel, das die Fehlersuche erleichtert.

Eine solche Kombination aus umkehrbaren Berechnungen bildet einen Kreis. Im Prinzip könnten die beiden Funktionen die Daten unendlich oft und ohne Verluste hin und her wandeln. Dabei müssen beide beteiligten Funktionen korrekt funktionieren und Hypothesis kann mit immer neuen Beispielen prüfen, ob keine von beiden einen Fehler macht.

Im kompakten Beispiel zu Beginn des Artikels sind die beiden Funktionen Quadrieren und Wurzel ziehen. Der Test prüft nur, ob die Kombination aus beidem funktioniert. Das Beispiel zeigt, dass Hypothesis dabei Fehler findet, an die man nicht sofort denkt.

Enigma

Einen ähnlichen Kreis stellen Verschlüsselungsverfahren dar. Ergebnis der Verschlüsselung (Funktion 1) ist ein Ciphertext, der dem Klartext nicht mal entfernt ähnlich sieht. Beim Entschlüsseln (Funktion 2) wird aus dem rätselhaften Ciphertext wieder derselbe Klartext, den die Verschlüsselung einst als Eingabe entgegennahm.

Als Beispiel haben wir das Prinzip der berühmten Enigma, einer mechanischen Verschlüsselungsmaschine, in Python nachgebaut. Die deutsche Wehrmacht nutzte Enigmas im Zweiten Weltkrieg, um Funksprüche zu verschlüsseln. Einem englischen Team um Alan Turing gelang es jedoch, mit enormem Aufwand die vermeintlich unknackbare Verschlüsselung der Enigma zu knacken.

Die Enigma nutzt vier Walzen und ein Steckbrett, die jeweils Buchstaben durch andere Buchstaben ersetzen können. Gibt ein Bediener einen Buchstaben über eine Tastatur ein, schließt das Kontakte zu den Walzen und eine zum verschlüsselten Buchstaben gehörende Lampe leuchtet. Gleichzeitig drehen sich die Walzen wie ein Uhrwerk weiter, sodass beim Verschlüsseln des nächsten Buchstabens eine ganz andere Lampe leuchtet, selbst wenn der Bediener dieselbe Taste noch mal drückt.

Der täglich wechselnde Schlüssel einer Enigma (die Wehrmacht hatte dafür Codebücher verteilt) besteht aus der Auswahl bestimmter Walzen. Es gab fünf verschiedene, die man an drei verschiedenen Stellen einbauen konnte. Dazu kam eine besondere Reflektor-Walze (hier standen drei zur Wahl). Zuletzt mussten die Bediener noch eine Konfiguration an Steckern, die am Steckbrett Buchstaben vertauschte, einstellen. Außerdem konnten die drei beweglichen Walzen auf eine von je 26 verschiedenen Startpositionen verdreht werden. Das `Enigma`-Objekt unserer Implementierung nimmt im Konstruktor eine Konfiguration all dieser Eigenschaften entgegen.

Unsere `Enigma` funktioniert vollautomatisch und stellt zum Verschlüsseln die Methode `encrypt()` und zum Entschlüsseln die Methode `decrypt()` zur Verfügung. Zum Testen muss man sie nicht nur mit einem passenden Klartext füttern (die Enigma frisst nur Großbuchstaben und kennt weder Leer- noch

Quadrate > 64 Bit

Wenn beim Quadrieren ein langer int herauskommt, ist das für Python zunächst kein Problem. Die Sprache arbeitet transparent mit beliebig großen Ganzzahlen. Beim Wurzelziehen liefert `math.sqrt()` aber einen `float`, den der Rechner ebenfalls mit 64 Bit speichert. Die Genauigkeit von `float` nimmt aber ab, je größer die Zahlen werden. Die Wurzel des Quadrats von 9007199254769004 und die von 9007199254769005 liefern daher den selben `float`. Hypothesis findet dieses Problem verlässlich. Abhilfe schafft eine Prüfung auf 64 Bit in `square()` und ein weiterer Test:

```
def square(x: int) -> int:
  sq = x*x
  if sq > sys.maxsize:
    raise ArithmeticError(
      "{0}*{0} is bigger than ?" +
      "?the maximum int.".format (x))
  return sq
@given(integers(min_value=3037000500))
def test_square_over_64_bit(self, x):
  self.assertRaises(ArithmeticError,
                    square, x)
  try:
    square(x)
  except ArithmeticError as ex:
    self.assertEqual(
       "{0}*{0} is bigger than ?" +
       "?the maximum int.".format(x),
         str(ex))
```

Satzzeichen), sondern auch mit ihrer Walzen- und Kabelposition:

```
def test_encrypt_decrypt(self,
      r1p: int, r2p: int, r3p: int,
      patch_key: str,
      rotor_selection: list,
      reflector_no: int,
      plaintext: str):
enigma = Enigma(r1_pos=r1p,
   r2_pos=r2p, r3_pos=r3p,
   patch_key=patch_key,
   rotor_selection=rotor_selection,
   reflector_selection=reflector_no)
cipher_text = enigma.encrypt(plaintext)
self.assertNotEqual(cipher_text, plaintext)
self.assertEqual(plaintext,
        enigma.decrypt(cipher_text))
```

Strategie-Sammlung

Die Test-Funktion gibt `@given()` einiges zu tun: Die drei Walzenpositionen kann Hypothesis wie im Einstiegsbeispiel mit der `integers()`-Strategie (Wertebereich 0 bis 26) erzeugen. Beim Patch-Key, also den per Steckbrett und Kabel vertauschten Buchstaben, wird es aber schwieriger. Jeder Buchstabe darf in dem String genau ein Mal vorkommen. Die Reihenfolge ist aber beliebig. Eine solche Kette ist eine Permutation des Alphabets und Hypothesis bringt eine praktische Strategie dafür mit: `permutations(list('ABCDEFGHIJKLMNOPQRSTUVWXYZ'))`

Bei den eingebauten Walzen (`rotor_selection`) stehen fünf zur Wahl, wovon aber nur drei in der Enigma Platz finden. Unsere `Enigma`-Implementierung ignoriert bei einer Liste von Walzennummern alle Elemente hinter dem Dritten, sodass Hypothesis hier ebenfalls mit der `permutations()`-Strategie Listen liefern darf: `permutations(list(range(5)))`

Die `reflector_selection` erzeugt Hypothesis wieder ganz einfach mit `integers(min_value=0, max_value=2)`.

Strategien in Strategien

Nun fehlt nur noch der Funkspruch als `plaintext`. Funksprüche bestehen aus normalen Wörtern, Eigennamen und Satzzeichen. Eigennamen hat die Wehrmacht stets mit `X`en umschlossen und verdoppelt. Satzzeichen kommen auf der Tastatur der Enigma nicht vor, weshalb die Funker sie durch `X` ersetzt haben.

Bei der Enigma-Verschlüsselungsmaschine tippt man wie bei einer Schreibmaschine Buchstaben auf der Tastatur. Zu jedem Buchstaben leuchtet oberhalb der Tastatur eine Lampe mit einem anderen Buchstaben. Aus der Folge der leuchtenden Buchstaben ergibt sich der Ciphertext.

Zum Generieren von Strings stellt Hypothesis die `text()`-Strategie bereit. Die erzeugt aber beliebige Sonderzeichen und auch den leeren String. Für die Wörter der Funksprüche eignet sich die alternative Strategie `from_regex()` besser. Diese Strategie erzeugt auch Strings, allerdings nur solche, die die angegebene Regular Expression erfüllen. Ausschließlich aus Großbuchstaben bestehende Wörter mit 1 bis 15 Buchstaben erzeugt

```
normal_word = from_regex(r'[A-Z]{1,15}', ↵
                                ↳fullmatch=True)
```

Ein Eigenname hat im Prinzip dasselbe Format, weshalb der Code die eben definierte Strategie weiter verwendet. Um die Wehrmacht-Regel mit der Verdoppelung und den X'en umzusetzen, eignet sich die `.map()`-Methode, die jede Strategie mitbringt. Ihr übergibt man ein Callable als Parameter, das die Funktion ausführt. Eine `lambda`-Funktion bietet sich als solches Callable an:

```
name = normal_word.map(lambda w:
          "X" + w + "X" + w + "X")
```

In einem Satz darf Hypothesis stets zwischen Namen und normalen Wörtern wählen. Eine solche Auswahl setzt die `one_of()`-Strategie um.

Der Satz entsteht nun als Liste (`lists()`-Strategie) aus Wörtern oder Eigennamen, die Hypothesis als String ohne Leerzeichen zusammensetzt. Der Satz muss dabei mindestens ein Wort lang sein; aus Performance-Gründen haben wir ihn auf maximal zwölf Wörter begrenzt. Ans Ende hängt die Strategie noch ein Satzzeichen, bestehend aus einem `X`. Ein Funkspruch besteht aus einem bis fünf Sätzen oder Nebensätzen, die das gleiche Format haben:

```
def radiogram() -> SearchStrategy:
  normal_word = from_regex(
                    r'[A-Z]{1,15}', fullmatch=True)
  name = normal_word.map(
                  lambda w: "X" + w + "X" + w + "X")
  sentence = lists(
                elements=one_of(normal_word, name),
                      min_size=1, max_size=12).map(
                        lambda x: "".join(x) + "X")
return lists(sentence,
                       min_size=1, max_size=5).map(
                              lambda s: "".join(s))
```

Die Funktion `radiogram()` erzeugt die beschriebene Hypothesis-Strategie für die Funksprüche. Der `@given()`-Decorator vor der Test-Funktion sieht damit insgesamt so aus:

```
@given(
 integers(min_value=0, max_value=26),
 integers(min_value=0, max_value=26),
 integers(min_value=0, max_value=26),
 permutations(list('ABCDEFGHIJKLMNOPQRSTUVWXYZ')),
 permutations(list(range(5))),
 integers(min_value=0, max_value=2),
 radiogram()
)
```

Der ewige Kreis

Die Hypothesis-Strategien lassen sich zwar toll kombinieren, die Rechenzeit steigt dabei aber erheblich an. Bei einer komplizierten Strategie wie den Funksprüchen entfällt ein nennenswerter Anteil der Zeit für die Ausführung der Tests auf das Erzeugen der Beispiele. Der folgende Code, der lediglich 100 Funksprüche ausgibt, braucht auf einem Core i5 bereits fast 2,5 Sekunden:

```
print("100 Beispiele für Funksprüche:")
for _ in range(100):
  print(" -", radiogram().example())
```

Findet Hypothesis einen Fehler, probiert das Framework beim Versuch, das Beispiel mit `simplify()` zu vereinfachen, eine Menge weitere Beispiele aus, was ebenfalls recht lange dauert. Bei komplexeren Hypothesis-Tests lohnt es sich daher, einen CI-Server einzurichten (beispielsweise GitHub-Actions [1], bei sehr langen Laufzeiten mit eigenem Runner [2] auf eigener Harware).

Für die Code-Qualität lohnt sich die Rechenzeit auf jeden Fall. Denn mit einer umfassenden Strategie erreicht ein Hypothesis-Test eine enorme Testabdeckung. Das Problem besteht eher darin, ein Szenario zu finden, in dem der Test im Kreis rechnen kann. Fehlt diese Möglichkeit, kann man bei Hypothesis-Tests nämlich meist nur grobe Rahmenbedingungen testen. Klassische Tests, wie auf Seite 120 beschrieben, mit Beispielpaaren sind dann die bessere Wahl.

Ist das Programm allerdings in der Lage, die gewünschte Ausgabe zu jedem automatisch erzeugten Beispiel zu berechnen, sollte jeder Entwickler die Gelegenheit beim Schopf packen und die wenigen nötigen Zeilen Quellcode runterschreiben. Hypothesis macht das für Python erstaunlich einfach, aber auch für andere Sprachen existieren Test-Frameworks, die der gleichen Idee folgen. (pmk)

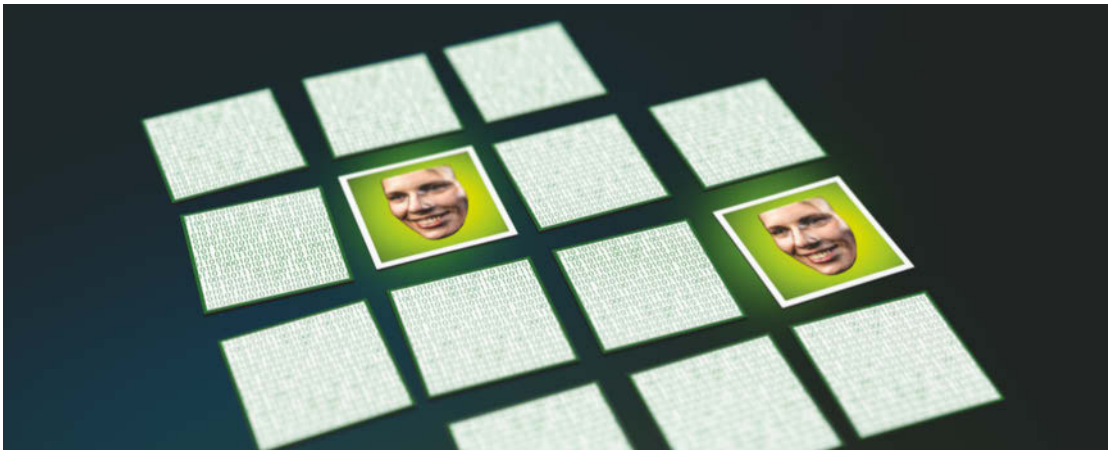
Literatur

[1] Merlin Schumacher, **Und Actions!,** Erste Schritte mit GitHubs CI/CD-Werkzeug Actions, c't 25/2019, S. 164

[2] Merlin Schumacher, **Heimarbeit,** Eigene Runner für GitHub Actionseinrichten, c't 3/2020, S. 150

Beispielcode bei GitHub, Dokumentation zu Hypothesis

www.ct.de/w3ma



Binäre Suche in 25 GB Passwörtern

Auf haveibeenpwned.com kann man eine sortierte Liste mit Hashes von Passwörtern aus Hacks und Leaks herunterladen. Entpackt ist die knapp 25 Gigabyte groß. Mit in Python implementierter binärer Suche ist sie trotz ihrer Größe in wenigen Millisekunden durchsucht.

Von Pina Merkert

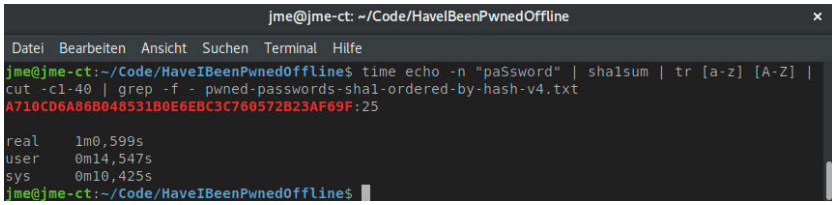
Die Webseite Haveibeenpwned.com sammelt seit Jahren Passwörter aus Leaks und Hacks. Inzwischen sind über 500 Millionen Klartext-Passwörter bei dem Dienst aufgelaufen, die Nutzer auf keinen Fall mehr benutzen sollten. Um zu prüfen, ob das eigene Passwort dabei ist, bietet der Dienst einen Webservice an. Die Webseite verschickt zwar nur auf fünf Zeichen gekürzte Hashes übers Netz, aber um sicherzugehen, dass wirklich kein Passwort im Klartext verschickt wird, müsste man vor jeder Nutzung den JavaScript-Code der Seite prüfen. Bei unserem Python-Skript können Sie sich das sparen: Es ist lokal auf Ihrer Platte gespeichert und dort vor unerwarteten Änderungen geschützt. Es arbeitet mit einer entpackt 25 Gigabyte großen Datei mit SHA-1-Hashes der geleakten Passwörter; Sie können die Datei als 10 Gigabyte großes 7Zip-Archiv bei haveibeenbwned.com (siehe ct.de/w2pk) herunterladen.

Abgetrennt durch einen Doppelpunkt enthält jede Zeile neben dem Hash in Hex-Darstellung auch die Anzahl, in wie vielen Leaks das Passwort bereits aufgetaucht ist. Eine Zeile der Datei sieht beispielsweise so aus:

```
7FF579BAE8FCFE030E099BC9CA6414B5C76B185A:12
```

Um in dieser Datei nach dem eigenen Passwort zu suchen, erstellt man einen SHA-1-Hash des eigenen Passworts (als Hex-String) und sucht alle Zeilen, in denen dieser auftaucht. Das geht auch ohne unser Python-Programm auf der Konsole mit:

Die Suche mit Grep funktioniert, dauert aber länger als eine Minute.




```
echo -n "paSsword" | sha1sum | ↵
      ↳tr [a-z] [A-Z] | cut -c1-40 | ↵
     ↳grep -f - pwned-passwords-sha1-↵
               ↳ordered-by-hash-v5.txt
```

`sha1sum` erstellt einen Hash in Hex-Darstellung, nur leider mit `a` bis `f` in Kleinbuchstaben. Die konvertiert `tr [a-z] [A-Z]` in Großbuchstaben und `cut -c1-40` schneidet unnötige Zeichen hinten ab. Mit dem Ergebnis sucht `grep`, das mit der Option `-f -` den zu filternden String von `stdin` liest. Der lange Dateiname ist der Name der Datei, wie er aktuell aus dem Archiv fällt.

Sortierung nutzen

Der Grep-Befehl durchsucht die Datei auf einem Rechner mit Core i5 mit SSD in etwas mehr als einer Minute. Er nutzt aber nicht aus, dass die Datei bereits nach Hashes sortiert ist. In einer sortierten Liste kann man per binärer Suche viel schneller suchen. Eine selbst programmierte binäre Suche in Python braucht nur wenige Zeilen Code. Wir haben daher kurzerhand ein Skript entwickelt, das die Datenmassen in Rekordzeit durchforstet (siehe ct.de/w2pk).

Damit Sie sicher sein können, dass dieser Code nichts Ungewolltes mit Ihren Passwörtern anstellt, erklären wir ihn en détail. Sie sind explizit aufgerufen, uns kritisch auf die Finger zu schauen.

Teile und herrsche

Die zentrale Idee der binären Suche besteht darin, einen Hashwert aus der Mitte der Liste herauszugreifen und mit dem Gesuchten zu vergleichen. Ist der gesuchte Hash kleiner als der herausgegriffene, muss der gesuchte Hash in der vorderen Hälfte der Liste zu finden sein. Ist er dagegen größer, kommt nur noch die hintere Hälfte der Liste infrage.

Hat man das herausgefunden, muss man nur noch eine halb so lange Liste durchsuchen, wofür man den gleichen Trick verwendet. Dieses Spiel spielt man, bis die zu durchsuchende Liste nur noch aus einem Eintrag besteht. Mit diesem vergleicht man den gesuchten Hash und erkennt daran, ob er in der Liste vorkam oder nicht.

Wir haben diese Idee mit einer rekursiven Funktion namens `search_hash()` umgesetzt. Sie nimmt neben der Datei mit der ganzen Liste auch einen über die Parameter `start` und `end` definierten Bereich entgegen. Dessen Größe halbiert sie bei jedem Aufruf und ruft sich dann selbst mit dem neuen Bereich auf. Dafür sucht sie sich zunächst die Mitte des Bereichs:

```
new_pos = start + (end - start) // 2
```

Diese Position ist aber die Position eines Bytes in der Datei. Ob an dieser Position eine Zeile anfängt oder sie in die Mitte einer Zeile verweist, weiß der Algorithmus nicht. Um den Hash zu vergleichen, muss er aber immer die ganze Zeile lesen. Das erledigt die Funktion `get_full_line()`:

```
def get_full_line(file, pos):
  file.seek(pos)
  while (pos > 0 and
         file.read(1) != "\n"):
    pos -= 1
    file.seek(pos)
  return file.readline(), pos
```

Die Methode `seek()` springt sehr schnell an eine bestimmte Position in einer Datei. Von dort aus sucht sie den Beginn der Zeile. Dafür geht sie rückwärts und liest so lange einzelne Bytes aus der Datei, bis sie einen Zeilensprung findet (`"\n"`). Von diesem Zeilenanfang ist es ein Leichtes, mit `readline()` die ganze Zeile auszulesen. Als zweiten Rückgabewert liefert die Funktion auch die Byte-Nummer des zuvor gesuchten Zeilenanfangs, da der Algorithmus den später nutzt, um den Suchbereich einzugrenzen.

Mit dieser Funktion lädt `search_hash()` die mittlere Zeile im Bereich. Die enthält neben dem Hash auch die Anzahl, was sich mit `split(':')` leicht trennen lässt. Danach gilt es zunächst zu prüfen, ob

der gesuchte und gefundene Hash übereinstimmen. Ist das der Fall, gibt `search_hash()` die Anzahl zurück und der Algorithmus ist fertig.

Stimmen sie nicht überein, muss die Funktion rekursiv weitersuchen. Dafür entscheidet `search_hash()`, ob die Suche in der vorderen oder hinteren Hälfte weitergeht. Es reicht dafür, die hexadezimalen Strings der Hashes zu vergleichen. Der Vergleich prüft die Strings nämlich zeichenweise und die Buchstaben haben einen größeren Unicode als die Ziffern. Es ist daher `A` > `9` und `A000` > `8AAA`:

```
def search_hash(file, my_hash,
                start, end):
  if start >= end:
    return 0
  new_pos = start + (end - start) // 2
  andidate_line, pivot=get_full_line(
                       file, new_pos)
  pwned_hash, count = candidate_line.\split(':')
  if pwned_hash == my_hash:
    print("Password found at byte ↵
          ↳{:11d}: \"{}\"".format(
       pivot, candidate_line.strip()))
    return int(count.strip())
  if my_hash > pwned_hash:
    return search_hash(file, my_hash,
                               file.tell(), end)
  else:
    return search_hash(file, my_hash,
                                    start, pivot)
```

Um im vorderen Bereich zu suchen, hat die Funktion `get_full_line()` mit `pivot` den passenden Endwert für den Bereich geliefert. Für den hinteren Bereich ergibt es aber Sinn, hinter der gerade gesuchten Zeile zu suchen. An dieser Position steht der Dateizeiger nach `readline()`. Die Funktion `file.tell()` gibt diese Position als Zahlenwert zurück.

In beiden Fällen darf die Suche auch enden, wenn der Bereich keine Zeile mehr enthält. Das übernehmen die ersten beiden Zeilen der Funktion. Der Bereich hat nur dann eine Größe von 0, wenn der Hash nicht in der Datei steht, weshalb die Funktion dann 0 zurückgibt.

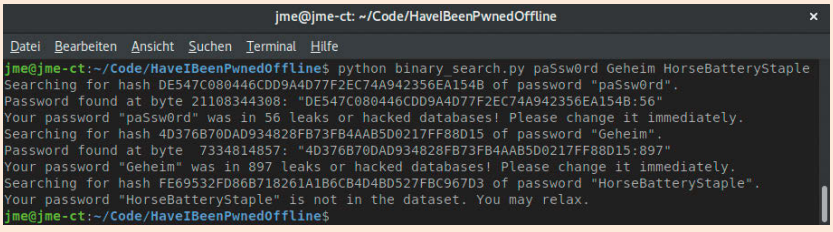
Die Unterfunktionen `get_full_line()` und `search_hash()` übernehmen die ganze Arbeit. Für die umschließende Funktion `binary_search()` reicht es daher, in einer Zeile die Rekursion mit einem Bereich von 0 bis zur Größe der Passwortdatei anzustoßen:

```
return search_hash(list_file,
                               hex_hash, 0, file_size)
```

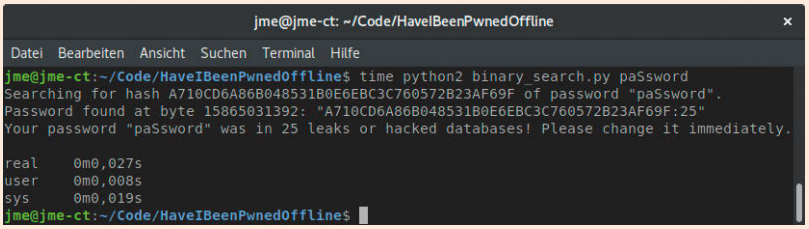
Damit ist der Algorithmus schon fertig. Das Programm umschifft aber zusätzlich noch ein paar Alltagsprobleme, damit man es einfach auf der Kommandozeile benutzen kann.

Encoding und Hashing

Hashfunktionen wie SHA-1 arbeiten mit Bytes statt Strings. Es spielt also eine Rolle, mit welchem Encoding die Hashes erstellt wurden. Troy Hunt, der Ersteller der Liste bei HaveIBeenPwned, erklärt in einem Kommentar zu seinem Blogpost zur Erstellung der Liste (siehe ct.de/w2pk), dass er die Pass-



Zwei der gesuchten Passwörter stehen in der Datenbank. Die sollte niemand benutzen. Das dritte Passwort kam bisher in keinem Leak im Klartext vor.



Die binäre Suche braucht nur 0,027 Sekunden, um ein Passwort zu finden.

wörter in UTF-8 kodiert hat, bevor er sie gehasht hat. Daher kodiert das Python-Skript ebenfalls alle zu testenden Passwörter in UTF-8.

Den Hash berechnet die Funktion `sha1()` aus der Hashlib, die das Programm ganz zu Beginn mit `from hashlib import sha1` importiert. Die Funktion gibt ein Objekt zurück, dem man mit der Methode `hexdigest()` einen Hash in Hex-Darstellung entlockt. Da der aber Kleinbuchstaben verwendet, die Liste aber Großbuchstaben enthält, muss die String-Funktion `upper()` noch alle Klein- in Großbuchstaben umwandeln:

```
if 'decode' in dir(str):
  password = password.decode('utf-8')
h = sha1(password.encode('utf-8')
         ).hexdigest().upper()
```

Die `if`-Anweisung in den ersten beiden Zeilen dient der Kompatibilität mit Python 2.7. Dort sind Strings nicht wie in Python 3 grundsätzlich als UTF-8 kodiert, sodass der Code sie für Python 2.7 noch dekodieren muss. Da Strings nur im alten Python eine `decode()`-Funktion besitzen, wird dieser Schritt bei Python 3 übersprungen.

Argumente parsen

Der übrige Code dient dazu, das Programm auf der Konsole leicht nutzbar zu machen. Dafür sorgt der `ArgumentParser` (`from argparse import ArgumentParser`), der Kommandozeilenargumente strukturiert entgegennimmt und automatisch eine Hilfenachricht generiert (Aufruf mit der Option `--help`):

```
parser = ArgumentParser(description=
           'Test passwords locally.')
parser.add_argument('passwords',
                    nargs='+')
parser.add_argument('--pwned-pass' +
  'words-ordered-by-hash-filename',
  required=False, default="pwned-" +
          "passwords-sha1-ordered" +
                "-by-hash-v5.txt")
args = parser.parse_args()
```

Die Methode `add_argument()` fügt jeweils einen Kommandozeilenparameter hinzu. Da der Name `passwords` nicht mit einem Minus (`-`) beginnt, muss man für diesen Parameter beim Aufruf kein Präfix angeben. Die Option `nargs='+'` legt fest, dass es sich bei diesem Parameter um eine Liste mit mindestens einem Element handelt. Dank dieser Definition sorgt der Parser dafür, dass man mindestens ein Passwort hinter dem Dateinamen angeben muss, jedoch auch gleich mehrere per Leerzeichen getrennt angeben darf.

Das zweite Argument funktioniert nur mit Präfix. Da es aber als `required=False` erstellt wird, muss man es nicht angeben. Die Option `default` sorgt dafür, dass der Parser für das Argument auch dann einen Wert liefert, wenn es der Nutzer weggelassen hat. Ein Aufruf des Programms mit allen Argumenten sieht beispielsweise so aus:

```
python binary_search.py paSswOrd ↵
 ↳Geheim HorseBatteryStaple --pwned-↵
 ↳passwords-ordered-by-hash-filename ↵
 ↳pwned-passwords-sha1-ordered-by-↵
 ↳hash-v5.txt
```

Die Reihenfolge der Argumente spielt praktischerweise keine Rolle.

Nach dem Aufruf von `parse_args()` stehen die Parameter als Properties zur Verfügung. Das Skript nutzt sie danach, um die Passwort-Datei zu öffnen und mit `stat()` aus dem `os`-Modul (`from os import stat`) die Dateigröße auszulesen:

```
with open(args.pwned_passwords_↵
        ↳ordered_by_hash_filename,
        'r') as pwned_passwords_file:
  pwned_passwords_file_size = stat(
     args.pwned_passwords_ordered_by_↵
              ↳hash_filename).st_size
```

In der Schleife `for password in args.passwords:` sucht das Skript danach nach jedem übergebenen Passwort und gibt die Ergebnisse aus (die `print()`-Befehle haben wir hier ausgelassen).

Das ganze Skript hat nur 57 Zeilen und steht zur Begutachtung und zum Download auf GitHub (siehe ct.de/w2pk). Es macht keine Netzwerkanfragen, was Sie bereits an den Imports ganz zuoberst erkennen können. Es schreibt die eingegebenen Passwörter auch nicht auf die Festplatte, was Sie leicht an einem `open()` mit der Option `w` erkennen würden.

Das Kommandozeilen-Interface erlaubt es Ihnen, bequem und ohne Netz ein oder mehrere Passwörter zu prüfen. Durch die schnelle binäre Suche geht das auch mit einem Dutzend verschiedener Passwörter in wenigen Millisekunden. Falls es Ihnen unangenehm ist, Passwörter in der Bash-History wiederzufinden, müssen Sie diese deaktivieren oder löschen. Alternativ bauen Sie die `getpass()`-Funktion aus dem `getpass`-Modul ins Skript ein. (pmk) c't

Das Programm bei GitHub:
www.ct.de/w2pk

Genom-Datamining mit Pandas

Wer sein Genom sequenzieren lässt, bekommt die Rohdaten als CSV-Datei mit hunderttausenden Zeilen. Das Python-Framework Pandas beweist sich gerade dann als Schweizer Messer der Datenanalyse, wenn die Tabellen wie bei Gendaten zu groß für grafische Tabellenkalkulationen wie Excel werden.

Von Pina Merkert

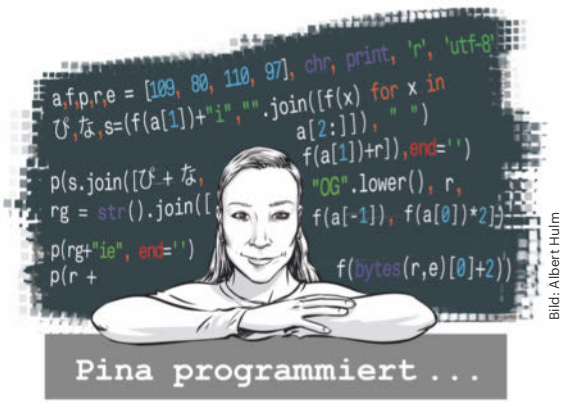


Bild: Albert Hulm

Sequenzierungsdienste fürs eigene Genom helfen bei der Ahnenforschung und das Risiko für manche erbliche Krankheiten zu schätzen. Neben hübsch aufbereiteten Zusammenfassungen liefern die Anbieter auch Rohdaten, die sie als CSV-Dateien von circa 20 Megabyte Größe verschicken. So große Dateien verarbeiten Excel, LibreOffice und Konsorten nicht mehr in erträglicher Geschwindigkeit. Das Python-Framework Pandas dagegen setzt unter der Haube auf die effizienten Datenmodelle von Numpy und analysiert Tabellen dieser Größe daher in Sekundenbruchteilen.

Zwecks Analyse ihrer DNA haben c't-Redakteure Proben an mehrere Ahnenforschungs-Plattformen geschickt [1] und mir von jedem Anbieter einen Satz mit Rohdaten zur Auswertung ausgehändigt. Die Dateien nutzen leicht unterschiedliche Formate, mal kommasepariert, mal mit Tabs, mal mit getrennt gelisteten Allelen (eine Base von den Genen der Mutter, eine von denen des Vaters), mal mit beiden Basen als String aus zwei Großbuchstaben. Die Daten enthalten kein vollständiges Genom, sondern Zeilen mit SNPs, also den Basen interessanter Mutationen. Die Anbieter ordnen jedem SNP eine Bedeutung beispielsweise für ein Krankheitsrisiko zu. Zu einem Identifier für das SNP wie „rs4475691“ steht in den Daten jeweils die Position im Genom als Zahl (846808) und das Chromosom, in dem das Basenpaar vorkommt (Nummer 1). Damit Sie den in diesem Artikel erklärten Pandas-Code nachvollziehen können, ohne gleich Ihr Genom analysieren zu lassen, finden Sie auf GitHub CSV-Dateien im gleichen Format, aber mit künstlich erzeugten, zufälligen Angaben zu den Basen (siehe ct.de/whbb).

Experimentierumgebung

Mit einem Jupyter-Notebook, ein Python-Interpreter im Browser, geht die Genanalyse leicht von der Hand, da es Pandas-Dataframes, die Tabellenobjekte des Frameworks, grafisch als Tabellen aufbereitet. Eine virtuelle Umgebung mit Jupyter und Pandas ist schnell eingerichtet, zum Beispiel unter Linux mit folgenden Kommandozeilenbefehlen:

```
mkdir PandasGenom
cd PandasGenom/
python3 -m venv env
source env/bin/activate
```

```
pip install pandas jupyter
jupyter notebook
```

Der letzte Befehl öffnet ein Browserfenster mit einem Knopf zum Anlegen neuer Notebooks. Jupyter-Notebooks sind Textdokumente, in denen man neben Text auch in „Zellen“ Code schreiben und mit Umschalt+Enter ausführen kann. Gibt der letzte Befehl einer Zelle etwas Darstellbares zurück, zeigt Jupyter diesen Wert unter der Zelle an. Steht beispielsweise in der letzten Zeile einer Zelle ein Pandas-Dataframe als Variable, zeigt Jupyter die Tabelle gekürzt an (die fünf ersten und letzten Zeilen).

CSVs einlesen

Die ersten Zeilen der ersten Zelle laden wie in Python-Skripten üblich per `import` die Bibliothek:

```
import pandas as pd
import os
```

Das `os`-Modul dient lediglich dazu, den Pfad der Dateien zusammenzubauen:

```
ingo_my_heritage= pd.read_csv(
  os.path.join("FakeGenome",
               "MyHeritage_raw_dna_data_fake.csv"),
  header=12,
  names=["rsid",
         "chromosome",
         "position",
         "result"
  ],
  dtype={"rsid": str,
         "chromosome": str,
         "position": int,
         "result": str})
```

Ums Einlesen kümmert sich `pd.read_csv()`. Der Parameter `header` teilt der Funktion mit, in welcher Zeile der Datei die Spaltenbeschriftungen stehen. Ein Nebeneffekt dieser Angabe ist, dass Pandas alle Zeilen davor ignoriert, was effektiv den Header abschneidet. Die Spaltennamen kann man nämlich mit einer Liste an `names` auch per Hand festlegen, falls die Datei keine zufriedenstellenden Spaltennamen liefert. Pandas bestimmt die Datentypen der Spalten normalerweise automatisch, arbeitet aber schneller, wenn man die Typen als Dictionary im Parameter `dtype` definiert. Die Keys im Dictionary entsprechen den bei `names` festgelegten Spaltennamen.

Die Daten von My Heritage fassen die Allele 1 und 2 im Schlüssel `result` zusammen. Um sie in einzelne Spalten aufzuteilen, schneidet der folgende Code die Allele als Slice heraus:

```
my_heritage["allele1"] = ↵
      ↳my_heritage["result"].str[:1]
my_heritage["allele2"] = ↵
      ↳my_heritage["result"].str[1:]
```

My Heritage hat praktischerweise die Werte mit Komma getrennt, was `read_csv()` als Standard annimmt. Für die tabulatorseparierten Daten von Ancestry muss man den Tab als Separator angeben: `sep="\t"`. Dafür enthält diese Datei eine schöne `header`-Zeile, was die Angabe der `names` spart:

```
ancestry = pd.read_csv(
  os.path.join("FakeGenome",
               "AncestryDNA_fake.txt"),
  sep="\t", header=18, dtype={
    "rsid": str,
    "chromosome": str,
    "position": int,
    "allele1": str, "allele2": str})
```

Um auf die gleichen Spalten wie bei My Heritage zu kommen, braucht die Ancestry-Tabelle noch ein „result“ kombiniert aus beiden Allelen:

```
ancestry["result"] = ↵
      ↳ancestry["allele1"] + ↵
      ↳ancestry["allele2"]
```

Danach liegen beide Tabellen als `Dataframe`-Objekte mit den gleichen Spalten vor. Die Auswertung kann beginnen.

Interessante Stellen

Jeder Anbieter scheint eine eigene Vorstellung davon zu haben, welche SNPs interessant sind (siehe Spalte „rsid“). Ich habe mich gefragt, wie viel Überschneidung es dabei gibt. Pandas beantwortet diese Frage mit der Funktion `isin()`. Sie berechnet eine ganze Spalte an Wahrheitswerten. Ein `True` steht nur in den Zeilen, bei denen der Wert auch in der angegebenen Spalte vorkommt. `value_counts()` zählt anschließend, wie häufig `True` und `False` in der Spalte vorkommen und listet die Zahlen auf:

```
my_heritage["rsid"].isin(
     ancestry["rsid"]).value_counts()
```

Ergebnis: Die Anbieter sind sich nur bei einem Drittel der SNPs einig, dass diese interessant sind.

405.910 der von My Heritage ausgewählten SNPs kommen bei Ancestry gar nicht vor.

Eine Liste aller Zeilen, deren SNPs nur My Heritage misst, liefert die folgende Zeile:

```
my_heritage[~my_heritage["rsid"].↵
                        ↳isin(ancestry["rsid"])]
```

Sie verwendet die zuvor berechnete Spalte als Auswahlkriterium, allerdings kehrt die vorangestellte Tilde ~ den Wahrheitswert um. Übrig bleiben so nur die einzigartigen Zeilen.

Prüfvereinigung

Ein Kollege hat neben den Daten von Ancestry und My Heritage auch einen Datensatz von 23 And Me beigesteuert. Für die 171.012 SNPs, die alle drei Anbieter gemessen haben, wollte ich wissen, wie viele Unterschiede es gibt, da das auf Messfehler oder Mutationen der gemessenen Zelle hinweisen würde.

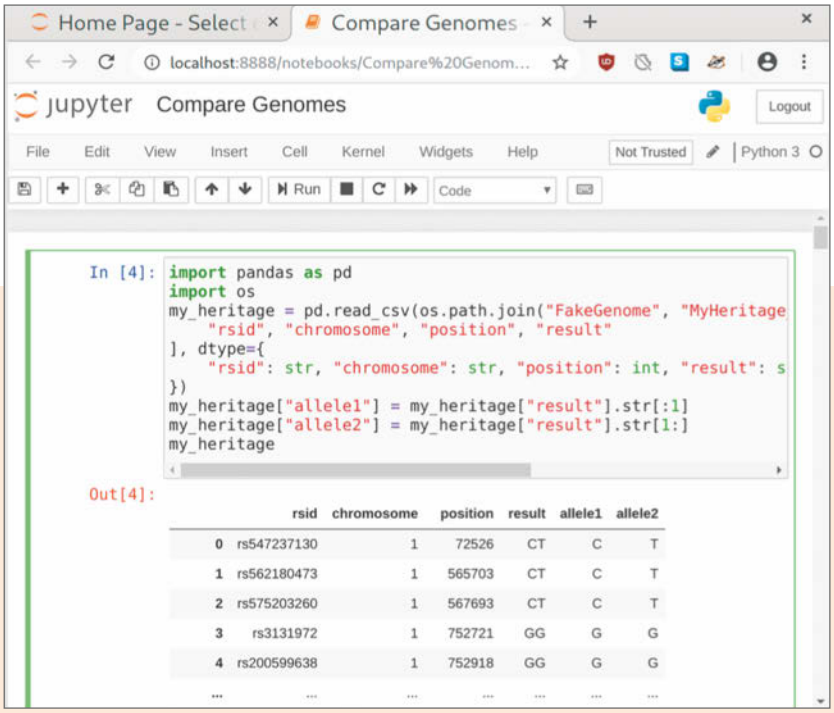
Für den Vergleich müssen die Daten aus den drei Dataframes in einen einzelnen zusammenfließen. Mit einem Index kann Pandas zuordnen, welche Zeilen zusammengehören und wo es fehlende Werte mit `None` belegen muss. `set_index()` markiert eine Spalte als Index, wodurch ein `join()` möglich wird:

```
rsid_indexed_my_heritage = ↵
       ↳my_heritage.set_index("rsid")
rsid_indexed_ancestry = ↵
         ↳ancestry.set_index("rsid")
rsid_indexed_23andme = ↵
       ↳fake23andme.set_index("rsid")
```

Danach fügt Pandas die Tabellen mit `.join()` klaglos zusammen, wobei `lsuffix=_my_heritage` und `rsuffix=_23andme` dafür sorgen, dass Pandas Spalten mit gleichen Namen automatisch umbenennt:

```
joined_df = rsid_indexed_my_heritage.↵
         ↳join(rsid_indexed_23andme, how="inner",
  lsuffix="_my_heritage",
  rsuffix="_23andme").join(
  rsid_indexed_ancestry, how="inner")
```

Die Option `how` wählt zwischen den vier von SQL bekannten Join-Typen `inner`, `left`, `right` und `outer`. Für den Vergleich der Messwerte sind nur Zeilen interessant, die es bei beiden Anbietern gibt, wes-



	rsid	chromosome	position	result	allele1	allele2
0	rs547237130	1	72526	CT	C	T
1	rs562180473	1	565703	CT	C	T
2	rs575203260	1	567693	CT	C	T
3	rs3131972	1	752721	GG	G	G
4	rs200599638	1	752918	GG	G	G
...	...	...	...	...	...	...

In Jupyter-Notebooks programmiert man Python im Browser. Pandas Dataframes zeigt Jupyter praktischerweise als gekürzte Tabelle an.

halb ein Inner-Join gern alle Zeilen verwerfen darf, die nur einer der Anbieter kennt.

Den so erweiterten Dataframe ergänzt ein weiterer Join um die Daten von Ancestry. Dabei ist kein Suffix nötig, da sich keine Spaltennamen doppeln. Damit erkennbar bleibt, dass die ergänzten Daten von Ancestry stammen, benennt ein `.rename()` die Spaltennamen um:

```
joined_df = joined_df.rename(columns={
  "chromosome": "chromosome_ancestry",
  "position": "position_ancestry",
  "result": "result_ancestry",
  "allele1": "allele1_ancestry",
  "allele2": "allele2_ancestry"
})
```

Ein Problem beim Vergleich ergibt sich daraus, dass die Anbieter die beiden Allele unterschiedlich sortieren. Mal stehen sie alphabetisch sortiert mit `A` links, mal umgekehrt. Eine `for`-Schleife über die drei Anbieter sortiert die Basen ruckzuck per `lambda`-Funktion in der `result`-Spalte und ergänzt beim Y-Chromosom, das es nur einmal geben kann, kurzerhand den zweiten Wert durch Kopieren des ersten:

```
for tester in ["_my_heritage", "_ancestry",
                                        "_23andme"]:
  joined_df["result" + tester] =
    joined_df["allele1" + tester] +
    joined_df["allele2" + tester]
  ).apply(lambda x: x
    if len(x) < 2 or x < x[1] + x[0]
    else x[1] + x[0])
  joined_df["result" + tester] = ↵
↳joined_df["result" + tester].apply(
        lambda x: x + x if len(x) < 2
                        else x)
```

Mit dieser Vorarbeit besteht der Test nur noch aus drei einzelnen Vergleichen mit `==`, die Pandas aber mit `&` statt wie bei Python sonst üblich mit `and` verknüpft haben möchte:

```
joined_df["allele_match"] = (
  (joined_df["result_ancestry"] == ↵
↳joined_df["result_23andme"]) &
  (joined_df["result_ancestry"] == ↵
↳joined_df["result_my_heritage"]) &
  (joined_df["result_23andme"] == ↵
↳joined_df["result_my_heritage"]))
```

Das Histogramm (`value_counts()` für die Spalte `allele_match`) bescheinigt anschließend bei den zufälligen Daten überwiegend Abweichungen. Im echten Datensatz erlaubten sich die Anbieter jedoch keinen einzigen Fehler.

Daten gebändigt

Das Beispiel zeigt anschaulich, wie die Arbeit mit Pandas aussehen kann. Weitere Inspirationen liefert das Jupyter-Notebook im Git-Repository zum Artikel. Dort finden Sie auch den Code zum Erzeugen der Zufallsdatensätze.

Pandas erweist sich gerade dann als nützlich, wenn die Tabellen wie bei den analysierten Gendaten zu groß für grafische Tabellenkalkulationen werden. Dazu kommt, dass mit Pandas-Dataframes noch einiges mehr möglich ist, beispielsweise schicke Diagramme mit dem Plotting-Framework Altair (siehe Seite 154). (pmk) c't

Literatur

[1] Arne Grävemeyer, Jan-Keno Janssen, **Nicht nur für Ahnenforscher: DNA gibt persönliche Details preis,** Das steckt in mir, c't 5/2020, S. 24

Beispielcode als Jupyter-Notebook

www.ct.de/whbb

```
In [7]: my_heritage["rsid"].isin(ancestry["rsid"]).value_counts()
Out[7]: False    405910
        True     203436
        Name: rsid, dtype: int64
```

Die Funktion value_counts() berechnet ein Histogramme der Werte in einer Spalte. Hier zeigt sie, welche SNPs sowohl My Heritage als auch Ancestry messen.

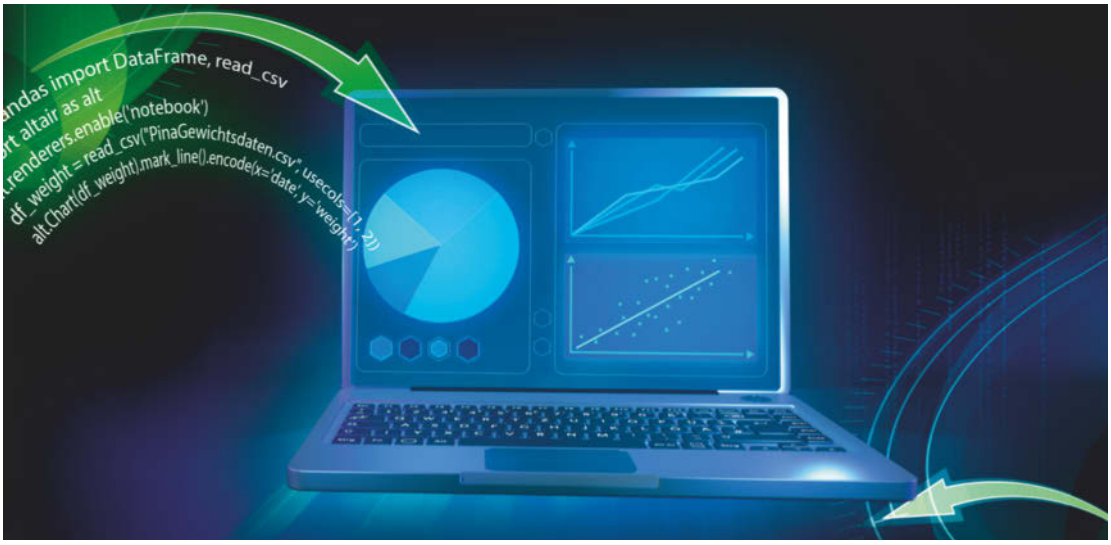


Diagramme mit Altair

Mit etwas Übung schreibt man Code schneller, als man klickt. Mit den Frameworks Pandas und Altair bereitet man in einer Handvoll Python-Zeilen Datensätze auf und erzeugt schicke Diagramme. Altair basiert auf dem JavaScript-Framework Vega, sodass sich die Plots mühelos im Web einbinden lassen – auch als interaktive Grafiken.

Von Pina Merkert

Ein Klick auf den Download-Link befördert eine 25 Megabyte große CSV-Datei ins Download-Verzeichnis. Ein Doppelklick darauf startet LibreOffice und danach heißt es warten. Der Import dauert nämlich auf einem nicht ganz taufrischen i5 über 30 Sekunden. Als die Riesen-Tabelle endlich erscheint, ungeduldig die Formel für den Durchschnitt in die freie Spalte ganz rechts tippen und anschließend eine Minute lang scrollen, um die Formel auf alle Zeilen zu übertragen. Das fühlt sich alles ziemlich zäh an. Führt man sich vor Augen, dass Office die Daten alle in sein Interface rendern muss, fällt auf: Das Programm ist eigentlich nicht langsam. Es ist nur nicht für so große Tabellen gemacht.

Damit das alles flotter geht, muss das grafische Interface weichen. Stattdessen kommt Python zum Einsatz, genauer die Bibliothek Pandas. Pandas nutzt unter der Haube Numpy und damit schnellen C-Code, um Arrays effizient zu speichern. Statt in `numpy.ndarray` (ohne Spaltennamen) landet alles in Objekten vom Typ `pandas.DataFrame`. So ein `DataFrame` ist gewissermaßen ein Tabellenblatt für Programmierer, das die gleichen Funktionen wie Excel bereitstellt. Für Numpy-Veteranen bekannt, für

Python eher ungewöhnlich: Die Daten in `DataFrames` sind hart typisiert. Wenn Sie ein DataFrame anlegen, müssen Sie daher gleich festlegen, ob dort Gleitkommazahlen, Ganzzahlen, Strings oder Zeitangaben in den Spalten stehen. Außerdem sind Spalten generell benannt, was in großen Tabellen für Übersicht sorgt.

Das Framework Altair zeichnet aus einem DataFrame mit einer Handvoll Zeilen ein hübsches Diagramm. Es bietet dafür eine kürzere und logischere Syntax als andere Python-Plotting-Bibliotheken wie Matplotlib. Altair erfindet das Rad nicht neu, sondern setzt intern auf der JavaScript-Plotting-Bibliothek Vega (oder Vega-Lite) auf, weshalb das Framework ohne Mehraufwand neben PNGs und SVGs auch Webseiten exportiert. Altairs integrierte Web-Affinität nutzt man am bequemsten, indem man den gesamten Python-Code gleich in einem Jupyter-Notebook schreibt, dort erscheinen die Diagramme direkt in der Weboberfläche des Notebook.

Damit die Datenanalyse mit Python Spaß macht, brauchen Sie also Pandas, Altair, Vega und Jupyter. Wie Sie diese Pakete am leichtesten beschaffen, hängt vom Betriebssystem ab.

Installation: Anaconda

Unter Windows haben wir gute Erfahrungen mit der Python-Distribution Anaconda gemacht, deren Installer Sie unter `www.anaconda.com/distribution/` herunterladen. Anaconda arbeitet mit „Umgebungen“, die die Abhängigkeiten verschiedener Python-Projekte voneinander abschotten. Lassen Sie den Installer ruhig eine Default-Umgebung einrichten. Die trägt alle nötigen Pfade in Umgebungsvariablen ein. Anaconda aktiviert die Umgebung auch gleich automatisch in allen neu gestarteten Konsolenfenstern. Wenn Sie das stört, schalten Sie es einfach nach der Installation mit folgendem Befehl ab:

```
conda config --set auto_activate_base false
```

Wenn Sie nun eine getrennte Umgebung namens „datavis“ für Ihr Projekt einrichten wollen, geht das mit folgendem Befehl:

```
conda create -n datavis python=3.8
```

Nutzen Sie auf jeden Fall Python 3 und nicht die veraltete Version 2.7. Falls Ihre Shell den Befehl `conda` nicht findet, geben Sie den ganzen Pfad an, beispielsweise:

```
~/anaconda3/bin/conda create ↵
                ↳-n datavis python=3.8
```

Unter Windows erzeugen und aktivieren Sie die Umgebungen über ein Menü. Linux-Nutzer von Anaconda aktivieren die Umgebung mit folgendem Befehl:

```
conda activate datavis
```

Die Bibliotheken installieren dann folgende Befehle auf der Konsole (unter Windows starten Sie dafür innerhalb der Umgebung eine Anaconda-Konsole):

```
conda install numpy pandas vega altair jupyter
```

Installation: Pip

Linuxer sparen ein paar Befehle mit der Python-Umgebung des Systems. Virtualenv erzeugt eine virtuelle Umgebung:

```
mkdir datavis && cd datavis
python3 -m venv env
source env/bin/activate
```

Pip installiert die Bibliotheken danach mit folgenden Befehlen:

```
pip install wheel numpy pandas vega altair jupyter
```

Wie gebe ich Geld aus?

Mit der perfekt vorbereiteten Arbeitsumgebung kann die Analyse losgehen. Starten Sie dafür den lokalen Webserver des Jupyter-Notebooks:

```
jupyter notebook
```

Der Befehl öffnet auch gleich ein Browser-Fenster mit der URL `127.0.0.1:8888/tree`, das eine Übersicht anzeigt. Mit dem Menü unter „New“ (oben rechts) starten Sie ein neues Notebook für Python3.

In dessen erste Zelle importieren Sie Pandas, Numpy und Altair:

```
import pandas as pd
import numpy as np
import altair as alt
```

Den Code einer Zelle im Jupyter-Notebook führen Sie mit Shift+Enter aus.

Nun gilt es, einen Datensatz zu besorgen. Für einen schnellen Start habe ich eine CSV-Datei aus meinem Online-Banking exportiert und jeder meiner Ausgaben über ein Jahr eine Kategorie zuge-

ordnet. `bankdaten.csv` finden Sie im Git-Repository zu diesem Artikel über ct.de/wbxj. Die Datei enthält die Spalten „Buchungstag“, „Betrag“ und „Kategorie“ und alle 627 Ausgaben vom 1. Juni 2018 bis 30. Juni 2019. Falls Sie lieber eigene Bankdaten analysieren möchten, passen Sie `categorize_bank_data.py` aus dem Repository an Ihre Bedürfnisse an.

Altair erwartet als Datenquelle immer ein Pandas-`DataFrame`. Pandas bringt eine mächtige Import-Funktion für CSV-Dateien mit, der Sie lediglich den Dateinamen mitteilen müssen:

```
df = pd.read_csv("bankdaten.csv")
```

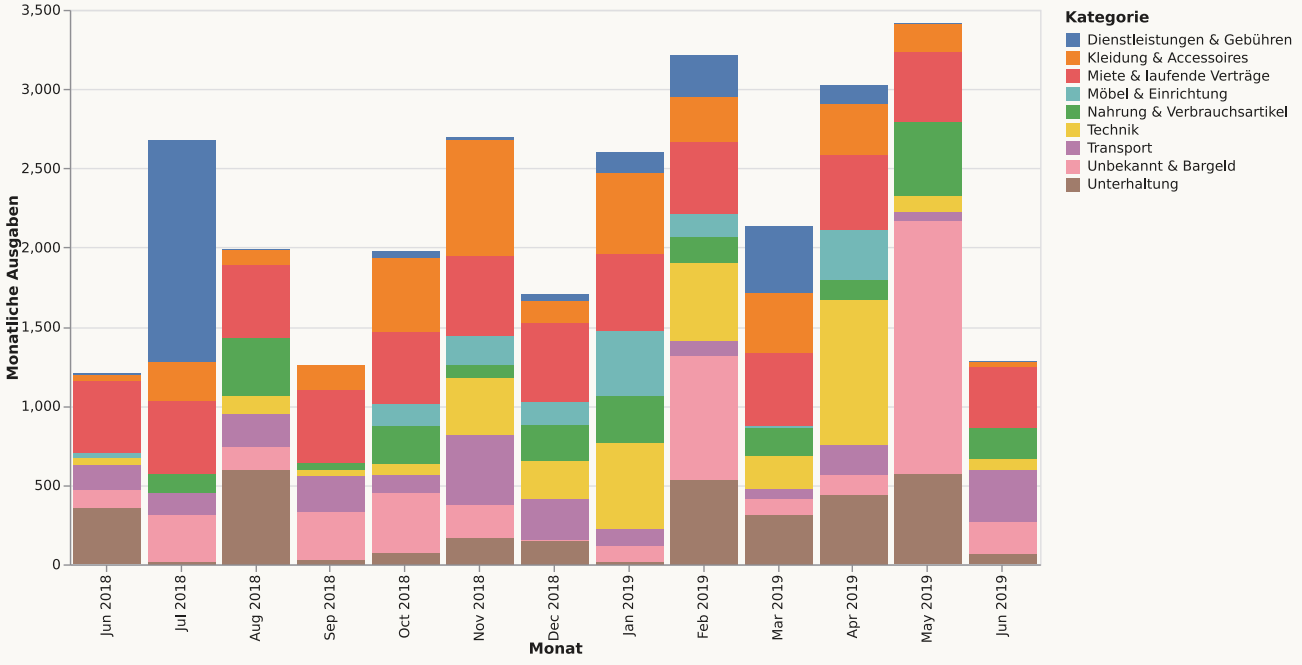
Wenn Sie die Tabelle nun mit `df` anzeigen (Jupyter rendert sie automatisch als in der Mitte gekürzte HTML-Tabelle) sehen Sie ganz links eine Spalte mit dem von Pandas automatisch erzeugten Index der Zeilen und daneben einen identischen Index, der in der ersten Spalte der CSV-Datei steht. Um den wegzulassen, müssen Sie nur die gewünschten Spalten beim Import angeben:

```
df = pd.read_csv("bankdaten.csv",
                 usecols=[1, 2, 3])
```

`df`, das `DataFrame`, ist nun bereit, um damit ein Diagramm zu zeichnen. Dafür frisst zuerst `alt.Chart()` den Datensatz. Das Objekt bringt die Methode `mark_bar()` mit, um intern auf ein Balkendiagramm umzustellen. Welche Spalten zu welchen Achsen des Diagramms gehören, legt anschließend die

Wofür gebe ich Geld aus?

Die Höhe der farbigen Balken zeigt die Summe der Ausgaben einer Kategorie. Die Höhe des Turms dieser Balken gibt die Gesamtsumme für den Monat wieder. Im Juli gut erkennbar sind die circa 1200 Euro, die ich für meine Personenstandsänderung bezahlen musste („Gebühr“). Die „Unterhaltung“ und das „Bargeld“ im Februar gehören zu einem Skiurlaub. Mit der „unbekannten“ Ausgabe im März habe ich meine Debitkarte für eine Dienstreise in die USA befüllt.



Methode `encode()` fest. Ihr gibt man die Spalten als Parameter mit. In Kombination sieht das folgendermaßen aus:

```
alt.Chart(df).mark_bar().encode(
          x="Buchungstag", y="Betrag")
```

Dabei entsteht ein enorm breites Balkendiagramm mit einem Balken pro Zeile im `DataFrame`. Darin sieht man leicht Ausreißer, die Übersicht geht aber verloren.

Das Diagramm spart viel Platz, wenn es die Ausgaben pro Monat nur als Summe darstellt. Für solche Nöte beim Datenauswerten bringt Altair Funktionen mit, sodass man das `DataFrame` nicht anpassen muss:

```
alt.Chart(df).mark_bar().encode(
         x="yearmonth(Buchungstag):O",
         y="sum(Betrag)")
```

Die Funktion `yearmonth()` fasst auf der x-Achse die Datumsangaben in der Spalte „Buchungstag“ zu Monaten zusammen. Obwohl Altair die Datumsangaben dadurch auf Monate rundet, geht es trotzdem von Tagen als kontinuierlicher Basisgröße aus. Das produziert sehr schmale Balken mit großen Abständen. Das Angehängte `:O` teilt Altair mit, dass es die Monate stattdessen als aufzählbare Größe behandeln soll, was zu sinnvoll breiten Balken führt.

Die `sum()`-Funktion bei der y-Achse addiert alle Ausgaben, die auf der X-Achse denselben Wert haben – was hier zur Summe der Ausgaben in einem Monat führt. Das entstehende Diagramm zeigt mit seinen sehr unterschiedlich hohen blauen Balken, dass ich je nach Monat sehr unterschiedlich viel Geld ausgebe.

Die Achsenbeschriftung lässt etwas zu wünschen übrig, da sie die Funktionen berücksichtigt und den deutschen Spaltennamen automatisch in Englisch ergänzt. Um die Achsenbeschriftung per Hand zu setzen, bringt Altair die Klassen `X` und `Y` mit:

```
alt.Chart(df).mark_bar().encode(
   x=alt.X("yearmonth(Buchungstag):O",
       title="Monat"),
   y=alt.Y("sum(Betrag)",
       title="Monatliche Ausgaben"))
```

Leider zeigt das blaue Diagramm nun aber nicht, aus welcher Kategorie die hohen Ausgaben stammen. Das könnte man gut sehen, wenn jede Kategorie einen eigenen Balken in einer eigenen Farbe pro Monat hätte und sich diese Balken so stapeln, dass die Höhe des Turms wieder die Gesamtsumme zeigt. Das klingt kompliziert? Mit Altair genügt dafür ein zusätzlicher Parameter:

```
alt.Chart(df).mark_bar().encode(
   x=alt.X("yearmonth(Buchungstag):O",
       title="Monat"),
   y=alt.Y("sum(Betrag)",
       title="Monatliche Ausgaben"))
   color="Kategorie").properties(
   width=700, heigt=400)
```

Die zuletzt angehängte `properties()`-Funktion legt zusätzlich nur noch die Größe des Diagramms fest.

Folgt das Gewicht dem Essen?

Balkendiagramme sind mit Altair also einfach. Wie sieht es aber mit Diagrammen aus, an denen man erkennt, dass zwei Größen korrelieren?

Ein persönliches Beispiel: Ich habe über viele Monate mit mehreren Apps protokolliert, wie viele Kalorien ich pro Tag gegessen und getrunken habe (`PinaKaloriendaten.csv`). Außerdem habe ich eine smarte Waage, auf der ich mich jeden Morgen unter vergleichbaren Bedingungen gewogen habe (`PinaGewichtsdaten.csv`). Man vermutet: Wenn man an einem Tag viel isst, bringt man am nächsten Tag entsprechend mehr Gewicht auf die Waage. Übt man sich dagegen im Verzicht, erwartet man am nächsten Morgen ein geringeres Gewicht. Die Gewichtskurve und die Kalorienkurve sollten dabei also etwa die gleiche Form zeigen.

Die beiden CSV-Dateien sind nicht für den Import in Pandas vorbereitet. Sie entsprechen damit eher dem, was Ihnen im Alltag als Datenquellen begegnen wird. Meist gilt es nach dem Import noch Kleinigkeiten anzupassen. Beispielsweise enthalten die CSVs Datumsangaben im deutschen Format mit Punkt als Trennzeichen. Das erkennt Pandas nicht automatisch als Datentyp `Datetime`. Das Framework bringt aber die Funktion `to_datetime()` mit, der man einen Format-String mitgeben kann:

```
df_weight = read_csv(
            "PinaGewichtsdaten.csv",
            usecols=[1, 2])
df_weight["date"] = to_datetime(
        df_weight["date"],
        format="%Y-%m-%d %H:%M:%S.%f")
df_nutri = read_csv(
```

```
            "PinaKaloriendaten.csv",
            usecols=[0, 1, 2, 3],
            dtype={"Tag": str,
                   "Noom": float,
                   "Lose It!": float,
                   "Yazio": float})
df_nutri.rename(index=str,
            columns={"Tag": "date"},
            inplace=True)
df_nutri["date"] = to_datetime(
        df_nutri["date"],
       format="%d.%m.%y")
```

Beim Import von Zahlen ist es außerdem sinnvoll, `read_csv()` mit dem Parameter `dtype=` mitzuteilen, wenn die Funktion auch dann Gleitkommazahlen laden soll, wenn sie in der CSV-Datei ohne Komma stehen. Die Funktion `.rename()` benennt Spalten um, indem man ihr ein Dictionary mit den alten und neuen Namen übergibt.

Ich habe meine Kalorien mit verschiedenen Apps protokolliert, da diese jeweils unterschiedliche Datenbanken mit Nahrungsmitteln benutzen. 170 Gramm Apfel haben dann mal 89 kcal in der einen App und 111 kcal in der anderen App. Eine neue Spalte mit dem Durchschnitt berechnet Pandas mit `.mean()`:

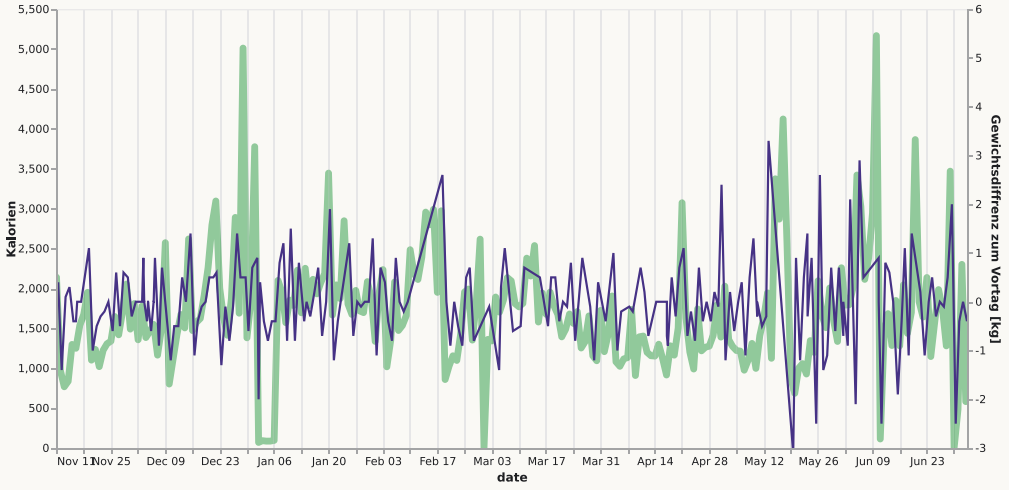
```
df_nutri["Kalorien"] = df_nutri[
        ["Noom", "Loose It!", "Yazio"]
        ].mean(axis=1, skipna=True)
```

Der Befehl filtert dafür zuerst die richtigen Spalten mit der Angabe in eckigen Klammern heraus. Dabei gibt man eine Liste an Spalten an, weshalb in der eckigen Klammer eine eckige Klammer steht.

Der Parameter `skipna=True` in der `.mean()`-Funktion sogt dafür, dass Pandas Zeilen mit undefinierten Werten verwirft. Undefinierte `NaN`-Werte ent-

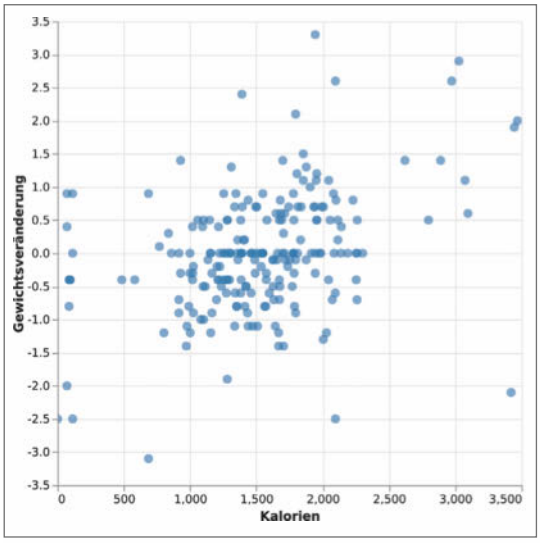
Gewicht folgt Konsum?

„Übt man sich im Verzicht, erwartet man am nächsten Morgen ein geringeres Gewicht.“ – Diese Aussage kann das Diagramm mit meinen Daten nicht bestätigen.



Die Kurven für Kalorien und Gewicht liegen nicht übereinander. Das deutet darauf hin, dass die beiden Werte nicht stark korrelieren.

Die Punkte im Streudiagramm ordnen sich nicht in einer Diagonalen an. Das heißt, dass es keine nennenswerte Korrelation zwischen Kalorien und Gewichtsveränderung gibt.



stehen beispielsweise, wenn in der CSV-Datei nichts zwischen den Trennzeichen steht.

Statt des Durchschnitts berechnet `.var()` die Varianz der Werte in den drei Spalten. Für die übliche Darstellung als Standardabweichung müssen Sie dabei noch die Wurzel ziehen, wofür Sie problemlos Numpy einspannen können, da Pandas intern ohnehin auf Numpy aufbaut:

```
df_nutri["var"] = np.sqrt(df_nutri[
        ["Noom", "Loose It!", "Yazio"]
       ].var(axis=1, skipna=True))
```

Die Korrelation zwischen Kalorien und Gewicht erwartet man ja nicht für den gleichen Tag. Nach einem umfangreichen Gelage geht man davon aus, dass die Waage erst am nächsten Tag nach oben ausschlägt. Um das im Diagramm zu sehen, müssen Sie die Kaloriendaten um einen Tag nach vorne verschieben:

```
from datetime import timedelta
df_nutri["date"] = (df_nutri["date"] -
                    timedelta(days=-1))
```

Außerdem erwartetet man, dass die Kalorienkurve nur die Veränderung des Gewichts, nicht den Verlauf von dessen Absolutwerten vorzeichnet. Um die Berechnung der diskreten „Ableitung“ kümmert sich `.diff()`:

```
df_weight["diff"] = df_weight["weight"].diff()
```

An diesem Punkt stehen die Kaloriendaten und die Gewichtsdaten noch getrennt in den beiden DataFrames `df_nutri` und `df_weight`. Die Waage liefert schon länger Daten als die Kalorien-Apps. Deswegen gibt es einige Gewichtsmessungen, zu denen keine Kaloriendaten existieren. Der folgende Befehl verwirft alle Zeilen in der Gewichttabelle, die älter sind als der älteste Kalorienwert:

```
df_weight = df_weight[
                df_weight["date"] >
                df_nutri["date"][-1]]
```

Die Daten für zwei Liniendiagramme stehen danach in zwei DataFrames. Ein Linien- und ein Balkendiagramm unterscheiden sich in Altair hauptsächlich durch den Aufruf von `.mark_line()` statt `.mark_bar()`. In dieser Funktion bestimmt der Parameter `color=`, in welcher Farbe das Framework die Linie zeichnet:

```
chart_weight = alt.Chart(df_weight
             ).mark_line(color="blue"
             ).encode(x='date',
                      y=alt.Y('diff',
 title="Gewichtsdiffrenz zum Vortag",
 scale=alt.Scale(zero=False,
       domain=(-3, 6), type='linear')))
```

Der Parameter `scale=` legt in der `.encode()`-Funktion fest, wie Altair die y-Achse skaliert. Das `Scale`-Objekt setzt dafür in `domain=` den Wertebereich und legt mit `zero=False` fest, dass die Skala nicht bei 0 beginnt. Mit dem Parameter `type=` könnte man hier beispielsweise auch eine logarithmische Skalierung einstellen.

Die Kalorienkurve sieht ähnlich aus, nutzt aber noch eine etwas aufwendigere Formatierung. Die Dicke der Linie soll nämlich einen grafischen Eindruck vermitteln, wie groß die Standardabweichung der Kalorienwerte ist. Die Dicke der Linie orientiert sich dafür an der über alle Zeilen gemittelten Varianz (`df_nutri["var"].mean()`):

```
chart_nutri = alt.Chart(df_nutri
            ).mark_line(color="green",
                 interpolate='linear',
                 shape='stroke',
                 strokeCap='round',
                 strokeJoin='round',
                 strokeOpacity=0.5,
                 strokeWidth=(
     df_nutri["var"].mean() /
```

```
    df_nutri["Kalorien"].max() * 400)
           ).encode(x='date',
                    y='Kalorien'
)
```

Um diese beiden Kurven in einem Diagramm zu sehen, reicht es, die `Chart`-Objekte zu addieren:

```
dbl_chart = chart_nutri + chart_weight
```

Altair versucht dann aber, die y-Achsen beider Diagramme im selben Wertebereich anzuzeigen. Da Kalorien und Gewicht ganz andere Einheiten verwenden, würden die Kurven dann nicht übereinander liegen. Der folgende Befehl schaltet das ab, sodass das Diagramm anschließend zwei y-Achsen enthält: eine links und eine rechts:

```
dbl_chart = dbl_chart.resolve_scale(
                      y='independent')
```

Die Angabe von `.properties(width=830, height=400)` sorgt für ein recht breites Diagramm. Da die CSV-Dateien aber enorm viele Daten enthalten, ist es trotzdem schwierig alle Zacken und Spitzen gut zu erkennen. Abhilfe schafft es da, ein interaktives Diagramm zu erstellen, in dem man mit dem Mausrad zoomen und scrollen kann. In Altair hängt man dafür einfach die Funktion `.interactive(bind_x=True, bind_y=False)` an. Die beiden Parameter legen fest, dass sich die Skalierung der y-Achse beim Scollen nicht verändert, während die x-Achse einen Zoom an bestimmte Daten erlaubt.

Im Diagramm liegen die Kurven nicht sauber übereinander, was sie bei einer starken Korrelation tun müssten. Den Code und ein fertiges Diagramm finden Sie über ct.de/wbxj im Repository auf GitHub in der Datei „Pinas Körperdaten.ipynb“.

Sind Kalorien und Gewicht überhaupt korreliert?

Ob zwei Größen korrelieren, sehen Sie leicht in einem Streudiagramm. Dafür tragen Sie eine Größe auf der x- und die zweite Größe auf der y-Achse auf und zeichnen Punkte in dieses Koordinatensystem. Bei einer starken Korrelation ordnen sich die Punkte entlang einer Diagonalen an, unkorrelierte Größen produzieren dagegen eine Wolke aus wahllos verteilten Punkten.

Für die Korrelation zwischen Kalorien und Gewichtsdifferenz vereinigen Sie die Werte dafür erst mal in einem gemeinsamen DataFrame. Die Zeitangaben passen dafür leider noch nicht zusammen: Während die Kaloriendaten jeweils nur einen Tag benennen, enthalten die Gewichtsmessungen auch die Uhrzeit der Messung. Pandas rundet die Zeitangaben mithilfe eines `DatetimeIndex`, der die passende `.round()`-Funktion mitbringt:

```
from pandas import DatetimeIndex
dw = df_weight[['date', 'diff']]
dw['date'] = DatetimeIndex(dw['date']
                           ).round(freq='D')
```

Danach stehen in beiden DataFrames auf Tage genaue Zeitangaben in der Spalte `"date"`. Die Funktion `.merge()` fügt diese nun zu einem DataFrame mit einer Tabelle zusammen:

```
dn = df_nutri[['date', 'Kalorien']]
dm = dn.merge(dw)
```

Merge nutzt hier standardmäßig den Parameter `how="inner"` sodass `.merge()` alle Zeilen aussortiert, die nicht in beiden Tabellen stehen. Setzt man stattdessen `how="outer"`, verwirft Pandas keine Zeilen und es entsteht eine Tabelle mit vielen undefinierten Einträgen (`NaN`). Andere von SQL bekannte Joins wie `"left"` und `"right"` funktionieren auch.

`.merge()` fügt die Tabellen außerdem standardmäßig so zusammen, dass die Werte in allen gleich benannten Spalten übereinstimmen müssen. Mit dem Parameter `on=` lässt sich das auf bestimmte Spalten eingrenzen. Die ausführliche Zeile `dm = dn.merge(dw, on="date", how="inner")` produziert daher das gleiche Ergebnis wie `.merge(dw)`.

Mit diesen Daten zeichnet Altair mit `.mark_circle()` ein Streudiagramm:

```
alt.Chart(dm).mark_circle(size=60
 ).encode(x='Kalorien',
          y=alt.Y('diff',
          title="Gewichtsveränderung"),
  tooltip=['date']
 ).properties(width=400, height=400
 ).interactive()
```

Das leere `.interactive()` am Ende sorgt für ein Diagramm, das sich gleichzeitig in x- und y-Richtung zoomen lässt. Der Trick versteckt sich im Parameter `tooltip=['date']` in der `.encode()`-Funktion. Durch ihn zeigt das Diagramm in einem kleinen Fenster das Datum zu jedem Punkt, sobald man mit der Maus darüber fährt. Das ist praktisch, um Ausreißer zu identifizieren.

Da sich im Diagramm keine Diagonale ergibt, ist die These widerlegt, dass auf ein Festmahl am nächsten Tag ein höheres Gewicht folgt. Scheinbar

fallen andere Effekte wie der Füllstand der Verdauung oder Wassereinlagerungen im Körper stärker ins Gewicht als die Größe der Mahlzeiten.

Wöchentliche Durchschnitte korreliert?

Sollten Verdauung und Wassereinlagerung das Gewicht im Tagesrhythmus schwanken lassen, wäre das ein hochfrequentes Rauschen, das sich ausgleicht, wenn man Durchschnitte über einen längeren Zeitraum bildet. Wöchentliche Durchschnitte berechnet Pandas auch im Handumdrehen:

```
from pandas import DateOffset
dm['dg'] = dm['date'] - DateOffset(
                    weekday=0, weeks=1)
dmm = dm.groupby(by='dg', as_index=False)[
          ['date', 'Kalorien', 'diff']
                  ].mean()
```

Zuerst erstellt der Code dafür eine neue Spalte mit auf Wochen gerundeten Zeitangaben. Dabei hilft `DateOffset(weekday=0, weeks=1)`, das für jede Zeile die Differenz zur nächsten vollen Woche berechnet. Zieht man diese Spalte von der tagesgenauen Zeit ab, bleiben auf Wochen gerundete Zeiten.

Im nächsten Schritt fasst `.groupby()` alle Zeilen zusammen, die zur selben Woche gehören. Für die so gruppierte Tabelle berechnet `.mean()` den Durchschnitt der Kalorien und Gewichtsdifferenz. Im DataFrame `dmm` steht damit am Ende eine wesentlich kürzere Tabelle mit einer Zeile pro Woche und den berechneten Durchschnitten. Das Diagramm dazu unterscheidet sich nur darin, dass es die Punkte im `tooltip=` mit der Woche beschriftet:

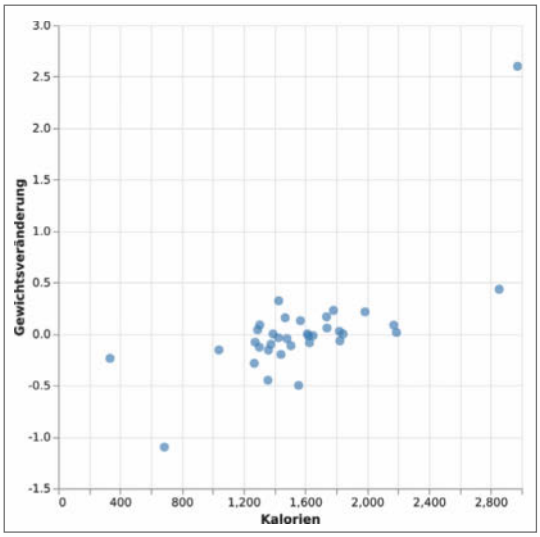
```
alt.Chart(dmm).mark_circle(size=60
 ).encode(x='Kalorien',
          y=alt.Y('diff',
          title="Gewichtsveränderung"),
  tooltip=['dg']
 ).properties(width=400, height=400
 ).interactive()
```

Das entstehende Diagramm zeigt weiter keine Diagonale. Die Punkte streuen aber deutlich weniger, was auf eine geringe Korrelation hindeutet.

Interessant sind vor allem die Ausreißer: Bei dem Punkt links in der Mitte habe ich eine Woche gefastet. Beim Fasten schaltet sich die Verdauung weitgehend ab. Da die Verdauung selbst circa 800 Kalorien verbraucht, habe ich durch eine Woche Fasten kaum Gewicht verloren. Der Punkt links unten zeigt eine Woche, in der ich nur knapp 700 kcal gegessen habe und trotzdem mehr abgenommen habe als in der Fastenwoche.

Der Punkt rechts oben entstand im Skiurlaub. Dort habe ich mich viel bewegt, viel gegessen und dadurch viel Muskelmasse zugelegt. Bei dem Punkt rechts in der Mitte war ich dagegen auf einer Konferenz, bei der ich zwar viel geschlemmt, mich aber nur wenig bewegt habe. Statt Muskeln habe ich dort nur ein halbes Kilo Fett angesetzt.

Notebooks bei GitHub, Testdaten:
www.ct.de/wbxj



Das Streudiagramm mit wochenweise gemittelten Werten zeigt immer noch keine starke Korrelation zwischen Kalorien und Gewicht. Die Ausreißer sind aber interpretierbar

Schnelle Statistiken

Pandas und Altair filtern und plotten auch große Datenberge in Windeseile. Damit steht Ihnen die ganze Macht statistischer Auswertung mit nur wenigen Zeilen Code zur Verfügung. Welche Statistiken Sie weiter bringen, müssen Sie allerdings vorher wissen. Die Hauptarbeit besteht daher meist in der Auswahl der statistischen Methode und vor allem im Sammeln der Daten.

Für diese Beispiele musste ich über Monate 5 bis 10 Minuten pro Tag ins Loggen von Kalorien investieren. Für die gesamte Auswertung mit Pandas und Altair habe ich dagegen nicht länger als zwei Stunden gebraucht. Im Idealfall sammelt ein Gerät die Daten daher automatisch, beispielsweise eine Solaranlage oder eine smarte Heizung. Es lohnt sich, nach solchen Datenquellen zu suchen und sie mal schnell in einem Jupyter-Notebook zu aufschlussreichen Diagrammen aufzuarbeiten. (pmk) c't

Covid-19-Rechenmodelle

Für eine Lösung der Corona-Krise müssen etwa zwei Drittel der Bevölkerung immun gegen das Virus werden. Das geht entweder mit Patienten, die die Krankheit überleben, oder mit einem Impfstoff. Wir zeigen, wie man das durchrechnet und visualisiert.

Von Pina Merkert

Nachrichten, Sondersendungen, Extra-Beiträge – das Corona-Virus ist das beherrschende Thema in den Medien. Dort kommen auch Forscher zu Wort, die zwar die Gefahren des Virus detailreich beschreiben, der Frage nach einer Lösung und der Zeitspanne bis zur Lösung aber ausweichen. Glücklicherweise können Sie die Lücke mit ein paar Zeilen Python-Code einfach selbst schließen.

Das zugrunde liegende Modell ist sehr grob, die Vorhersagen daher vage. Für eine grobe Einschätzung, wie lange die Politik gezwungen sein wird, die Ausgangsbeschränkungen in Deutschland bestehen zu lassen, reicht es aber aus.

Die Anzahl der Mitmenschen, die ein Infizierter im Schnitt ansteckt – die sogenannte Basisreproduktionszahl R_0 des Virus –, liegt in einer Gesellschaft wie in Deutschland (vor der Einführung der aktuellen Bewegungseinschränkungen) bei ungefähr 3. Dadurch wächst die Anzahl der Erkrankten exponentiell. Ohne Gegenmaßnahmen würde das schnell zu einer massiven Überlastung des Gesundheitssystems führen.

Damit sich das Virus nicht mehr weiter verbreiten kann, muss R_0 unter 1 sinken. Dann würde es langsam aussterben. Denn wenn keiner immun ist, stecken sich 3 Leute an. Wenn 2 von 3 immun sind, steckt sich nur noch einer an und die Epidemie stoppt.

Zwei Drittel Immune in der Bevölkerung erreicht am schnellsten ein Impfstoff. Es wird zwar an mehreren geforscht, keines der Teams glaubt aber an eine Massenproduktion vor Frühjahr 2021. Alternativ gibt es irgendwann genügend Menschen, die eine Covid-19-Infektion überstanden haben. Deren Anzahl wächst aber bei einer Infektionsrate, die für das Gesundheitssystem verkraftbar ist, nur sehr langsam.

Eine direkte Möglichkeit, um R_0 unter 1 zu drücken, besteht darin, die Kontakte von Infizierten mit Mitmenschen so weit zu beschränken, dass sich nur noch ganz wenige anstecken können. Das versucht die Politik mit den derzeitigen Ausgangsbeschränkungen und Social Distancing zu erreichen. Dabei nimmt man in Kauf, dass die Zahl der genesenen und damit immunen Menschen deutlich langsamer wächst, als für eine ausreichende „Herdenimmunität“ nötig wäre. Letztlich dienen die derzeitigen Maßnahmen also dazu, auf einen Impfstoff zu warten. Würde man die Ausgangsbeschränkungen nämlich aufheben, würde R_0 sofort wieder auf etwa 3 steigen und es gäbe eine zweite Infektionswelle.

Das im Folgenden vorgestellte Rechenmodell benutzt aktuelle Infektionszahlen, um eine Prognose über den weiteren Verlauf der Epidemie zu berechnen und zu visualisieren. Sie können es mit jeder aktuellen Python-Distribution (Beispielsweise Anaconda unter Windows) nachvollziehen. Den Code (ein Jupyter-Notebook) bekommen Sie aus unserem GitHub-Repository (siehe ct.de/wwpm). Alle nötigen Bibliotheken wie SciPy, Pandas und Altair installieren Sie bequem mit `pip`. Für MacOS und Linux reichen die folgenden Befehle auf der Konsole:

```
git clone https://github.com/pinae/↴
            ↳Covid-Predict.git
cd Covid-Predict
python3 -m venv env
```

```
source env/bin/activate
pip install -U
pip wheel pip install -r requirements.txt
```

Unter Windows aktivieren Sie das Virtualenv grafisch über Anaconda, öffnen über das Menü ein Konsolenfenster und tippen dort nur die letzten beiden Befehle ein. Den lokalen Webserver für das Jupyter-Notebook starten Sie mit

```
jupyter notebook
```

Daten sammeln

Für eine Abschätzung, wie sich die Infektionszahlen weiter entwickeln, kann man die bisherigen Werte extrapolieren. Dafür haben wir Zahlen für die Corona-Infektionen in Deutschland von worldometers.info (siehe ct.de/wwpm) abgetippt. Pandas lädt die Daten mit einer Zeile, eine weitere konvertiert die Datumsangaben:

```
data = pd.read_csv(
    "corona_infections.csv",
    header=0, ?
    names=["day", "cases"])
data["day"] = [?
  dt.datetime.strptime(d + '2020',
    "%b %d %Y") for d in data["day"]]
```

Die CSV-Datei enthält lediglich in der ersten Spalte eine Datumsangabe wie „`Mar 20`“ und nach einem Komma in der zweiten Spalte die Zahl der Infizierten. Der zweite Befehl des obigen Codes konvertiert die kurzen Datumsangaben unter Ergänzung des Jahres in `datetime`-Objekte, womit später die X-Achse im Diagramm sinnvoll beschriftet wird.

Leider enthält der Datensatz bisher nur wenige verwertbare Datenpunkte. Daher ist es sinnvoll, zunächst mit einem einfachen Modell anzufangen: Ein Virus trifft zunächst auf viele Wirte, wodurch es sich exponentiell verbreitet. Je mehr der Wirte jedoch bereits infiziert sind, desto weniger neue Wirte findet es, sodass sich die Verbreitung später einem Maximum annähert. Ein solches Wachstum modelliert die logistische Funktion:

```
def corona_curve(x, b0, x0, k, s):
  return s * 1 / (1 + np.exp(
        -1 * k * s * (x - x0)
        ) * (s / b0 - 1))
```

Sie steigt vom Basiswert (ca. 20.000 Infizierte) `b0` zunächst exponentiell an, erreicht aber einen Wendepunkt, ab dem sie abflacht und sich an ein Maximum `s` annähert. Eine sinnvolle obere Grenze für `s` ist die Bevölkerungszahl Deutschlands (ca. 83 Mio.). In der Praxis ergeben sich aber kleinere Werte, da sich die Kurve bei geringerer Reproduktionszahl des Virus nicht nur abflacht, sondern auch schneller eine Sättigung erreicht.

Maßgeblich für die Steilheit der größten Steigung ist die Wachstumskonstante `k`. Die Bedeutung der Konstante wird klarer, wenn man ein paar Berechnungen mit ihr anstellt, die in der Formel nicht vorkommen müssen: Rechnet man e^k (in Python `np.exp(k)`), kommt man von der Konstante auf den Wachstumsfaktor `b`. `1-b/100` gibt an, wie viele Prozentpunkte die Kurve pro Zeitschritt wächst.

Der Parameter `x0` verschiebt die Kurve lediglich entlang der X-Achse, damit die Werte zum richtigen Datum passen.

Um ein Modell zu entwickeln, das beschreibt, wie sich die Infektionszahlen entwickeln werden, muss man die vier Parameter der Funktion `corona_curve()` so wählen, dass sie möglichst gut zu den Werten aus dem Datensatz passen. Um diese Aufgabe kümmert sich `curve_fit()` aus dem Python-Modul `scipy.optimize` (siehe Seite 154). Um sie zu nutzen, brauchen Sie Daten für X und Y als Numpy-Array.

Seit dem 23. März sind in allen Bundesländern weitgehende Maßnahmen inkraft, die die Kontakte der Bevölkerung stark einschränken und damit die Wachstumskonstante beeinflussen. Wegen der Inkubationszeit sind diese Maßnahmen in der Kurve aber durchschnittlich erst ab dem 28. März sichtbar (Variable `fqd`). Eine Prognose sollte sich deshalb nur auf die Zahl der nachgewiesenermaßen Erkrankten seit diesem Tag stützen. Die folgenden Zeilen extrahieren daher die Werte seit dem 28. März als Numpy-Arrays und speichern die Tag-Nummern in `days_since_quarantine` und die Infektionszahlen in `cases_since_quarantine`:

```
fqd = dt.datetime(year=2020, month=3, day=28)
cases_since_quarantine = np.array(
                data[data["day"] >= fqd]["cases"])
day_no_since_quarantine = np.array(
 [d.toordinal() for d in data[
                    data["day"] >= fqd]["day"]])
```

Die vier Parameter von `corona_curve()` soll `curve_fit()` nun automatisch so wählen, dass sie möglichst genau zum vorbereiteten Datensatz passen. Die Funktion startet dafür standardmäßig mit einem Wert von 1 für alle Parameter und arbeitet sich

schrittweise an ein optimales Ergebnis heran. Falls die Exponentialfunktion bei diesem Prozess zu große Werte liefert, kommt es zu Fehlern.

Deswegen übergibt der Code im Jupyter-Notebook über den Parameter `p0` zusätzlich eine Liste mit besseren Startwerten als 1, mit denen die Optimierung stabiler läuft. Eine ähnliche Hilfestellung bieten die `bounds`, die Listen mit Minimal- und Maximalwerten für die Parameter enthalten. Beispielsweise kann die Maximalzahl infizierter Personen `s` nie die Bevölkerungszahl Deutschlands übersteigen. `xdata` und `ydata` enthalten die Eingaben und erwarteten Ausgaben der Funktion:

```
params, _ = curve_fit(
  corona_curve,
  xdata=day_no_since_quarantine,
  ydata=cases_since_quarantine,
  p0=[cases_since_quarantine[0],
   dt.datetime(year=2020, month=3,
   day=22).toordinal(),
   8e-9, 5.6e7],
  bounds=(
   [0, day_no_since_quarantine[0],
   1e-11,
   cases_since_quarantine[-1]],
   [cases_since_quarantine[-1],
   dt.datetime(year=2021,
   month=6, day=1).toordinal(),
   1e-8, 8.35e7])
)
```

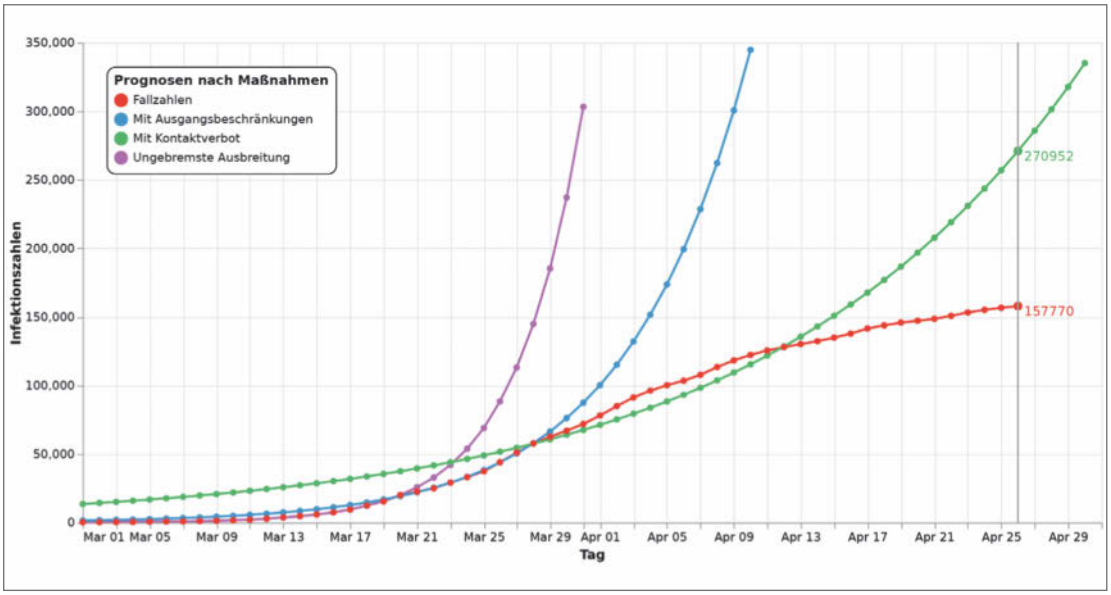
Die vier Parameter `b0, x0, k` und `s` gibt `curve_fit()` anschließend als ersten Rückgabewert `params` zurück (der zweite enthält Informationen zum Fehler).

In dem 4-Tupel `params` interessieren danach vor allem der dritte und vierte Wert: Je kleiner `k`, desto weniger schnell steigen die Infektionszahlen. Der vierte Wert `s` ist mit Vorsicht zu genießen, da das Maximum der Infizierten (wer genesen ist, wird in dieser Statistik weiter mitgezählt) mit den wenigen bisher zur Verfügung stehenden Werten stark schwanken kann. Ein verlässliches `s` liefert SciPy wohl erst, wenn sich die Kurve dem Wendepunkt nähert.

Visualisiert

Die Kurve der tatsächlichen und der erwarteten Infektionszahlen zeigt ein Diagramm viel anschaulicher als eine Tabelle. Mittel der Wahl ist da Altair (siehe S. 154), dessen Diagramme Jupyter-Notebooks praktischerweise direkt anzeigen:

```
infections_chart = alt.Chart(
  data).mark_line(point=True,
  color="red").encode(
 alt.X("monthdate(day):O",
```



Vorhersage des exponentiellen Wachstums je nach Einschränkungen bis zum 1. Mai: Bei zirka 240.000 erkrankten werden die Intensivbetten nicht mehr ausreichen.

Aktualisieren Sie dieses Diagramm selbst mit dem zum Artikel gehörenden Jupyter-Notebook.

```
    title="Tag"),
 alt.Y("cases:Q", title="Corona-Fallzahlen"))
projection_chart = alt.Chart(
    projection_data).mark_line(
    point=False).encode(
 alt.X("monthdate(day):O",
    title="Tag"),
 alt.Y("curve:Q",
    title="Fallzahlen(Projektion)"))
projection_chart + infections_chart
```

Die Deutsche Interdisziplinäre Vereinigung für Intensiv- und Notfallmedizin (DIVI) hat am 20. März 4800 Intensivbetten vermeldet, die für schwere COVID-19-Fälle zur Verfügung stehen (Quellen siehe ct.de/wwpm). Schwer, mit Bedarf für Beatmung, verlaufen nach den FAQ des Kobert-Koch-Instituts etwa 2 Prozent der Fälle. Eine Überlastung könnte daher bereits ab 240.000 Erkrankten eintreten.

Immunität statt Impfstoff

Statt auf einen Impfstoff zu warten, könnten einfach so viele Menschen Covid-19 überleben, dass irgendwann zwei Drittel auf natürlichem Weg immun werden. Experten vermuten aber, dass die Corona-Immunität wie bei der Grippe nicht ewig hält. Schätzungen reichen von sechs Monaten bis anderthalb Jahren Immunität. Um rechtzeitig zwei Drittel zu erreichen, müssten viel mehr Menschen gleichzeitig krank werden als jetzt. Für die Berechnung reicht ein Dreisatz:

```
bevoelkerung_de = 83019213
infizierte_pro_tag =
    ["{:.0f}".format(?
    bevoelkerung_de*0.7/(365*dauer)
    ) for dauer in [0.5, 1, 1.5]]
```

Ergebnis: Damit der Plan aufgeht, müssten sich im besten Fall, wenn die Immunität 18 Monate hält, täglich über hunderttausend Menschen neu anstecken.

Die hier gezeigten Modelle sind sehr grob und ersetzen keinesfalls die ausgefeilteren Vorhersagen der Experten. Wenn die aber Antworten schuldig bleiben, kann man sich mit ein paar Zeilen Python effektiv selber behelfen. Nach den aktuellen Zahlen und mit etwas Hoffnung muss Deutschland wohl noch mindestens über das komplette Jahr zu Hause bleiben.

Falls Sie Gefallen an der Modellierung der Kurve gefunden haben, können Sie nicht nur mit dem Code im Jupyter-Notebook herumspielen. Über ct.de/wwpm finden Sie außerdem Links zu mehreren webbasierten Berechnungen mit komplexeren Modellen, darunter auch ein interaktiver Plot aufbauend auf dem Standardmodell der Epidemiologie. (pmk) c't

Literatur

[1] Pina Merkert, **Formelsuche für Faule,** Parameter optimieren mit SciPy curve_fit(), c't 12/2019, S. 188

Anaconda, Jupyter-Notebook bei GitHub

ct.de/wwpm

Bild: Albert Hulm

Beliebige Bäume in flachem SQL

SQL-Datenbanken speichern nur flache Tabellen. Reale Daten liegen aber oft baumartig strukturiert vor. Mit einem generischen Konzept für die Tabellen und den modernen Array- und JSON-Feldern von PostgreSQL speichert die Datenbank beliebige Daten und prüft dabei auch deren Struktur.

Von Pina Merkert

Texte zu speichern für Blogs, Flugblätter oder Magazine erscheint einfach: Eine Datenbankanwendung speichert Artikel mit Überschriften, Autorenzeilen und etwas Text mittels SQL und erzeugt daraus automatisch alle nötigen Dateien für den Webserver oder das Desktop-Publishing-Programm.

Doch in der Praxis tauchen schnell Begehrlichkeiten auf. Beispielsweise sollen Textkästen in den Artikeln die Leser mit Zusatzinfos versorgen – und schon müsste das Datenbankschema für die neue Anwendung angepasst werden. Schreiben außer dem Programmierer andere Autoren mit der Software, fehlen denen die Fähigkeiten, das System an neue Bedürfnisse anzupassen („Im Fließtext hervorgehobene Zitate wären doch toll!“). Der Programmierer wird zwangsläufig zum Flaschenhals.

Das Problem besteht im Aufbau von relationalen Datenbanken, da diese Daten tabellarisch und nicht hierarchisch ablegen. Mit dem richtigen Datenbankschema und den raffinierten Array- und JSON-Feldern von PostgreSQL zeigen wir am Beispiel einer Django-Anwendung namens „AssetStorm“, wie man strukturierte Daten in eine SQL-Datenbank bekommt, ohne sich während des Entwickelns auf deren Schema festzulegen. Völlig

schemalos wie in einer MongoDB soll es aber auch nicht zugehen, damit Anwender gar nicht erst die Möglichkeit bekommen, falsche Daten abzulegen. Stattdessen speichert die Datenbank sowohl die erlaubten Schemas als auch die Daten. Das gestattet der Anwendung, die Daten beim Speichern zu validieren. Dass Datenstrukturen dabei in anderen Datenstrukturen stecken, stört das System nicht: Die Daten dürfen nämlich baumförmig verästelt sein. AssetStorm bleibt dabei offen für neue Begehrlichkeiten, die auch Anwender ohne Programmierkenntnisse befriedigen können. Unsere Referenzimplementierung der Idee finden Sie im AssetStorm-Repository über ct.de/ymhx.

Baumförmige Daten

Strukturierte Daten wie die Texte in unserem Beispiel sind baumförmig verschachtelt: In einem Artikel (Stamm) gibt es Absätze und andere Blockelemente (Äste), in denen Inline-Formatierungen wie Fettungen oder Links eingebettet sind (Blätter). Sie kennen das beispielsweise aus der HTML-Struktur von Webseiten. Damit die Datenbank für diese sehr unterschiedlich strukturierten Elemente nur eine Tabelle braucht, muss sie die Anwendung in einem generischen Format speichern, das immer passt, egal an welche Stelle im Baum die Daten gehören. Unser Beispiel bezeichnet alle Elemente eines solchen Baums, die eine innere Struktur besitzen, als „Asset“.

Der Trick besteht darin, dass jedes `Asset` eine Referenz auf einen Typ (`AssetType`) besitzt. In einer anderen Tabelle speichert AssetStorm, wie die Struktur auszusehen hat. Das Schema eines solchen Typs speichert die Datenbank als JSON, ein Datentyp, den PostgreSQL seit Version 9.2 kennt. Das Schema enthält Key-Value-Paare, bei denen der Wert zu jedem Schlüssel aus einer ID eines Typs besteht. Gibt man stattdessen eine Liste mit einer einzelnen ID an, erlaubt die Anwendung laut Schema eine beliebig lange Liste an Assets des angegebenen Typs.

Blätter für den Baum

Während Asset-Typen die Strukturen definieren, enthalten sie jedoch lediglich Schlüssel und keine Daten. Für die gibt es in AssetStorm drei atomare Datentypen, die die Blätter des Baums darstellen: Textelemente, URLs und Enums, also aufzählbare Daten, wie die Programmiersprache eines Listings, bei denen der `EnumType` festlegt, welche Einträge zur Auswahl stehen. Diese atomaren Datentypen nutzen in den Schemas die IDs 1, 2 und 3. Ab ID 4 geht es mit den Asset-Typen aus der `AssetType`-Tabelle weiter.

Bei Enums ist es nötig, im Schema nicht nur den Typ 3 anzugeben, sondern zusätzlich den `EnumType`. AssetStorm erwartet deswegen im Schema für Enums ein Dictionary mit dem einzelnen Schlüssel `"3"` und der ID des `EnumType`. Erlaubt ein Schlüssel beispielsweise die Auswahl einer Programmiersprache (`EnumType` mit ID 2 in den Tests), gibt man im Schema folgendes Dictionary an:

```
"language": {"3": 2}
```

Assets in Assets

Fragt ein Request den Inhalt eines Assets ab, holt sich die Anwendung zuerst den zugeordneten Typ und dessen Schema. Passend zum Schema enthält jedes `Asset` ein JSON-Feld namens `content_ids` mit Key-Value-Paaren. In dem steht hinter jedem Schlüssel eine Referenz in eine der vier Tabellen mit weiteren Daten: Texte, URLs, Enums oder andere Assets. In welcher Tabelle die Anwendung jeweils nachschauen muss, verrät das Schema aus dem `AssetType`, in dem dieselben Schlüssel stehen (ID 1 bis 3 sind die Tabellen der Texte, URLs und Enums, Werte ab 4 geben die Asset-Typen an).

Mit diesen Informationen hangelt sich die Anwendung durch den `Asset`-Baum: Hinter jedem Schlüssel in den `content_ids` kann ein weiteres Asset stecken, das seine eigene Struktur mit Key-Value-Paaren mitbringt. Die eingebetteten Assets fügt das Programm dann als JSON-Dictionary unterhalb des Schlüssels ein. Erlaubt das Schema eine Liste, packt es die Assets hintereinander in eine JSON-Liste. Die Verästelung endet bei den Blättern vom Typ 1 bis 3, die als einfache Strings in der Datenstruktur landen. Ein weitgehend vom Text befreites `Asset` namens `"article"` könnte beispielsweise so aussehen:

```
{
  "type": "article",
  "title": "Testilinio",
  "subtitle": "Test-Artikel",
  "abstract": "Ein Vorlauf ganz kurz",
  "author": "Pina Merkert",
  "content": [
    {
      "type": "block-paragraph",
```

```
      "spans": [
        {
          "type": "span-regular",
          "text": "Text mit Code:"
        }
      ]
    },
    {
      "type": "block-listing",
      "language": "python",
      "code": "a = 2 + 5\nprint(a)"
    }
  ]
}
```

Dieses Beispiel verwendet vier Asset-Typen: `"article"`, `"block-paragraph"`, `"block-listing"` und `"span-regular"`. Bei der Liste an Blockelementen im `"content"` des `"article"` gibt es eine Besonderheit: Das Schema des `"article"` definiert hier nämlich eine Liste von Assets vom Typ 5 (`"block"`):

```
{"title": 1,
 "subtitle": 1,
 "author": 1,
 "abstract": 1,
 "content": [5]}
```

Die tatsächlichen Asset-Typen `"block-paragraph"` und `"block-listing"` haben aber im Beispiel die IDs 15 und 31. Sie passen ins Schema, weil ein `AssetType` mit der optionalen Referenz `parent_type` einen Basistyp angeben kann. Überall wo der Basistyp `"block"` mit der ID 5 erlaubt ist, passen auch Assets in den Baum, die beim `parent_type` auf ID 5 verweisen.

Caching

Für jede Abfrage eines Assets einen Baum zu durchlaufen und jeweils ein ganzes Paket an `Asset`, `Text`, `UriElement`, `Enum`, `EnumType` und `AssetType`-Objekten aus der Datenbank zu laden, wäre zu langsam. Deswegen befüllt ein `Asset`-Objekt einen `content_cache`, sobald es nach seinem Inhalt gefragt wird (beim Zugriff auf die Property `content`). Der `content_cache` ist ein JSON-Datenfeld, das den ganzen Baum unterhalb dieses Assets speichert.

Ein Problem entsteht, wenn Assets verändert werden: Dann ist der Cache nämlich plötzlich veraltet und muss gelöscht werden. Wegen der Baumstruktur veralten aber auch alle Caches, in denen dieses Asset eingebettet ist. Um all diese Assets schnell zu finden, pflegt jedes Asset je eine Liste für die `Text`, `UriElement`, `Enum` und `Asset`-Objekte, die in `content_ids` referenziert werden. Diese Array-Felder lassen schnelle Abfragen zu, in welchen Assets ein Asset mit frisch invalidiertem Cache eingebettet ist. Wegen der Baumstruktur ist allerdings für jedes der gefundenen Assets eine rekursive Abfrage nötig. Änderungen in der Asset-Datenbank sind deswegen bei der Rechenzeit deutlich teurer als Leseoperationen.

JSON-Schema

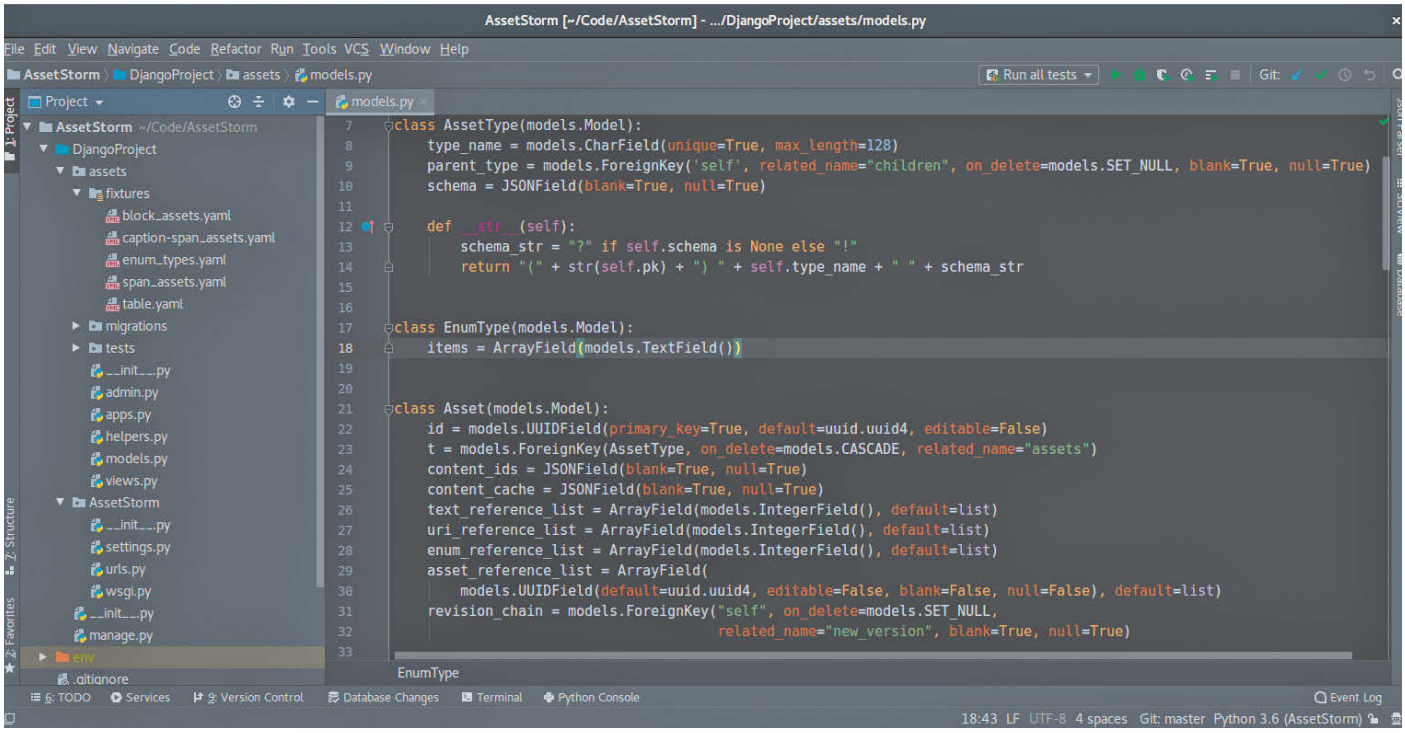
Was AssetStorm als „Schema“ bezeichnet, ist nichts weiter als eine Liste von Schlüsseln, deren zugehörige Werte einem angegebenen Datentyp entsprechen. Beispielsweise hat ein Auto die Schlüssel „Motor“, „Modellname“ und „Energieverbrauch“. Die Datentypen wären in dem Fall Motortyp, Text und eine Angabe in der Einheit kWh/100km:

```
"Auto" = {
  "Motor": 7,
  "Modellname": 1,
  "Energieverbrauch": 1
}
```

Dargestellt als JSON geben die Schlüssel jeweils die ID des Typs an. Ein Typ wie der „Motortyp“ mit ID 7 hat dabei ein eigenes Schema:

```
"Motortyp" = {
  "Drehmoment": 1
  "Maximaldrehzahl": 1
}
```

Definiert man an der `AssetType`-Tabelle in AssetStorm diverse Typen mit jeweils einem Schema, erlaubt das baumförmig strukturierte Daten. Wie diese Bäume aussehen und wo welche Schlüssel erlaubt sind, ist dabei alles andere als frei: Die Schemas legen genau fest, wann Datensätze eine valide Struktur besitzen.



Der Screenshot aus der IDE PyCharm zeigt, wie einfach die Datenbankmodelle trotz PostgreSQL-spezifischer JSON- und Array-Felder in Django aussehen.

Speichern ohne Fehler

Beim Speichern schickt ein Frontend mit Editor eine JSON-Datenstruktur an die Datenbank. Für unser Beispiel haben wir kein Frontend programmiert, in einer realen Anwendung gehörte das aber dazu. Das Frontend könnte dabei Fehler machen, beispielsweise wenn es beliebiges Markdown oder XML stumpf in JSON übersetzt, ohne sich darum zu scheren, welche Asset-Typen es gibt und wie diese auszusehen haben. Die Anwendung muss die angelieferten Bäume daher zuerst prüfen. Bäume, die irgendwo den Schemas der Asset-Typen widersprechen, muss die Anwendung mit einer Fehlermeldung ablehnen.

Dafür hangelt sich die Anwendung durch den im Request gelieferten JSON-Baum und sucht auf jeder Ebene nach dem Schlüssel `"type"`. Hinter dem muss der Baum nämlich stets den Klarnamen (`type_name`) des `AssetType` angeben. Fehlt `"type"`, liefert die Anwendung direkt eine Fehlermeldung im JSON-Format, die die fehlerhafte Stelle im Baum benennt.

Mit dem Wissen um den Asset-Typ kann die Anwendung den gelieferten Baum dahingehend überprüfen, ob er dem im Typ gespeicherten Schema entspricht. Jede Abweichung führt zu einer Antwort mit Fehlerbeschreibung. Nur wenn alle im Schema verlangten Schlüssel vorhanden sind und die Typen zum Schema passen, bevölkert das Programm auch die Datenbank.

Da ein Speichern-Request einen ganzen Baum enthält, legt die Anwendung meist einen ganzen Schwung an `Asset`-Objekten an. Dass die einzelnen Assets dabei nur im Cache einen Baum speichern und sich sonst mit Key-Value-Paaren begnügen, ist für Nutzer der Anwendung nicht von Belang. Sie schicken JSON-Bäume zu den API-Endpunkten und bekommen bei Anfragen auch nur komplette JSON-Bäume vom Server zurück.

Die Antwort einer Anfrage zum Speichern enthält im Erfolgsfall die ID des Wurzel-Assets, das alle

anderen Assets enthält. Mit dieser ID kann ein Frontend den Baum sofort wieder abfragen, um zu erfahren, welche IDs das System auf allen Ebenen vergeben hat.

Assets modifizieren

Zum Modifizieren braucht das Frontend keinen eigenen API-Endpunkt. Es schickt einfach eine Baumstruktur mit gültigen `"id"`-Schlüsseln an den API-Endpunkt zum Speichern. Die Anwendung lädt dann die per ID benannten Assets und vergleicht deren Inhalt mit den Informationen im Baum. Gibt es Unterschiede, ändert sie die Assets, löscht an den nötigen Stellen die Caches und speichert die alte Version des geänderten Assets mit neuer ID in einer Kette, um eine Versionsverwaltung zu implementieren.

Bei Anfragen zum Speichern mit `"id"` ist die Software auch etwas nachgiebiger beim Erfüllen der Schemas: Fehlt ein Schlüssel im Baum, holt sich die Anwendung den Wert aus dem bekannten Asset.

Ein Konzept für alle Bäume

Die Erklärung hat sich bisher auf ein Beispiel mit Artikeln, Absätzen und formatierten Textschnipseln gestützt, wie sie bei Webseiten oder einer Print-Publikationen vorkämne. Die Unittests der Implementierung auf GitHub (siehe ct.de/ymhx) nutzen auch dieses Beispiel. Das Konzept läuft aber mit allen strukturierten Daten, die sich über die Schemas in den Asset-Typen definieren lassen.

Beispielsweise ließe sich damit auch eine hierarchische Datensammlung zu Hirnregionen und den passenden Papern aus der Neurowissenschaft umsetzen. Die Asset-Typen wären dann Regionen, Unterregionen und Bibliographie-Einträge. Am Code müsste man dafür nichts ändern. Lediglich die Schemas der Asset-Typen würden anders aussehen.

PostgreSQL für alles

AssetStorm zeigt, wie flexibel eine SQL-Datenbank wird, wenn Arrays, Key-Value-Sammlungen und JSON-Felder die üblichen Datentypen erweitern. Mit diesen Typen verschwimmt die Grenze zu modernen Dokumentendatenbanken wie MongoDB. Djangos Datenbank-Wrapper macht den Umgang mit den PostgreSQL-spezifischen SQL-Erweiterungen zu einem Spaziergang. Die Definition der Datenbankmodelle in `DjangoProject/assets/models.py` ist selbsterklärend – schauen Sie ruhig mal in den Quellcode. Lediglich der Code in den Methoden und Propertys der Model-Klassen hat die Komplexität von üblichem Python in Django-Projekten.

AssetStorm ist extrem flexibel. Um es an die eigenen Anwendungsfälle anzupassen, müssen Sie lediglich JSON-Strukturen definieren. Schreiben Sie uns, wenn Sie die Schemas an Ihre Bedürfnisse anpassen. Falls wir etwas vergessen haben, freuen wir uns über Pull-Requests auf GitHub. (pmk) c't

Microservice

Unsere Implementierung AssetStorm ist als Microservice gedacht. Die Anwendung läuft dafür in einem Container oder einer virtuellen Maschine und kommuniziert mit der Außenwelt ausschließlich über ihr REST-API per HTTP. Eine API-Definition nach OpenAPI-3.0-Standard liegt dem Repository bei.

Das bedeutet aber auch, dass AssetStorm kein Frontend mitbringt. Das Frontend muss in einem eigenen Microservice laufen, beispielsweise als progressive Web-App mit React. Dass AssetStorm funktioniert, belegt dabei auch nicht die Kommunikation mit einem Frontend, sondern zahlreiche automatische Tests. Django erweitert die Python-Unittest-Infrastruktur, sodass auch Integrationstests mit simulierten HTTP-Requests möglich werden. Die Tests dienen nicht nur dazu, die korrekte Funktion von AssetStorm zu sichern, sie eignen sich auch als Beispiele zur Orientierung für Frontend-Programmierer.

Pandoc in Flask

In Markdown geschriebener Text sieht schon im Texteditor lesbar aus. Der Markup-Konverter Pandoc konvertiert so einen Text in noch hübschere Formate. Mit einem in Python programmierten Webdienst auf Basis des schlanken Frameworks Flask fällt am Ende genau das Markup aus dem Konverter, das man will.

Von Pina Merkert

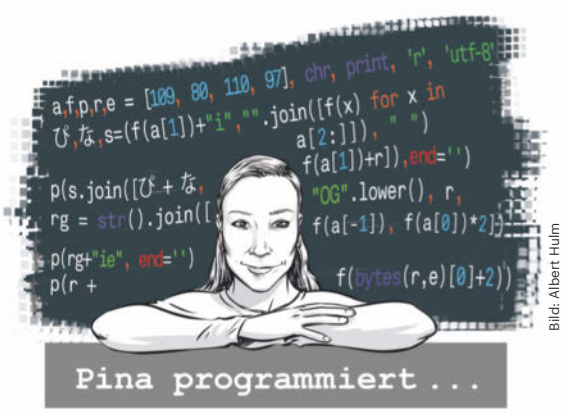


Bild: Albert Hulm

Das Konsolentool Pandoc konvertiert Markup-Formate wie MediaWiki-Markup, ReStructuredText oder Markdown in andere Markup-Formate wie ePub, Latex oder HTML. Um beispielsweise aus einer Markdown-Datei `test.md` ein HTML-Dokument `test.html` zu erzeugen reicht die folgende Zeile:

```
pandoc test.md -f markdown -t html -s -o test.html
```

Das Tool arbeitet auch als Filter auf der Konsole und konvertiert daher auch Text in Shell-Skripten. Für Python-Programmierer steht mit `pypandoc` ein Python-Wrapper bereit.

Als Programmiererin arbeite ich gern mit schmucklosen Texteditoren. Formate wie Markdown erlauben mir, meine Lieblingstools auch zum Schreiben von Prosa zu verwenden. Die effiziente Syntax garantiert eine gute Übersicht über den Text, gestattet mir aber auch mal eine Hervorhebung oder einen Quellcode-Schnipsel. Außerdem kann ich eine so erzeugte Dokumentation ohne Umwege auf GitHub hochladen.

So sehr ich beim Schreiben den spartanischen Editor schätze – soll ein Text gedruckt werden oder eine schicke Webseite befüllen, möchte ich auf Layouts mit Bildern, Kästen und Grafiken nicht verzichten. Fürs Layout gibt es Templates für InDesign oder CSS für den Browser. Und damit Text und Bilder ordentlich zusammenfinden, gibt es Datenbanken wie AssetStorm (siehe S. 166).

Leider kennt Markdown nur übliche Auszeichnungen wie Text, Links und Bilder. Kästen oder Bildergalerien fehlen in der Sprachdefinition von Markdown. Dementsprechend kann ich von Pandoc auch nicht erwarten, Kästen oder Bildergalerien in seine HTML- oder Latex-Ausgabe zu integrieren. Mit etwas Programmierfinesse und magischen Kommentaren im Dokument konvertiert ein Flask-Microservice mit Pandoc-Unterbau trotzdem alle Inhalte ins richtige Format für AssetStorm. Das Repository auf GitHub (ct.de/wphe) dokumentiert in allen Details, wie das funktioniert – eine lesbare Übersicht für Ungeduldige liefert dieser Artikel.

Flask für schlanke Microservices

Ein Microservice wie der Markdown-Konverter dieses Beispiels antwortet auf API-Aufrufe per HTTP. Die Python-Anwendung muss dafür aber nicht unbedingt ein großes Web-Framework wie Django einbinden. Flask kann zwar viel weniger, wer aber beispielsweise keine Datenbank anbindet, kommt damit effizienter und mit weniger Code zum Ziel.

Damit Flask läuft, reichen nämlich zwei Zeilen:

```
from flask import Flask
app = Flask(__name__)
```

Mit der so initialisierten `app` definiert man anschließend einfach eine Funktion für jeden Pfad (im Flask-Jargon eine `route()`), auf den die Anwendung antwortet. In dieser Funktion formuliert das Programm die Antwort als `flask.Response`-Objekt, wofür das `app`-Objekt eine bequeme Factory-Methode namens `response_class()` mitbringt:

```
@app.route("/", methods=['POST'])
def convert():
  md = request.get_data(as_text=True)
  data = {
    "type": "block-blocks",
    "blocks": json_from_markdown(md)}
  response = app.response_class(
    response=json.dumps(data),
    status=200,
    mimetype='application/json'
  )
  return response
```

Damit in der Antwort valides JSON steht, nutzt die Funktion Pythons `json`-Modul. Die selbst geschriebene Funktion `json_from_markdown()` übernimmt die eigentliche Arbeit.

Um zu testen, ob Flask korrekt funktioniert, kann man das Skript direkt aufrufen. `app.run()` startet dann einen Webserver, der HTTP-Anfragen auf Port 5000 beantwortet:

```
if __name__ == "__main__":
  app.run()
```

Auf der Konsole feuert folgender Befehl den passenden Request ab (das zu konvertierende Markdown steht in der Datei `test.md`):

```
curl -X POST --data-binary @test.md ↵
 ↳--header "Content-Type:text/plain" ↵
 ↳http://localhost:5000/
```

Die Flask-Dokumentation (siehe ct.de/wphe) erklärt, unter welchen Bedingungen Flask auf Produktivsystemen Gas gibt, beispielsweise mit uWSGI und Nginx.

Pandocs internes JSON-Format

Nun zum Markdown-Konverter: Pandoc konvertiert in diverse Formate. Meine flüchtige Zusammenstellung von Blockelementen und Spans (Formatierungen und Links im Fließtext) in AssetStorm ist allerdings nicht dabei. Damit Pandoc so viele Kombinationen von Markup-Sprachen übersetzen kann, nutzt es ein internes Datenformat, in das es zuerst einliest und aus dem es in einem zweiten Schritt das Markup der Ausgabesprache erzeugt. An die interne Repräsentation kommt man mit der Ausgabesprache `json` auch direkt.

Mit PyPandoc genügt ein einziger Befehl, um Markdown-Text (aus der Variable `markdown`) in JSON zu konvertieren. Der Befehl gibt allerdings einen String zurück, sodass das `json`-Modul die Datenstruktur noch parsen muss:

```
import pypandoc, json
pandoc_tree = json.loads(
      pypandoc.convert_text(markdown,
            to='json', format='md'))
```

Im `pandoc_tree` landet dann eine Datenstruktur mit einer Liste an Blockelementen. Blockelemente sind beispielsweise Absätze (`"Para"`), Überschriften oder Quellcode mit mehreren Zeilen.

Absätze bestehen ihrerseits wieder aus einer Liste an Spans. Das sind als Strings dargestellte Wörter, Leerzeichen, Sonderzeichen, aber auch fett gedruckte oder kursive Abschnitte. Spans bilden hintereinander gehängt die Zeilen, aus denen die Blockelemente bestehen.

JSON zu JSON

Blockelemente mit eingebetteten Spans sind eine Grundidee layouteter Texte. Um Pandocs Format so zu konvertieren, dass AssetStorm es versteht, reicht es daher, das Format der Objekte umzustellen und Objekte zusammenzufassen. Dafür iteriert die Flask-Anwendung zunächst über alle Blockelemente. Je nach Typ (Pandoc legt den in der Eigenschaft `'t'` ab) konvertiert es alle Spans (bei Absätzen), sammelt den Text-Inhalt (bei Zitaten) oder parst den Inhalt mit `yaml` (bei „magischen“ Kommentaren – dazu später mehr). Die Grundstruktur dafür ist eine simple Schleife mit Fallunterscheidung:

```
block_assets_list = []
for block in pandoc_tree['blocks']:
  if block['t'] == 'Para':
    paragraph_asset = {
     "type": 'block-paragraph',
     "spans": convert_list(block['c'])
    }
    add_to_asset_list(paragraph_asset)
  elif block['t'] == 'BlockQuote':
    # ...
  elif block['t'] == 'RawBlock':
```

```
    # ...
return block_assets_list
```

In `block['c']` bettet Pandoc bei Absätzen die Liste von Spans ein. Die haben jeweils die gleiche Grundstruktur mit Typ `'t'` und Inhalt `'c'`:

```
[{'t': 'Str', 'c': 'Pina'},
 {'t': 'Space'},
 {'t': 'Str', 'c': 'Programmiert'}]
```

Ausnahmen bilden Leerzeichen und Sonderzeichen, die keinen Inhalt haben. Für sie pflegt das Programm eine Typ-Übersetzungstabelle:

```
CHARACTER_TYPES = {
  "Space": " ",
  "DoubleQuote": '"',
  "SingleQuote": "'",
  "SoftBreak": "\n"
}
```

Da in einem Span für fetten oder kursiven Text andere Spans eingebettet sind, konvertiert das Programm Spans mit einer Funktion, die es rekursiv aufrufen kann, am Ende aber alle konvertierten Spans in einer einzelnen Liste sammelt:

```
def convert_list(span_list,
            span_type="span-regular"):
  spans = []
  for span_element in span_list:
    convert_elem(spans, span_element)
  merge_list(spans)
  return spans
```

Die eigentliche Arbeit erledigt `convert_elem()`, das die Fallunterscheidung nach Typ übernimmt. Die Variable `span_type` stammt aus den Parametern der übergeordneten Funktion `convert_list()`, `convert_elem()` ist als Unterfunktion von dieser definiert. Hier ein Auszug:

```
def convert_elem(spans, span_elem):
  if (span_elem['t'] in
      CHARACTER_TYPES.keys()):
    spans.append(create_span(span_type,
     CHARACTER_TYPES[span_elem['t']]))
    return
  if span_elem['t'] == "Str":
    spans.append(create_span(
     span_type,span_elem['c']))
    return
  if span_elem['t'] in ["Strong","Emph"]:
    spans += convert_list(
     span_elem['c'],
     PANDOC_SPAN_TYPES[span_elem['t']])
    return
```

Im ersten `if`-Block erschließt sich, wozu das zuvor erstellte Dictionary `CHARACTER_TYPES` dient: Es liefert den Text für Sonder- und Leerzeichen-Objekte. Der Typ `"Str"` gibt seinen Inhalt dagegen ganz einfach über den Schlüssel `'c'` preis. Die Funktion `create_span()` erstellt in beiden Fällen ein Dictionary mit den Schlüsseln, die AssetStorm erwartet.

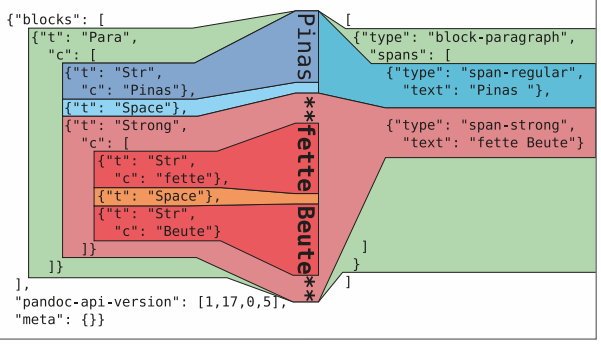
Bei den beiden Hervorhebungen kommt es dagegen zur zuvor erwähnten Rekursion. Die Spans enthalten nämlich ihrerseits wieder eine Liste mit Spans, die `convert_list()` ins Format von AssetStorm übersetzt. Statt des `span_type` `"span-regular"` nutzen die beiden Hervorhebungen aber `"span-strong"` oder `"span-emphasized"`. Dadurch bekommen die von der rekursiv aufgerufenen Funktion erstellten Spans einen anderen `"type"`. Trotzdem landen am Ende alle Spans in einer einzelnen flachen Liste. Das Programm entfernt somit bei Bedarf die Baumstruktur der Spans aus Pandocs internem JSON. Die flache Struktur erschien mir beim Definieren der Asset-Typen in AssetStorm als übersichtlicher und weniger fehleranfällig.

Kompressibler Text

Die so konvertierte Liste enthält des Öfteren Spans mit gleichem Typ direkt hintereinander. Das widerspricht zwar nicht den in AssetStorm definierten Schemas, bläht die Liste aber unnötig auf. Deswegen fasst `merge_list()` die Einträge gleichen Typs zu jeweils einem zusammen:

Markdown zu JSON

Aus dem Satz in Markdown (mitte) erzeugt Pandoc das JSON links. Aus dem erzeugt das Python-Programm JSON im richtigen Format für AssetStorm (rechts)



```
def merge_list(span_list):
  pos = 0
  while len(span_list) > pos+1:
    if (span_list[pos]['type'] ==
        span_list[pos+1]['type']):
      content_key = "text"
      if (span_list[pos]['type'] ==
          "span-listing"):
        content_key = "listing_text"
      pop_item = span_list.pop(pos+1)
      span_list[pos][content_key
            ] += pop_item[content_key]
    else:
      pos += 1
```

YAML-Kommentare

Bildergalerien oder Textkästen kennt Markdown gar nicht. Um sie trotzdem im gleichen Dokument eingeben zu können, musste ich etwas tricksen: Viele Markdown-Parser verstehen HTML-ähnliche Kommentare, die mit `<!---` beginnen und mit `-->` aufhören. Mein Markdown-Editor stellt so eingefasste Blöcke einfach als grauen Text dar. Pandoc versteht diese Syntax auch und gibt den Blöcken den Typ `"RawBlock"`. Den Inhalt eines solchen Blocks lässt Pandoc unangetastet und gibt ihn unter dem Schlüssel `'c'` im Block an.

Mein Programm prüft mit einer Regex, ob der `"RawBlock"` dem Format eines solchen Kommentars entspricht:

```
r"^<!---(?P<yaml>[\s\S]*?)-->$"
```

In der Capturing-Group „yaml“ sammelt sich bei einem Match der eigentliche Inhalt zwischen den Start- und Endzeichen. Der Name „yaml“ verrät schon, wie es mit meinen magischen Kommentaren weitergeht: Ich kippe ihren Inhalt einfach in einen YAML-Parser und übernehme die angegebene Struktur:

```
import yaml
yaml_tree = yaml.safe_load(
          matches.groupdict()['yaml'])
```

Dass Menschen YAML leicht lesen und nur schwerlich Fehler darin einbauen können, kommt der Syntaxerweiterung zugute.

Eine Schwierigkeit stellt sich aber dennoch: Textkästen enthalten Blockelemente mit Spans, die ich nicht als YAML darstellen möchte, sondern lieber als Markdown. Deswegen sucht das Programm bei allen Schlüsseln im YAML nach dem magischen Inhalt `MD_BLOCK`. Findet es den, legt es den aktuellen Block in der Variable `unfinished_block` ab und speichert den magischen Schlüssel, unter dem es die Blockelemente einfügen muss.

Gibt es einen solchen Block ohne Abschluss, liest das Programm hinter dem Block wie üblich Markdown ein, fügt es aber nicht an die Liste der Blöcke an. Stattdessen wartet es auf den nächsten magischen Kommentar, der den Textblock abschließt. So ein magischer Kommentar darf sogar ganz ohne Inhalt sein.

Die Blockelemente zwischen den beiden Kommentaren fügt das Programm dann unter dem Schlüssel in den vom Kommentarpaar definierten Block ein, bei dem es `MD_BLOCK` als Inhalt angegeben hat. Außerdem setzt es `unfinished_block` zurück und liest folgendes Markdown wieder als Blockelemente auf höchster Ebene.

Ein einfacher Textkasten sieht mit dieser Syntax so aus:

```
Erste Zeile.
<!--
type: block-info-box
title: Kastenüberschrift
content: MD_BLOCK
-->
Dieser Text gehört in den Kasten.
Er hat **zwei** Absätze.
<!---->
Text hinter dem Kasten.
```

Markup statt Formular

Mit eigenen Erweiterungen wie meinen magischen Markdown-Kommentaren gibt man ohne Mühe manches ein, wo die Markup-Sprache zu kurz greift. Eigene Parser ersparen dabei manchmal sogar ein Web-Formular oder eine grafische Oberfläche. Denn für viele Anwender sind die Markup-Sprachen samt Erweiterungen leichter zu lernen als die Bedienung verschachtelter Oberflächen – zumindest wenn der Anwender zum Parser auch die passende Doku bekommt.

Programmiert ist so ein Parser schnell, wenn man die schwere Arbeit an Pandoc auslagert. Als Flask-Microservice läuft er irgendwo im Heimnetz und arbeitet mit beliebigen Frontends zusammen. Deswegen integriert sich so eine Lösung auch mit wenig Aufwand in bestehende Systeme zum Schreiben, Dokumentieren und Veröffentlichen.

(pmk) c't

Programm bei GitHub, Dokumentation:
www.ct.de/wphe

API-Calls in Threads in PyQt5

Parallele Anwendungen haben Tücken. Bei Netzwerkanfragen in grafischen Programmen mit PyQt5 kommt man um Threads jedoch kaum herum. Ich habe mir mit einer scheinbar naheliegenden Implementierung selbst ins Knie geschossen. Aber ich zeige auch, wie man es ohne Zusatzaufwand besser macht.

Von Pina Merkert

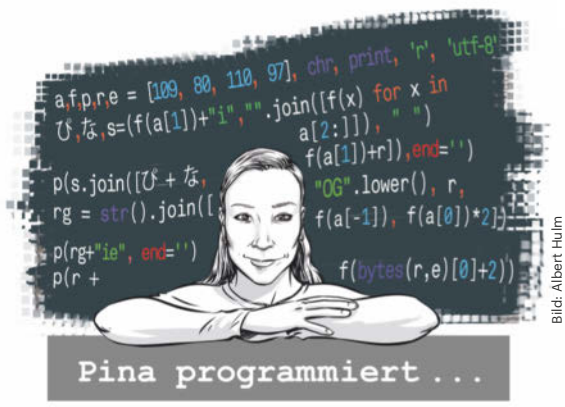


Bild: Albert Hulm

Bei der Android-App zum Kalorien zählen „Lose It!“ fragte ich mich zuletzt, ob sie ihre Daten auch wie beworben mit Google-Fit synchronisiert. Also schrieb ich mir ein Programm mit PyQt5, das Google Fit abfragt und alle Nahrungsmitteleinträge der letzten Woche anzeigt (siehe Seite 42). Für den Zugriff auf das REST-API sind natürlich Netzwerkanfragen nötig und als brave Programmiererin wusste ich: Netzwerkanfragen sollten man nie im Haupt-Thread eines grafischen Programms absetzen, weil diese das Interface blockieren, bis die Antwort da ist.

Zum Glück bringt Python die Klasse `Thread` mit, mit der ich in ein paar Zeilen einen eigenen Thread für den Request definieren kann (siehe linker Kasten auf Seite 177).

In `self.session` speichert die Klasse ein Objekt, das neben der eigentlichen Anfrage auch die Authentifizierung per Token erledigt (siehe Seite 40). `self.url` enthält den API-Endpunkt. Der eigentliche Trick ist `self.callback`, denn das ist eine Funktion, die der Haupt-Thread beim Aufruf übergeben kann. Zumindest kam ich mir trickreich vor, als ich das so programmiert habe.

Den Thread rief das Programm aus der von `QWidget` geerbten Klasse des Haupt-Fensters, die im Haupt-Thread läuft, wie folgt auf:

```
def button_clicked(self):
  network_thread = RequestThread(
    self.session, self.request_url,
    self.data_loaded)
  network_thread.start()
def data_loaded(self, json_data):
  self.nutrient_widget.set_json_data(
    json_data)
```

Fehlerloses Versagen

Ich startete mein Programm und freute mich zunächst, dass in der Konsole keine Fehlermeldungen auftauchten. Nur leider tauchten in meinem Interface auch keine Daten auf.

Trotz heftigem Doku lesen, Ausprobieren, Nachdenken und Haareraufen konnte ich einfach nicht verstehen, warum die Daten nicht im Interface auftauchten. Geladen hatte mein Thread die Daten völlig problemlos und sie kamen auch wie erwartet in der Callback-Funktion an. Das konnte ich mit einfachen `print()`-Anweisungen prüfen. Der Code rief auch die Funktionen zum Aktualisieren des Interface auf. Nur sehen konnte ich davon weiter nichts.

Links die Implementierung, bei der Veränderungen am Interface im falschen Thread versauern. Rechts die Lösung mit Qts Signal-Slot-Mechanismus.

```
class RequestThread(Thread):

  def __init__(self, session, url,
                       callback, *args):
    super(RequestThread, self).__init__()
    self.callback = callback
    self.session = session
    self.url = url
  def run(self):
    response = self.session.request(
      method="GET", url=self.url)
    self.callback(response.json())
    super(RequestThread, self).run()
```

```
class RequestThread(Thread, QObject):
  data_loaded = pyqtSignal(list)
  def __init__(self, session, url,
                                 *args):
    super(RequestThread, self).__init__()
    QObject.__init__(self, *args)
    self.session = session
    self.url = url
  def run(self):
    response = self.session.request(
      method="GET", url=self.url)
    self.data_loaded.emit(response.json())
    super(RequestThread, self).run()
```

Beim Debuggen bemerkte ich eine Fehlermeldung der Qt-Bibliothek: „QObject::setParent: Cannot set parent, new parent is in a different thread“. Die brachte mich zum Grübeln: Auch wenn ich die Callback-Funktion als Methode der Klasse implementierte, die im Haupt-Thread das Interface zeichnete, war dadurch ja nicht sichergestellt, dass die Funktion auch immer in diesem Thread aufgerufen wird. Was wäre, wenn sie im Netzwerk-Thread abliefe? Das würde erklären, warum die Zeichenfunktionen nichts Sichtbares veränderten. Ich hatte aber erwartet, dass mir Qt die Aufrufe um die Ohren haut und mir erklärt, dass es im Netzwerk-Thread keine Widgets gibt. Qt fängt den Fehler aber ab, sodass mein Programm trotz des Fehlers nicht abstürzt und nur die Fehlermeldung auf die Konsole schreibt.

Mehr Eltern helfen

Meinen Fehler zu erkennen war eine ungemeine Erleichterung! Denn von der Erkenntnis, dass die Callback-Funktion im falschen Thread lief, war es nicht mehr weit zur Lösung: Statt mit Callback konnte ich das Ergebnis der Netzwerkanfrage auch über den Signal-Slot-Mechanismus von Qt übermitteln. Der bietet nämlich auch einen Thread-sicheren Modus `Qt.QueuedConnection`.

Damit der Netzwerk-Thread ein `pyqtSignal` abschicken kann, muss er von `QObject` erben. Da Python Mehrfachvererbung unterstützt, ist das aber kein Problem.

Die geänderte Klasse (im Kasten rechts) unterscheidet sich nur in Details: Die zweite Zeile definiert ein Signal `data_loaded`, das die Callback-Funktion ersetzt. Der Konstruktor dieser Klasse muss wegen der Mehrfachvererbung sowohl den Konstruktor von `Thread` als auch den von `QObject` aufrufen. Da die Klasse `Thread` zuerst nennt, konnte ich den Konstruktor von `Thread` mit `super()` aufrufen. Den für `QObject` musste ich dagegen direkt angeben. Zuletzt steht statt des Callback-Aufrufs in `run()` nun `self.data_loaded.emit()`. Der Rest bleibt gleich.

Beim Starten des Netzwerk-Threads ist dagegen eine zusätzliche Zeile nötig, die die Slot-Verbindung zum Signal aufbaut:

```
def button_clicked(self):
  network_thread = RequestThread(
    self.session, self.request_url)
  network_thread.data_loaded.connect(
    self.data_loaded,
    type=Qt.QueuedConnection)
  network_thread.start()
```

Der `type=Qt.QueuedConnection` sorgt nämlich schon dafür, dass sie wie geplant im Haupt-Thread abläuft. Das geht, weil eine `QueuedConnection` die mit dem Slot verbundene Funktion nicht direkt aufruft. Stattdessen schiebt sie das Ereignis in eine Warteschlange, die die von Qt ausgeführte Ereignisschleife dann abholt. Jeder Thread hat dabei eine eigene Warteschlange und Qt sorgt automatisch dafür, dass das Ereignis in der richtigen Schlange landet.

Nach dieser kleinen Änderung lief der Netzwerk-Code wie geschmiert. Beim nächsten `Thread` werde ich früher prüfen, unter welcher Herrschaft welche Funktionen laufen. Die vergebliche Suche in den Anzeigefunktionen von Qt hat mich nämlich unnötig Zeit gekostet. (pmk) c't

Beispielcode bei GitHub, Dokumentation:
www.ct.de/wy77

IMPRESSUM

Redaktion

Postfach 61 04 07, 30604 Hannover
Karl-Wiechert-Allee 10, 30625 Hannover
Telefon: 05 11/53 52-300
Telefax: 05 11/53 52-417
Internet: www.ct-special.de

Leserbriefe und Fragen zum Heft: sonderhefte@ct.de

Die E-Mail-Adressen der Redakteure haben die Form xx@ct.de oder xxx@ct.de. Setzen Sie statt „xx“ oder „xxx“ bitte das Redakteurs-Kürzel ein. Die Kürzel finden Sie am Ende der Artikel und hier im Impressum.

Chefredakteur: Jobst H. Kehrhahn (keh)
(verantwortlich für den Textteil)

Konzeption: Pina Merkert (pmk)

Koordination: Ann-Sophie Eikermann (ase), Ilona Krause (ilk), Angela Meyer (anm)

Redaktion: Anke Brandt (apoi), Ilona Krause (ilk), Jan Mahn (jam), Angela Meyer (anm), Pina Merkert (pmk)

Mitarbeiter dieser Ausgabe: Pit Noack, Nikolaus Schüler, Otis Sotek, Sebastian Stabinger, Bastian Wandt

Assistenz: Susanne Cölle (suc@ct.de), Tim Rittmeier (tir), Christopher Tränkmann (cht), Martin Triadan (mat)

DTP-Produktion: Madlen Grunert, Lisa Hemmerling, Kirsten Last, Steffi Martens, Marei Stade, Matthias Timm, Ricardo Ulbricht, Ninett Wagner

Digitale Produktion: Christine Kreye (Ltg.), Melanie Becker, Anna Hoffmann, Joana Hollasch, Martin Kreft, Dominique Kuhn

Illustration: Rudolf A. Blaha, Frankfurt am Main; Thorsten Hübner, Braunschweig; Albert Hulm, Berlin; Sophia Sanner, Hannover; tsamedien, Düsseldorf

Titel: Steffi Martens

Verlag

Heise Medien GmbH & Co. KG
Postfach 61 04 07, 30604 Hannover
Karl-Wiechert-Allee 10, 30625 Hannover
Telefon: 05 11/53 52-0
Telefax: 05 11/53 52-129
Internet: www.heise.de

Herausgeber: Christian Heise, Ansgar Heise, Christian Persson

Geschäftsführer: Ansgar Heise, Dr. Alfons Schräder

Mitglieder der Geschäftsleitung: Beate Gerold, Jörg Mühle

Verlagsleiter: Dr. Alfons Schräder

Anzeigenleitung: Michael Hanke (-167)
(verantwortlich für den Anzeigenteil),
www.heise.de/mediadaten/ct

Anzeigenverkauf: Verlagsbüro ID GmbH & Co. KG, Tel.: 05 11/61 65 95-0, www.verlagsbuero-id.de

Leiter Vertrieb und Marketing: André Lux (-299)

Service Sonderdrucke: Julia Conrades (-156)

Druck: Firmengruppe APPL Druck GmbH & Co. KG, Senefelder Str. 3–11, 86650 Wemding

Vertrieb Einzelverkauf:
VU Verlagsunion KG
Meßberg 1
20086 Hamburg
Tel.: 040/3019 1800, Fax: 040/3019 145 1800
E-Mail: info@verlagsunion.de

Einzelpreis: € 14,90; Schweiz CHF 27.90; Österreich € 16,40; Luxemburg € 17,10

Erstverkaufstag: 19.05.2020

ONLINE-KONFERENZ 9. Juni | 18. Juni

heise Security Tour 2020

Wissen schützt

Was tun, wenn's brennt – richtige Vorbereitung ist mehr als die halbe Miete

Auszug aus dem Programm

- IT-Security im Überblick – damit haben wir es akut zu tun, das kommt auf uns zu
  (Jürgen Schmidt)
- IT-Sicherheitsgesetz 2.0 – Grundlagen und Umsetzung in der Praxis
  (Wilhelm Dolle / Christoph Wegener)
- Notfall-Planung – so bereiten Sie sich und Ihre Kollegen richtig vor
  (Manuel Atug / Lukas Reike-Kunze)
- Anatomie eines Datenschutz-GAUs
  (Joerg Heidrich)
- Monitoring und Einbruchserkennung – ein Überblick zu Markt & Techniken
  (Steffen Gundel / Stefan Strobel)
- Forensik und Incident Response: Möglichkeiten und Grenzen der Spurensuche
  (Björn Schemberger)

Richtig Vorbeugen – Sinnvoll Eingreifen – Aus Erfahrung lernen

www.heise-events.de/securitytour

Partner: F-Secure

Eine Veranstaltung von: heise Security

Organisiert von: heise Events

Für einen erweiterten Horizont:

NEU

c't Android 2020

Das Rundum-Android-Paket auf über 150 Seiten: die Auswahl des perfekten Kamera-Smartphones, Tests günstiger Android-Geräte und Falt-Smartphones oder Öko-Handys.
Außerdem: halten Sie Ihr Android-Phone sauber und sicher und richten Sie Smartphones absolut kindersicher ein.

Auch komplett digital erhältlich!

shop.heise.de/android20

12,90 € ›

c't Projekte 2019

c't Projekte 2019 führt in die Welt der Einplatinencomputer ein, stellt Plattformen vor und vermittelt Know-How für anspruchsvollere Projekte. In zahlreichen Bau- und Programmiervorschlägen finden Einsteiger wie Fortgeschrittene Anleitungen zum Nachbau und Anregungen für eigene Ideen.

Auch komplett digital erhältlich!

shop.heise.de/ct-projekte19

12,90 € ›

c't Admin

IT-Praxis für Heim- und Büronetzwerke
Das Sonderheft unterstützt bei Themen wie Windows-Einrichtung und -Vernetzung, Server-Administration und Server-Wahl, LAN-Aufrüstungen ohne neue Kabel oder auch Router-Optimierungen.

Auch komplett digital verfügbar.

shop.heise.de/ct-admin19

12,90 € ›

Weitere Sonderhefte zu vielen spannenden Themen finden Sie hier: shop.heise.de/specials-aktuell

› Generell portofreie Lieferung für Heise Medien- oder Maker Media Zeitschriften-Abonnenten oder ab einem Einkaufswert von 15 €. Nur solange der Vorrat reicht. Preisänderungen vorbehalten.

heise shop
shop.heise.de/specials-aktuell ›