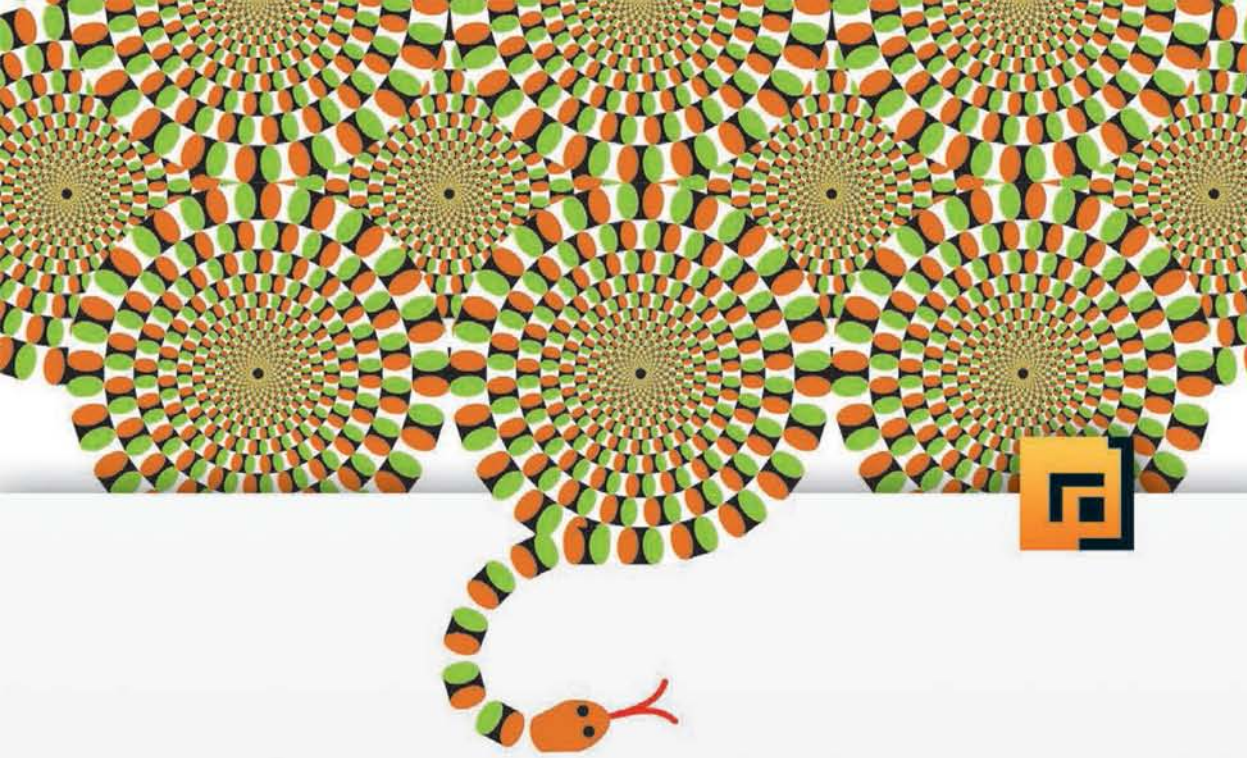


Jason R. Briggs

Python kinderleicht!

Einfach programmieren lernen – nicht nur für Kids

US-Bestseller

dpunkt.verlag

Jason Briggs

Python kinderleicht!

Einfach programmieren lernen – nicht nur für Kids

Übersetzung aus dem Amerikanischen
von Volker Haxsen

dpunkt.verlag

Lektorat: Dr. Michael Barabas
Übersetzung: Volker Haxsen, Heidelberg
Copy-Editing: Friederike Daenecke, Zülpich
Herstellung: Birgit Bäuerlein
Illustrationen: Miran Lipovača
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN
Buch 978-3-86490-022-8
PDF 978-3-86491-332-7
ePub 978-3-86491-333-4

1. Auflage 2013

Ringstraße 19 B
69115 Heidelberg

Titel der Originalausgabe: Python For Kids – A Playful Introduction To Programming
No Starch Press, Inc. · 38 Ringold Street, San Francisco, CA 94103 · http://www.nostarch.com/
ISBN 978-1-59327-407-8

5 4 3 2 1 0

Vorwort

Über den Autor

Jason R. Briggs ist seit dem Alter von acht Jahren Programmierer und hat als erste Programmiersprache BASIC auf einem Radio Shack TRS-80 erlernt. Er hat als Entwickler und Systemarchitekt professionell Software programmiert und als Autor für das *Java Developers's Journal* gearbeitet. Seine Artikel sind in *JavaWorld*, *ONJava* und *ONLamp* erschienen. *Python kinderleicht* ist sein erstes Buch.

Du kannst mit Jason über seine Homepage *http://jasonbriggs.com/* oder per E-Mail *mail@jasonbriggs.com* Kontakt aufnehmen.

Über die Fachkorrektoren

Der 15-jährige **Josh Pollock** ist frischgebackener Absolvent der *The Nueva School* und jetzt neu auf der *Lick-Wilmerding High School* in San Francisco. Er fing im Alter von neun Jahren mit dem Programmieren in Scratch an, begann in der sechsten Klasse mit TI-BASIC, ging dann in der siebten Klasse zu Java und Python über und machte in der achten Klasse mit UnityScript weiter. Neben dem Programmieren spielt er Trompete, entwickelt Computerspiele und unterrichtet Leute in MINT-Fächern.

Maria Fernandez hat einen Masterabschluss in angewandter Linguistik und interessiert sich schon seit über 20 Jahren für Computer und Technik. Sie hat jungen Flüchtlingsfrauen im *Global-Village*-Projekt in Georgia (USA) Englisch beige-

bracht, lebt zurzeit in Nord-Kalifornien und arbeitet für den ETS (Educational Testing Service).

Danksagungen

So ungefähr muss es sein, wenn man beim Empfang einer Ehrung die Bühne betritt und dann feststellt, dass man die Liste der Personen zu Hause hat liegen lassen, die man bei seiner Danksagung berücksichtigen will: Man vergisst garantiert jemanden, und die Musik setzt ganz schnell ein, um einen von der Bühne herunterzukomplimentieren.

Deswegen kommt jetzt eine (zweifelsohne) unvollständige Liste von Leuten, denen ich zu tiefem Dank verpflichtet bin, da sie mir geholfen haben, das Buch so gut werden zu lassen, wie es jetzt ist.

Ich möchte dem Team von No Starch danken, vor allem Bill Pollock, für seine bei der Bearbeitung immer wieder gestellte Frage, was denn ein Kind von alldem halten würde.

Wenn man schon sehr lange programmiert, vergisst man nur allzu leicht, wie schwer diese Dinge für Anfänger sind, und Bill war eine wertvolle Hilfe, weil er mich auf diese oft übersehenen und überkomplizierten Passagen aufmerksam machte. Mein Dank gilt auch Serena Yang, der exzellenten Produktionsmanagerin. Ich hoffe, dass sie sich nicht allzu sehr die Haare gerauft hat, als sie die richtige Farbgebung des Codes auf über 300 Seiten überprüfen musste.

Ein großes Dankeschön geht an Miran Lipovaca für ihre überaus gelungenen Illustrationen. Sie sind viel mehr als nur gelungen. Nein ehrlich! Wenn ich das gemacht hätte, könnte man von Glück sagen, wenn man ab und zu eine hingeschmierte Figur erkennen könnte. Ist es ein Bär? Ist es ein Hund? Nein, warte … soll das ein Baum sein?

Vielen Dank den Korrektoren! Ich muss mich dafür entschuldigen, dass nicht alle Vorschläge am Ende berücksichtigt wurden. Wahrscheinlich hattet Ihr recht, und ich kann nur eine schlechte Charaktereigenschaft von mir dafür verantwortlich machen, falls noch Fehler enthalten sind. Besonderer Dank geht an Josh für einige wirklich tolle Vorschläge und Ideen. Mein Bedauern gilt Maria, weil sie sich mit zum Teil uneinheitlich formatiertem Code herumschlagen musste.

Ich danke meiner Frau und meiner Tochter dafür, dass sie sich mit einem Mann und Vater abfinden mussten, der sich noch mehr als sonst hinter dem Computerbildschirm versteckt hat.

Meiner Mutter danke ich für all die unermüdliche Aufmunterung über all die Jahre.

Und zu guter Letzt danke ich meinem Vater dafür, dass er sich damals in den 1970er-Jahren einen Computer gekauft hat und es ertragen hat, dass ich diesen genauso oft nutzen wollte wie er. Nichts von alledem wäre ohne ihn möglich gewesen.

Inhaltsübersicht

Inhaltsverzeichnis

1 Einleitung

Warum soll man das Programmieren erlernen?

Programmieren fördert die Kreativität, das logische Denken und die Fähigkeit, Probleme zu lösen. Programmierer und Programmiererinnen haben die Möglichkeit, etwas aus dem Nichts zu erschaffen. Mithilfe der Logik bringen sie Programmstrukturen in eine Form, sodass ein Computer damit funktioniert. Und wenn die Dinge nicht ganz so gut funktionieren wie erwartet, können sie durch die Fähigkeit zur Problemlösung herausfinden, was schiefgelaufen ist. Programmieren macht Spaß, ist manchmal schwierig (gelegentlich frustrierend), und die Fähigkeiten, die man dabei erwirbt, können sowohl in der Schule als auch bei der Arbeit nützlich sein – selbst wenn Dein Berufsleben später nichts mit Computern zu tun haben sollte.

Außerdem ist das Programmieren ein prima Zeitvertreib bei miesem Wetter.

1.1 Warum Python?

Python ist eine leicht zu erlernende Programmiersprache, die für den Programmieranfänger einige nützliche Eigenschaften hat. Der Code ist im Vergleich zu anderen Programmiersprachen recht einfach zu lesen, und es gibt eine interaktive Shell, in die man seine Programme eingeben und sehen kann, wie sie laufen. Zusätzlich zu seiner einfachen Programmstruktur und seiner interaktiven Shell hat Python einige Merkmale, die den Lernvorgang sehr bereichern und mit denen Du einfache Animationen zum Erstellen Deiner eigenen Spiele zusammenbauen

kannst. Eines davon ist das Modul `turtle`, das von Turtle Graphics inspiriert wurde (das in den 1960er-Jahren von der Programmiersprache Logo verwendet wurde) und für Lernzwecke geschaffen wurde. Ein weiteres Modul ist `tkinter`, mit dem man auf das Tk GUI Toolkit zugreifen kann, um damit ziemlich einfach ein bisschen anspruchsvollere Grafiken und Animationen zu erstellen.

1.2 Wie man das Programmieren lernt

Wie bei allem, was man zum ersten Mal probiert, ist es am besten, mit den Grundlagen anzufangen. Beginne daher mit den ersten Kapiteln, und blättere nicht voller Ungeduld zu den Kapiteln weiter hinten. Niemand kann beim ersten Mal, wenn er ein Musikinstrument in die Hand nimmt, im Sinfonieorchester mitspielen. Flugschüler fliegen auch nicht, bevor sie die grundlegenden Steuerelemente verstanden haben, und Turner kriegen (normalerweise) beim ersten Versuch keinen Salto rückwärts hin. Wenn Du zu Anfang zu ungeduldig bist, haben die grundlegenden Prinzipien keine Zeit, sich richtig in Deinem Kopf festzusetzen. Dir wird dann der Inhalt der Kapitel weiter hinten viel komplizierter vorkommen, als er in Wirklichkeit ist.

Während Du dieses Buch durchliest, solltest Du jedes Beispiel selbst ausprobieren, um zu sehen, wie es funktioniert. Am Ende der meisten Kapitel gibt es auch Programmier-Puzzles, die Du lösen kannst. Sie werden Deine Programmierfähigkeiten fördern. Denke immer daran: Je besser Du die Grundlagen verstanden hast, desto leichter werden Dir die komplizierteren Konzepte später vorkommen.

Wenn Dich etwas frustriert oder Dir zu schwierig vorkommt, hier ein paar Ratschläge, die ich sehr hilfreich finde:

- Teile das Problem in kleinere Teile auf. Versuche zu verstehen, was ein kleiner Teil des Codes macht, oder denke nur an einen kleinen Teil einer komplexen Stelle. (Konzentriere Dich lieber auf einen kleinen Teil des Codes, statt alles auf einmal verstehen zu wollen.)
- Wenn das alles nichts hilft, ist es manchmal am besten, wenn man es für eine Weile einfach liegen lässt. Schlafe drüber, und mache an einem anderen Tag weiter. Auf diese Weise lösen sich viele Probleme von allein – besonders Programmierprobleme.

1.3 Wer dieses Buch lesen sollte

Dieses Buch ist für jeden geschrieben, der sich für das Programmieren interessiert, ganz egal, ob man nun Kind oder Erwachsener ist, wenn man zum ersten Mal programmiert. Wenn man lernen will, wie man seine eigene Software schreibt, anstatt nur von anderen entwickelte Programme zu nutzen, ist *Python kinderleicht* ein toller Einstieg.

In den folgenden Kapiteln erfährst Du, wie man Python installiert, die Python-Shell startet, einfache Berechnungen anstellt, Text auf den Bildschirm bekommt und Listen erstellt. Du lernst, wie man einfache Fallunterscheidungen mit `if`-Anweisungen und `for`-Schleifen durchführt. (Und natürlich erfährst Du, was `if`-Anweisungen und `for`-Schleifen eigentlich sind!) Du erfährst, wie man Code mit Funktionen wiederverwendet. Du lernst die Grundlagen von Klassen und Objekten kennen und bekommst Beschreibungen der vielen in Python eingebauten Funktionen und Module.

Es gibt Kapitel über einfache und fortgeschrittene Turtle-Grafiken und über die Benutzung des Moduls `tkinter`, um auf dem Computerbildschirm zu zeichnen. Am Ende vieler Kapitel gibt es Programmier-Puzzles mit unterschiedlichen Schwierigkeitsgraden, die dabei helfen, das gerade Gelernte zu verfestigen. Sie bieten Dir auch die Möglichkeit, selbst kleine Programme zu schreiben.

Wenn Du Dir die Grundlagen des Programmierens angeeignet hast, wirst Du lernen, wie Du Deine eigenen Spiele schreiben kannst. Du wirst zwei grafische Spiele entwickeln und etwas über Kollisionsdetektion, Events und diverse Animationstechniken erfahren.

Die meisten Beispiele in diesem Buch benutzen die IDLE-Shell (*Integrated DeveLopment Environment*; integrierte Entwicklungsumgebung) von Python. IDLE bietet Syntax-Markierung, eine Kopieren- und Einfügen-Funktionalität (so, wie Du es von anderen Anwendungen kennst) und ein Editor-Fenster, in dem Du Deinen Code für den späteren Gebrauch speichern kannst. IDLE ist daher eine Entwicklungsumgebung zum Experimentieren und hat auch ein bisschen was von einem Text-Editor. Die Beispiele funktionieren genauso gut in der Standard-Konsole und in einem üblichen Text-Editor, aber die Syntax-Markierung und die benutzerfreundlichere Umgebung von IDLE helfen Dir, den Code schneller zu verstehen. Deshalb wird im ersten Kapitel erklärt, wie man IDLE einrichtet.

1.4 Was in diesem Buch steht

Hier ist ein kurzer Überblick, was Dich in den einzelnen Kapiteln erwartet:

- **Kapitel 2** ist eine Einführung in das Programmieren. Außerdem findest Du Anleitungen zur ersten Installation von Python.
- **Kapitel 3** führt einfache Berechnungen und Variablen ein.
- **Kapitel 4** erklärt einige der grundlegenden Python-Elemente, wie etwa Strings, Listen und Tupel.
- **Kapitel 5** bietet Dir einen Vorgeschmack auf das Modul `turtle`. Wir springen dabei von den Grundlagen des Programmierens zum Bewegen einer Schildkröte (engl. *turtle*, die aber hier die Form eines Pfeils hat) über den Bildschirm.
- **Kapitel 6** behandelt die Varianten der Bedingungen und `if`-Anweisungen, und **Kapitel 7** macht bei den `for`- und `while`-Schleifen weiter.

- In **Kapitel 8** beginnen wir mit der Benutzung und Erstellung von Funktionen, und in **Kapitel 9** geht es um Klassen und Objekte. Wir decken in diesen beiden Kapiteln so viel von den grundsätzlichen Prinzipien der Programmiertechniken ab, dass wir in den weiteren Kapiteln zur Spiele-Entwicklung übergehen können. Von dort an wird es ein bisschen komplizierter.
- **Kapitel 10** stellt die meisten der eingebauten Funktionen von Python vor, und **Kapitel 11** macht mit ein paar Modulen (die im Prinzip Behälter voller nützlicher Funktionalität sind) weiter, die automatisch mit Python installiert wurden.
- **Kapitel 12** kehrt zum `turtle`-Modul zurück, da Du jetzt lernst, mit komplexeren Formen umzugehen. **Kapitel 13** geht zum Modul `tkinter` über – und damit zu fortgeschritteneren grafischen Kreationen.
- In den **Kapiteln 14** und **15** programmieren wir unser erstes Spiel, »Bounce!«, das auf dem Erlernten aus den vorigen Kapiteln aufbaut.
- In den **Kapiteln 16** bis **19** programmieren wir unser zweites Spiel: »Mr. Stick – Man rennt zum Ausgang.« In den Spieleentwicklungs-Kapiteln können die Dinge aus dem Ruder laufen. Wenn nichts mehr geht, lädst Du den Code von der Website zu diesem Buch (*www.dpunkt.de/python)* herunter und vergleichst Deinen Code mit den funktionierenden Beispielen von dort.
- Im **Nachwort** fassen wir das Gelernte mit einem Blick auf PyGame und andere beliebte Programmiersprachen zusammen.
- Zum Schluss sind im **Anhang** noch einmal alle Python-Schlüsselwörter genau erklärt, und im **Glossar** findest Du alle Definitionen der Programmierbegriffe, die in diesem Buch verwendet werden.

1.5 Die Website zum Buch

Wenn Du meinst, dass Du während des Lesens Hilfe brauchst, kannst Du die Website *www.dpunkt.de/python* aufsuchen, wo Du Downloads für alle Beispiele in diesem Buch und noch mehr Programmier-Puzzles findest. Du findest dort auch die Lösungen für alle Programmier-Puzzles in diesem Buch, falls Du nicht mehr weiter weißt oder Deine Programme überprüfen möchtest.

1.6 Viel Vergnügen!

Vergiss beim Durcharbeiten dieses Buches nie, dass Programmieren Spaß machen kann. Sieh es nicht als Arbeit an: Das Programmieren ist eine Möglichkeit, lustige Spiele oder Anwendungen zu erzeugen, die Du mit Deinen Freunden oder anderen teilen kannst.

Programmieren zu lernen ist ein tolles Training fürs Gehirn, und die Ergebnisse können sehr bereichernd sein. Aber vor allem gilt: Egal was Du tust, hab Spaß dabei!

Teil I

Programmieren lernen

2
Nicht alle Schlangen schlängeln sich

Ein Computerprogramm ist eine Gruppe von Anweisungen, die einen Computer dazu bringen, irgendetwas Bestimmtes zu machen. Es geht uns hier nicht um die physischen Bestandteile eines Computers, also die Drähte, Mikrochips, Karten, die Festplatte usw., sondern um die verborgenen Dinge, die auf dieser Hardware laufen. Ein Computerprogramm, das ich meist einfach nur *Programm* nenne, ist diese Gruppe von Befehlen, die der dummen Hardware sagt, was sie zu tun hat. Die *Software* ist eine Sammlung von Computerprogrammen.

Ohne Computerprogramme würde fast jedes Gerät, das wir täglich nutzen, entweder gar nicht funktionieren oder wäre weit weniger nützlich. In der einen oder anderen Form steuern Computerprogramme nicht nur Deinen Computer, sondern auch Videospiele, Mobiltelefone und Navigationsgeräte in Autos. Auch bei weniger offensichtlichen Dingen wie Flachbild-Fernsehern und deren Fernbedienungen sowie modernen Radios, DVD-Playern, Herden und einigen Kühlschränken übernimmt eine Software die Steuerung. Sogar Automotoren, Ampeln, die Straßenbeleuchtung, Zugsignale, elektronische Anzeigetafeln und Aufzüge werden von Programmen geregelt.

Programme sind ein bisschen wie Gedanken. Wenn Du keine Gedanken hättest, würdest Du wahrscheinlich nur auf dem Boden sitzen und auf Dein T-Shirt sabbern. Dein Gedanke »Stehe auf!« ist eine Anweisung, die Deinem Körper sagt, dass er aufstehen soll. Genauso sagen Computerprogramme dem Computer, was er zu tun hat.

Wenn Du weißt, wie man Computerprogramme schreibt, kannst Du allerlei nützliche Dinge anstellen. Sicherlich kannst Du dann nicht direkt Programme schreiben, die Autos, Ampeln oder Deinen Kühlschrank steuern, aber Du kannst damit Webseiten erzeugen, Deine eigenen Spiele programmieren oder Dir ein Programm schreiben, das Dir bei den Hausaufgaben hilft.

2.1 Ein paar Bemerkungen zum Thema Sprache

Wie wir Menschen auch benutzen Computer verschiedene Sprachen, um zu kommunizieren – nämlich Programmiersprachen. Eine *Programmiersprache* ist einfach eine bestimmte Art, mit dem Computer zu reden – eine Art, Anweisungen zu benutzen, die sowohl der Mensch als auch der Computer verstehen.

Es gibt Programmiersprachen, die nach Leuten benannt wurden (z.B. Ada und Pascal), solche, die Abkürzungen darstellen (z.B. BASIC und FORTRAN) und sogar solche, die wie Python nach Fernsehsendungen benannt wurden.

Ja, die Programmiersprache Python wurde nach der Sendung *Monty Python's Flying Circus* benannt und nicht nach der Python-Schlange.

Achtung!

Monty Python's Flying Circus war eine britische Comedy-Sendung, die in den 1970er-Jahren das erste Mal ausgestrahlt wurde. Sie ist bis heute bei einigen sehr beliebt. Die Show enthielt Sketche wie »The Ministry of Silly Walks«, »The Fish-Slapping Dance« und »The Cheese Shop« (in dem überhaupt kein Käse verkauft wurde). Mittlerweile sind Monty Python wohl durch ihre Spielfilme »Das Leben des Brian« und »Die Ritter der Kokosnuss« bekannter.

Eine ganze Reihe von Eigenschaften der Programmiersprache Python machen sie für Anfänger besonders geeignet. Die wichtigste Eigenschaft ist, dass Du mit Python ziemlich schnell einfache, aber wirkungsvolle Programme schreiben kannst. Python verwendet nicht viele dieser komplizierten Zeichen, wie geschweifte Klammern ({ }), Doppelkreuze (#) oder Dollarzeichen($), die andere Programmiersprachen viel schwerer zu lesen machen und daher auf Anfänger abschreckend wirken.

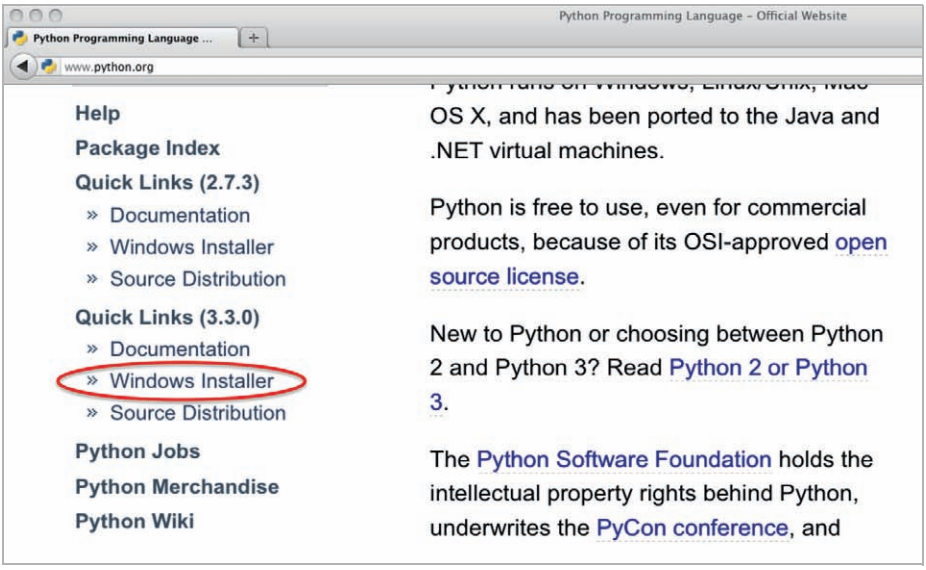
2.2 Python installieren

Die Installation von Python ist sehr unkompliziert. Wir gehen hier die Schritte der Installation in Windows 7, MacOSX und Ubuntu durch. Beim Installieren von Python legst Du Dir auch eine Verknüpfung zum Programm IDLE an. Das ist die integrierte Entwicklungsumgebung, in der Du später Deine Programme schreiben kannst.

Falls Python schon auf Deinem Computer installiert ist, kannst Du zu Abschnitt 2.3, weiterblättern.

Python unter Windows 7 installieren

Um Python für Microsoft Windows zu installieren, gehst Du mit Deinem Browser auf *http://www.python.org* und lädst Dir den aktuellen Windows-Installer für Python 3 herunter. Suche nach dem Abschnitt im Menü, der sich **Quick Links** nennt:



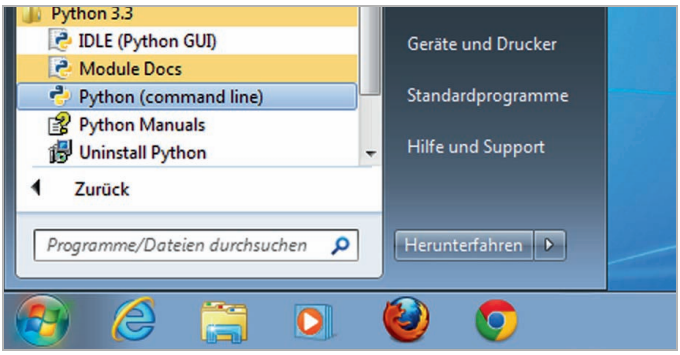
Achtung!

Welche Python-Version genau Du herunterlädst, ist nicht entscheidend, solange vorne eine 3 steht.

Nachdem Du den Windows-Installer heruntergeladen hast, machst Du einen Doppelklick auf sein Icon und folgst dann den Anweisungen, um Python an seinem voreingestellten Speicherort wie folgt zu installieren:

1. Wähle **Install for all Users**, und klicke unten auf **Next.**
2. Lasse den eingestellten Pfad so, wie er ist, notiere Dir aber den Namen des Installationpfades (vermutlich *C:\Python32* oder *C:\Python33)*. Klicke auf **Next.**

Am Ende dieses Vorgangs solltest Du einen Python-3-Eintrag in Deinem Start-Menü haben:

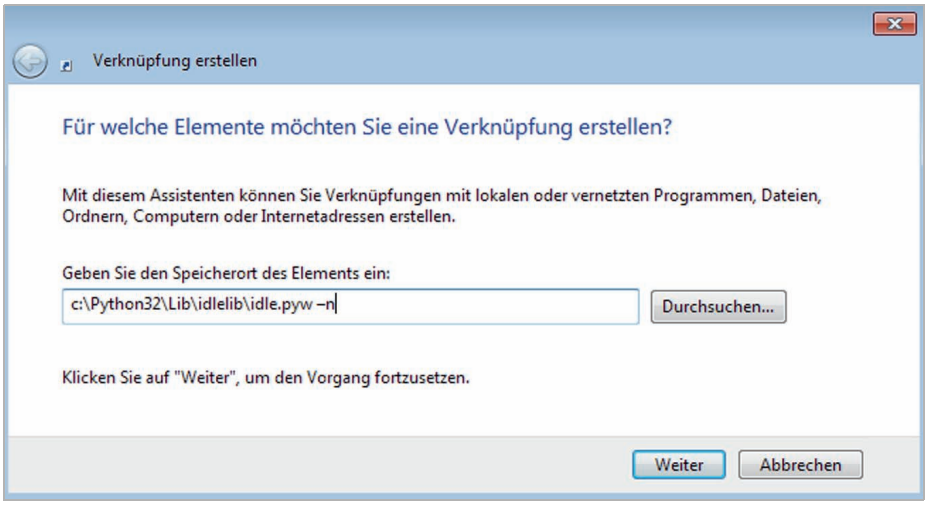


Als Nächstes machst Du Folgendes, um Dir eine Verknüpfung auf dem Desktop anzulegen:

1. Mache einen Rechtsklick auf Deinem Desktop, und wähle im Kontextmenü **Neu ▶ Verknüpfung**.
2. Im nun folgenden Dialogfenster, wo es heißt **Geben Sie den Speicherort des Elementes ein** (achte darauf, dass der Pfad der gleiche ist, den Du vorher notiert hast) gibst Du Folgendes ein:

   ```
   c:\Python32\Lib\idlelib\idle.pyw –n
   ```

   Das Dialogfenster sollte jetzt so aussehen:



3. Klicke auf **Weiter**, um zum nächsten Dialogfenster zu gelangen.
4. Als Namen gibst Du *IDLE* ein und klickst auf **Fertig stellen**, um die Verknüpfung zu erstellen.

Jetzt kannst Du zu »Wenn Du Python installiert hast« auf Seite 10 weiterblättern.

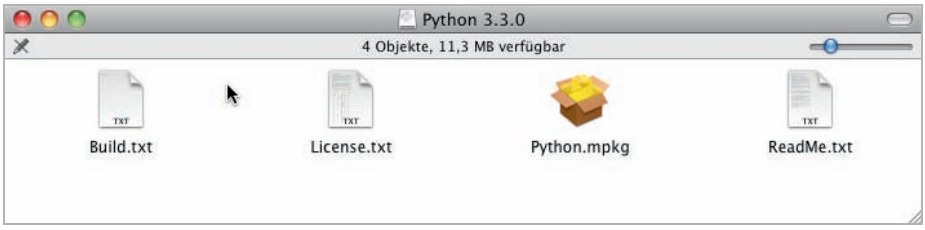
Python in MacOSX installieren

Falls Du einen Mac benutzt, solltest Du bereits eine Version von Python vorfinden. Dabei handelt es sich aber wahrscheinlich um eine ältere Version. Um ganz sicherzugehen, dass Du die aktuelle Version hast, gehst Du mit Deinem Browser auf *http://www.python.org/getit/* und lädst Dir den aktuellen Installer für Mac herunter.

Es gibt dort zwei verschiedene Installer. Welchen Du herunterladen solltest, hängt von der MacOSX-Version ab, die Du benutzt (um das herauszufinden, klickst Du in der obersten Menüleiste auf das Apple-Symbol und gehst auf **Über diesen Mac**.) Wähle dann wie folgt den Installer:

- Wenn Du eine MacOSX-Version zwischen 10.3 und 10.6 hast, lädst Du die 32-Bit-Version von Python 3 für i386/PPC herunter.
- Wenn Du die MacOSX-Version 10.6 oder eine höhere hast, lädst Du die 64-Bit/32-Bit-Version von Python 3 für x86-64/i386 herunter.

Sobald Du die Datei heruntergeladen hast (sie wird das Suffix *.dmg* haben), machst Du einen Doppelklick darauf. Danach siehst Du ein Fenster mit den Inhalten dieser Datei.

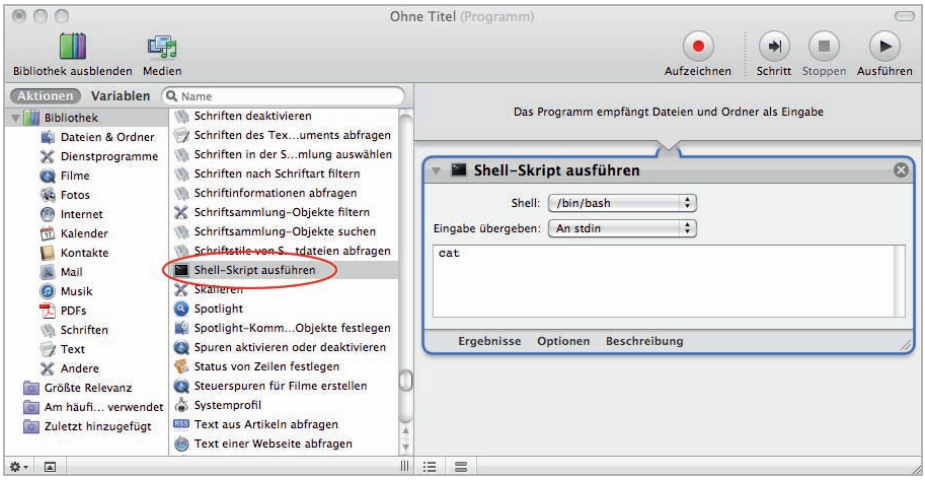


In diesem Fenster doppelklickst Du auf *Python.mpkg* und folgst dann den Anweisungen beim Installieren der Software. Du wirst aufgefordert, Dein Administrator-Kennwort einzugeben, bevor sich Python installiert. (Du hast kein Administrator-Kennwort? Dann müssen es vielleicht Deine Eltern eingeben.)

Als Nächstes musst Du ein Skript zum Desktop hinzufügen, um Pythons IDLE-Anwendung zu starten:

1. Klicke auf das Spotlight-Icon, die kleine Lupe ganz oben rechts in der Ecke des Bildschirms.
2. In die eingeblendete Zeile gibst Du *Automator* ein.
3. Klicke auf die Anwendung, die wie ein Roboter aussieht, sobald sie im Menü auftaucht. Sie befindet sich entweder im Abschnitt **Top-Treffer** oder unter **Programme**.
4. Sobald der Automator geöffnet ist, wähle die Vorlage **Programm**.

5. Klicke auf **Auswählen**, um weiterzugehen.
6. In der Liste der Aktionen suchst Du nach **Shell-Skript ausführen** und bewegst es dann auf die leere Fläche rechts. Das sollte in etwa so aussehen:



7. Im Textfeld siehst Du das Wort *cat*. Markiere das Wort, und ersetze es durch den folgenden Text (alles von `open` bis `–n`, und Du musst vielleicht den Pfad je nach der Version von Python, die Du installiert hast, ändern):

```
open -a "/Applications/Python 3.3/IDLE.app" --args -n
```

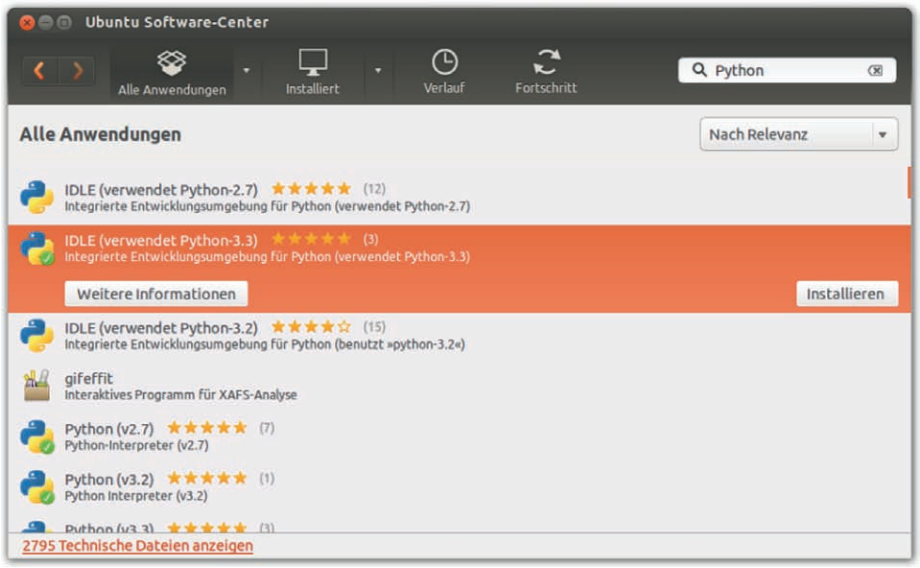
8. Gehe auf **Datei ▶ Speichern**, und gib *IDLE* als Dateinamen an.
9. Wähle im Speicherdialog als Ort den Schreibtisch, und klicke auf **Sichern.**

Jetzt kannst Du zu Seite 10, »Wenn Du Python installiert hast«, gehen und mit Python loslegen.

Python in Ubuntu installieren

Python ist bei der Ubuntu-Distribution schon vorinstalliert, aber es könnte sich dabei um eine ältere Version handeln. Um Python 3 in Ubuntu 12.x zu installieren, führe folgende Schritte durch:

1. Klicke in der Seitenleiste auf den Button für das Ubuntu-Software-Center. (Das ist das Icon, das wie eine orangefarbene Tasche aussieht – falls Du es nicht siehst, kannst Du auch auf den Dash-Startseite-Button klicken und in das Suchfeld *Software* eingeben.)
2. Gib im Suchfeld ganz oben rechts im Software-Center *Python ein.*
3. In der Liste der angebotenen Software wählst Du die aktuelle Version von IDLE, also in diesem Fall *IDLE (using Python 3.3),* aus.



4. Klicke auf **Installieren**.
5. Um die Software zu installieren, gibst Du Dein Administrator-Passwort ein und klickst dann auf **Authentifizieren**. (Du hast kein Administrator-Kennwort? Dann müssen es vielleicht Deine Eltern eingeben.)

Achtung!

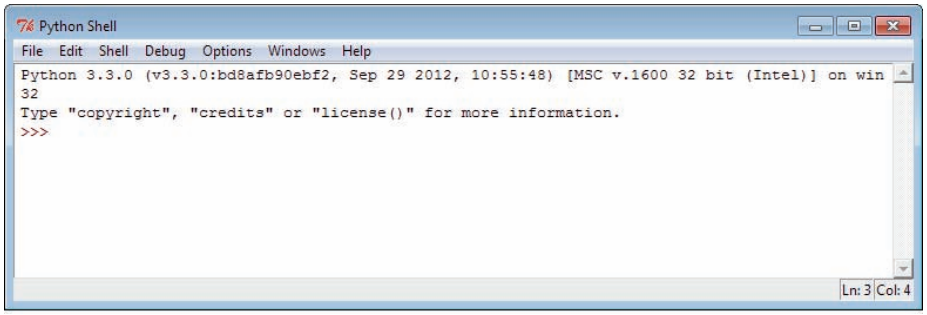
Bei einigen Ubuntu-Versionen siehst Du vielleicht nur Python (v.3.3) im Hauptmenü (statt IDLE). Dies kannst Du dann stattdessen installieren.

Jetzt, da Du die aktuelle Version von Python installiert hast, wollen wir es einmal ausprobieren.

2.3 Wenn Du Python installiert hast

Jetzt solltest Du ein Icon auf Deinem Windows- oder MacOSX-Schreibtisch respektive Desktop haben, das mit **IDLE** beschriftet ist. Wenn Du Ubuntu nutzt, solltest Du auf der **Dash-Startseite** unter *Anwendungen* **IDLE (using Python 3.3)** (oder eine spätere Version) finden.

Mache einen Doppelklick auf das Icon, oder wähle die Menü-Option. Danach sollte dieses Fenster erscheinen:



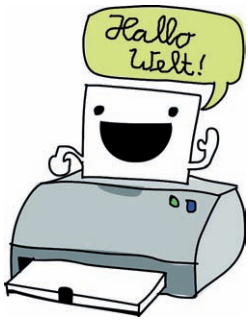
Dies ist die Python-Shell, die zur integrierten Enwicklungsumgebung von Python gehört. Die drei Größer-als-Zeichen (>>>) nennt man den *Prompt*.

Lasst uns nun einige Befehle hinter dem Prompt eingeben. Wir fangen mit diesem hier an:

```
>>> print("Hallo Welt!")
```

Achte darauf, dass Du die beiden Anführungsstriche oben (" ") mit eingibst. Drücke dann auf die ENTER-Taste auf Deiner Tastatur. Wenn Du den Befehl korrekt eingegeben hast, solltest Du so etwas sehen:

```
>>> print("Hallo Welt!")
Hallo Welt!
>>>
```



Der Prompt sollte danach wieder erscheinen, damit Du weißt, dass Python wieder bereit ist, weitere Befehle zu empfangen.

Glückwunsch! Du hast soeben Dein erstes Python-Programm geschrieben. Das Wort `print` gehört zu der Gruppe von Python-Befehlen, die man *Funktionen* nennt. Die Funktion `print` gibt alles auf dem Bildschirm aus, was zwischen den Anführungsstrichen steht. Du hast also dem Computer gesagt, dass er die Worte »Hallo Welt!« anzeigen soll – eine Anweisung also, die Du genauso verstehst wie auch der Computer.

2.4 Deine Python-Programme sichern

Python-Programme wären nicht sehr nützlich, wenn man sie jedes Mal wieder neu schreiben müsste, wenn man sie benutzen möchte. Außerdem müsste man sie auch noch ausdrucken, um eine Vorlage fürs nächste Mal zu haben.

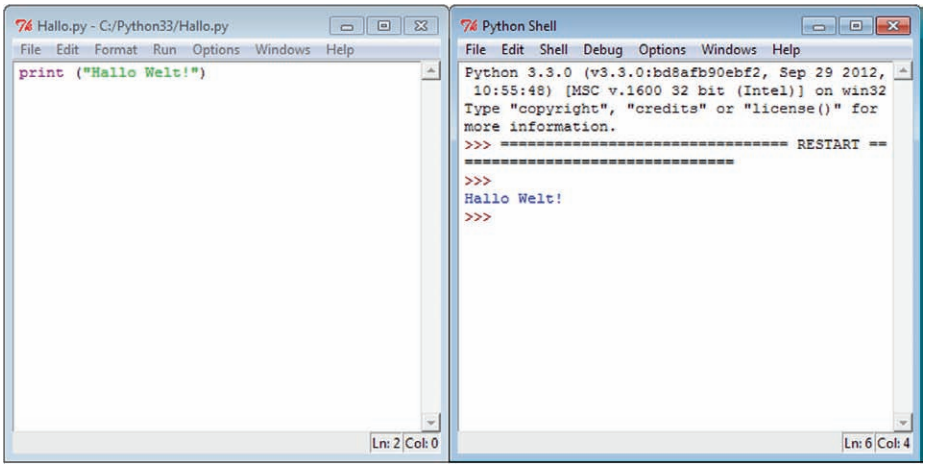
Natürlich ist es kein Problem, kleine Programme neu zu schreiben, aber große Programme, wie etwa eine Textverarbeitung, können Millionen von Programmzeilen enthalten. Wenn Du die ausdrucken würdest, hättest Du weit über 10.000 Seiten. Stell Dir nur einmal vor, Du willst diesen riesigen Papierstapel nach Hause tragen. Da kannst Du nur hoffen, dass kein heftiger Windstoß kommt.

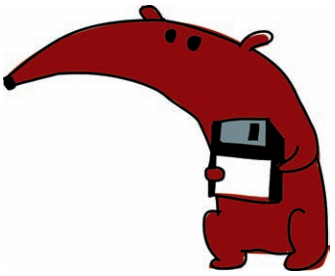
Zum Glück können wir unsere Programme für den späteren Gebrauch speichern. Um ein neues Programm zu speichern, öffne IDLE und gehe auf **File ▶ New Window**. Es öffnet sich dann ein neues Fenster, das über der Menüleiste mit ***Untitled*** (ohne Titel) bezeichnet ist.

Gib nun folgenden Code in das neue Shell-Fenster ein:

```
print("Hallo Welt!")
```

Jetzt gehst Du im Menü auf **File ▶ Save**. Sobald Du aufgefordert wirst, einen Dateinamen zu vergeben, gibst Du *Hallo.py* ein und speicherst die Datei auf Deinem Desktop. Dann gehst Du in diesem neuen Fenster im Menü auf **Run ▶Run Module**. Mit etwas Glück läuft nun Dein gespeichertes Programm und sieht in etwa so aus:

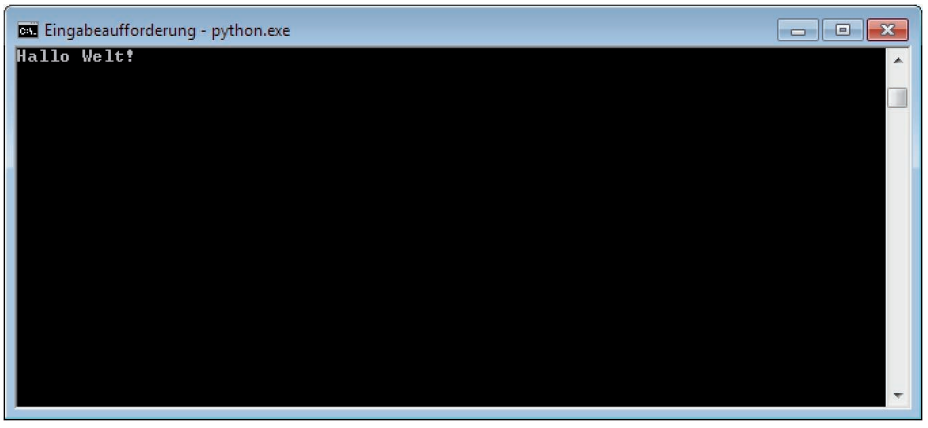




Wenn Du nun das Shell-Fenster schließt, das mit *Hallo.py* beschriftete Fenster aber offen lässt und dort wieder auf **Run ▶ Run Module** gehst, sollte das Shell-Fenster wieder auftauchen. (Um die Python-Shell wieder zu öffnen, ohne dass das Programm läuft, gehe auf **Run ▶ Python Shell**.)

Nach der Ausführung des Codes wirst Du ein neues Icon auf Deinem Desktop finden, das *Hallo.py* heißt. Wenn Du darauf doppelklickst, erscheint ganz kurz ein schwarzes Fenster. Was ist da passiert?

Du siehst kurz die Python-Eingabeaufforderung (ähnlich wie die Shell). Sie gibt »Hallo Welt!« aus und wird sofort wieder geschlossen. Dies würdest Du sehen, wenn Du die Augen eines Superhelden hättest und im Fenster erkennen könntest, was darin zu sehen ist:



Zusätzlich zu den Menüs kannst Du auch Tastaturkürzel benutzen, um ein neues Shell-Fenster zu erzeugen, eine Datei zu speichern und um ein Programm laufen zu lassen:

- Unter Windows und Ubuntu drückst Du *Ctrl*-N für ein neues Shell-Fenster, *Ctrl*-S zum Speichern Deiner Datei nach dem Bearbeiten und F5, um Dein Programm laufen zu lassen.
- Unter MacOSX drückst Du ⌘-N für ein neues Shell-Fenster, ⌘-S zum Speichern Deiner Datei nach dem Bearbeiten und F5 (eventuell brauchst Du noch die Taste *fn* dazu), um Dein Programm laufen zu lassen.

2.5 Was Du gelernt hast

In diesem Kapitel sind wir ganz einfach mit dem Programm »Hallo Welt!« eingestiegen – dem Programm, mit dem fast jeder anfängt, wenn er das Programmieren erlernt. Im nächsten Kapitel stellen wir mit der Python-Shell ein paar nützlichere Dinge an.

3 Berechnungen und Variablen

Du hast Python installiert und weißt, wie man die Python-Shell startet. Jetzt kannst Du etwas damit machen. Wir fangen mit ein paar einfachen Berechnungen an und wenden uns dann den Variablen zu. *Variablen* sind eine Möglichkeit, Dinge in einem Computerprogramm zu speichern. Sie helfen uns dabei, nützliche Programme zu schreiben.

3.1 Mit Python rechnen

Wenn jemand Dich jemand nach dem Produkt einer Multiplikation (wie z.B. 8×3,57) fragt, würdest Du normalerweise zum Taschenrechner greifen oder es auf dem Papier schriftlich ausrechnen. Du kannst aber auch die Python-Shell verwenden, um die Berechnung durchzuführen. Wir machen das jetzt einmal.

Starte die Python-Shell durch Doppelklick auf das IDLE-Icon oder, wenn Du Ubuntu nutzt, auf das IDLE-Icon im **Programme**-Menü. Nach dem Prompt gibst Du diese Gleichung ein:

```
>>> 8 * 3.57
28.56
```

Achtung: Wenn Du eine Multiplikation in Python eingibst, musst Du das Sternzeichen (*) statt eines Multiplikationszeichens (x) verwenden. Als Dezimaltrennzeichen musst Du in Python den Punkt (.) nehmen, kein Komma (,).

Wie wäre es, wenn wir jetzt eine Gleichung eingeben, die noch nützlicher ist?

Stell Dir vor, Du findest beim Graben im Garten eine Tasche mit 20 Goldmünzen. Am nächsten Tag schleichst Du Dich in den Keller und steckst die Münzen in die dampfgetriebene Kopiermaschine, die Dein Großvater erfunden hat (glücklicherweise passen genau 20 Münzen hinein). Du hörst ein Rollen und Stampfen, und nach ein paar Stunden kommen 10 weitere glänzende Goldmünzen heraus.

Wie viele Goldmünzen hättest Du in Deiner Schatzkiste, wenn Du das ein Jahr lang jeden Tag machen würdest? Auf dem Papier gerechnet, könnte das so aussehen:

$10 \times 365 = 3650$
$20 + 3650 = 3670$

Natürlich könnte man diese Berechnungen mit einem Taschenrechner oder schriftlich ganz einfach durchführen, aber in der Python-Shell geht das genauso gut. Als Erstes multiplizieren wir die 10 Münzen mit 365 Tagen eines Jahres und bekommen 3650. Danach zählen wir unsere 20 Original-Münzen dazu und erhalten 3670.

```
>>> 10 * 365
3650
>>> 20 + 3650
3670
```

Was wäre nun aber, wenn eine Elster Deine glänzenden Goldmünzen in Deinem Schlafzimmer entdecken würde und jede Woche hineingeflogen käme und dabei jeweils drei Münzen stehlen würde? Wie viele Münzen hättest Du dann nach einem Jahr? So sieht die Berechnung in der Shell aus:

```
>>> 3 * 52
156
>>> 3670 - 156
3514
```

Als Erstes multiplizieren wir 3 Münzen mit der Anzahl der Wochen eines Jahres, also 52. Das Ergebnis ist 156. Diese Zahl ziehen wir von unserer Gesamtanzahl von Münzen nach einem Jahr (3670) ab. So haben wir 3514 Münzen nach einem Jahr.

Dies war ein sehr einfaches Programm. In diesem Buch wirst Du lernen, wie man diese Konzepte beim Programmeschreiben immer weiter ausbaut und noch nützlichere Programme schreibt.

Operatoren in Python

In Python kann man Multiplikationen, Additionen, Subtraktionen und Divisionen in der Shell durchführen, aber auch andere mathematische Operationen, die wir jetzt nicht besprechen. Die grundlegenden Symbole, die Python für mathematische Operationen benutzt, heißen *Operatoren*. Sie sind in Tabelle 3–1 aufgeführt

Symbol	Operation
+	Addition
-	Subtraktion
*	Multiplikation
/	Division

Tab. 3–1 Grundlegende Operatoren in Python

Der *Vorwärtsschrägstrich* (/) wird bei Divisionen verwendet, da er an den Bruchstrich erinnert. Wenn Du zum Beispiel 100 Piraten und 20 große Fässer hättest und wissen möchtest, wie viele Piraten Du in ein Fass stecken müsstest, könntest Du 100 Piraten durch 20 Fässer teilen (100÷20) und `100/20` in die Shell eingeben. Der Vorwärtsschrägstrich ist derjenige, der nach rechts fällt. (Du findest ihn auf der Tastatur über der Ziffer 7.)

Die Rangfolge der Operationen

Um die Rangfolge von Operationen in einer Programmiersprache zu bestimmen, benutzen wir Klammern. Als *Operation* bezeichnet man alles, was Operatoren benutzt. Multiplikation und Division haben einen höheren Rang als Addition und Subtraktion, sie werden also als Erstes ausgeführt. Oder anders gesagt: Wenn Du in Python eine Gleichung eingibst, werden die Multiplikationen und Divisionen vor den Additionen und Subtraktionen ausgeführt.

Im folgenden Beispiel werden die Zahlen 30 und 20 zuerst multipliziert, und zu dem Produkt wird dann die Zahl 5 addiert.

```
>>> 5 + 30 * 20
605
```

Diese Gleichung bedeutet: »Multipliziere 30 mit 20 und addiere zum Produkt 5 dazu.« Das Ergebnis ist 605. Die Reihenfolge der Operationen können wir durch Klammern um die ersten beiden Zahlen ändern, und zwar so:

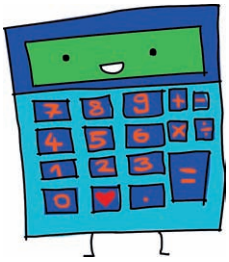
```
>>> (5 + 30) * 20
700
```

Das Ergebnis dieser Gleichung ist jetzt 700 (und nicht mehr 605), da die Klammern Python sagen, dass es zuerst die Operation innerhalb der Klammer ausführen soll und erst danach die Operation außerhalb der Klammer. Dieses Beispiel sagt also: »Addiere 5 zu 30, und multipliziere die Summe mit 20.«

Klammern können auch verschachtelt werden. Das heißt, dass Klammern innerhalb von Klammern verwendet werden können, z. B. so:

```
>>> ((5 + 30) * 20) / 10
70.0
```

In diesem Fall berechnet Python erst, was innerhalb der innersten Klammern steht, danach die Anweisung in den äußeren Klammern und zum Schluss die Division: »Addiere 5 zu 30, multipliziere die Summe mit 20, und teile das Produkt durch 10.« So läuft es ab:



- 5 addiert zu 30 ergibt 35.
- 35 mit 20 multipliziert, ergibt 700.
- 700 durch 10 dividiert, ergibt am Ende 70.

Ohne Klammern wäre das Ergebnis ein klein wenig anders:

```
>>> 5 + 30 * 20 / 10
65.0
```

In diesem Fall wird 30 erst mit 20 multipliziert (ergibt 600) und 600 durch 10 geteilt (ergibt 60). Zum Schluss wird 5 addiert, und es kommt 65 dabei heraus.

Achtung!

Achte darauf, dass Multiplikation und Division immer vor Addition und Subtraktion durchgeführt werden (»Punktrechnung geht vor Strichrechnung«) – es sei denn, dass Klammern die Rangfolge der Operationen regeln.

3.2 Variablen sind wie Bezeichnungen

Beim Programmieren steht das Wort *Variable* für einen Platz, an dem Informationen wie Zahlen, Text, Listen von Zahlen und Text usw. gespeichert werden. Eine andere Art, sich eine Variable vorzustellen, ist die, dass sie eine Bezeichnung für etwas ist.

Um zum Beispiel eine Variable mit dem Namen `fred` zu erzeugen, nehmen wir ein Gleichheitszeichen (=) und sagen Python, für welche Information die Variable eine Bezeichnung sein soll. Hier erzeugen wir jetzt die Variable `fred` und sagen, dass sie für die Zahl 100 steht (was nicht heißt, dass eine andere Variable nicht den gleichen Wert haben könnte):

```
>>> fred = 100
```

Um herauszufinden, für welchen Wert eine Variable steht, gibst Du in der Shell den Befehl `print` und danach den Namen der Variable in Klammern ein, und zwar so:

```
>>> print(fred)
100
```

Wir können Python auch sagen, dass die Variable `fred` geändert werden soll, sodass sie für etwas anderes steht. So zum Beispiel ändert man `fred` in die Zahl 200:

```
>>> fred = 200
>>> print(fred)
200
```

In der ersten Zeile sagen wir, dass fred für die Zahl 200 steht. In der zweiten Zeile fragen wir, für was `fred` steht, um uns die Änderung bestätigen zu lassen. Python gibt das Ergebnis in der letzten Zeile aus.

Wir können auch mehr als eine Bezeichnung (mehr als eine Variable) für die gleiche Sache verwenden:

```
>>> fred = 200
>>> john = fred
>>> print(john)
200
```

In diesem Beispiel sagen wir Python, dass wir den Namen (oder die Variable) `john` benutzen wollen, um die gleiche Sache damit zu bezeichnen wie mit `fred`. Dazu setzen wir einfach ein Gleichheitszeichen zwischen `john` und `fred`.

Natürlich ist `fred` wahrscheinlich kein sehr guter Name für eine Variable, da er kaum etwas darüber aussagt, wofür die Variable gebraucht wird. Statt `fred` nennen wir unsere Variable jetzt `Anzahl_der_Münzen`:

```
>>> Anzahl_der_Münzen = 200
>>> print(Anzahl_der_Münzen)
200
```

So ist klar, dass wir von 200 Münzen reden.

Die Namen der Variablen können aus Buchstaben, Zahlen und dem Unterstrich (_) bestehen, dürfen aber nicht mit einer Zahl beginnen. Man kann alles – von einzelnen Buchstaben (wie `a`) bis zu langen Sätzen – als Variablennamen verwenden.

Variablennamen dürfen aber keine Leerzeichen enthalten. Benutze daher einen Unterstrich, um Wörter zu trennen.

Manchmal, wenn man etwas Schnelles macht, sind kurze Variablennamen am besten. Der Name, für den Du Dich entscheidest, sollte so aussagekräftig sein, wie er gerade sein muss.

Jetzt, da Du weißt, wie man Variablen erzeugt, schauen wir uns an, wie man sie benutzt.

3.3 Variablen benutzen

Erinnerst Du Dich an die Gleichung, mit der wir herausgefunden haben, wie viele Münzen Du nach einem Jahr hast, wenn die komische Erfindung Deines Großvaters im Keller auf wundersame Weise neue Münzen kopiert? Wir hatten diese Rechnungen (nachdem die diebische Elster auftauchte):

```
>>> 20 + 10 * 365
3670
>>> 3 * 52
156
>>> 3670 - 156
3514
```

Wir können daraus eine einzige Programmzeile machen:

```
>>> 20 + 10 * 365 – 3 * 52
3514
```

Was wäre, wenn wir aus den Zahlen Variablen machen würden? Versuche doch einmal, Folgendes einzugeben:

```
>>> gefundene_Münzen = 20
>>> kopierte_Münzen = 10
>>> gestohlene_Münzen = 3
```

Diese Eingaben erzeugen die Variablen `gefundene_Münzen`, `kopierte_Münzen` und `gestohlene_Münzen`.

Jetzt geben wir die Gleichung noch einmal so ein:

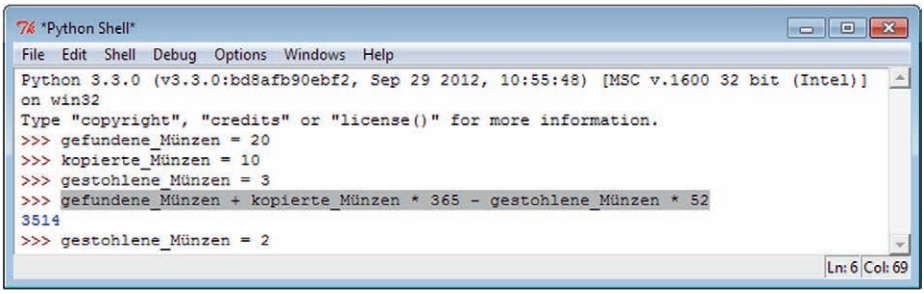
```
>>> gefundene_Münzen + kopierte_Münzen * 365 - gestohlene Münzen * 52
3514
```

Wie Du siehst, ergibt dies das gleiche Ergebnis. Was soll das Ganze jetzt? Hier kommt die Magie der Variablen ins Spiel. Was wäre, wenn Du eine Vogelscheuche vor Deinem Fenster aufstellst, und die Elster jedes Mal nur noch zwei statt drei Münzen stiehlt? Wenn wir eine Variable einsetzen, können wir die Variable, die für diese Zahl steht, einfach ändern, sodass sie sich überall, wo sie in der Gleichung steht, ändert. Wir können die Variable `gestohlene_Münzen` in 2 ändern, indem wir Folgendes eingeben:

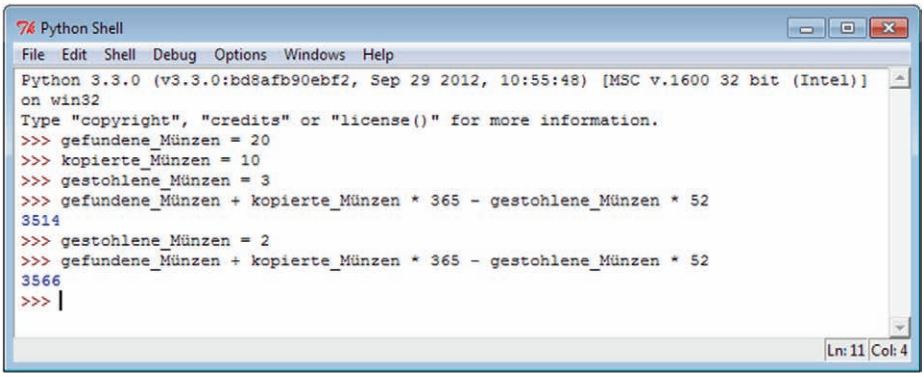
```
>>> gestohlene_Münzen = 2
```

Wir können die Gleichung wie folgt kopieren und einfügen, um das Ergebnis zu berechnen:

1. Wähle den Text, den Du kopieren möchtest, aus, indem Du mit der Maus am Anfang der Zeile klickst und dann (halte die Maustaste weiter gedrückt) bis zum Ende der Zeile ziehst. Danach sieht es aus wie hier:



2. Halte die Ctrl-Taste gedrückt (wenn Du einen Mac benutzt, ist es die ⌘-Taste), und drücke gleichzeitig auf C um den ausgewählten Text zu kopieren. (Ab jetzt sage ich dazu nur noch Ctrl-C.)
3. Klicke auf die letzte Prompt-Zeile (nach `gestohlene_Münzen = 2`).
4. Halte jetzt wieder die Ctrl-Taste gedrückt, und drücke gleichzeitig V, um den ausgewählten und kopierten Text einzufügen. (Ab jetzt sage ich dazu nur noch Ctrl-V.)
5. Drücke die Enter-Taste, um das neue Ergebnis zu sehen:



Ist das nicht viel einfacher, als die ganze Gleichung noch einmal einzugeben? Auf jeden Fall!

Du kannst auch ausprobieren, andere Variablen zu ändern, und dann durch Kopieren (Ctrl-C) und Einfügen (Ctrl-V) der Berechnung schauen, wie sich die Änderungen bemerkbar machen. Es könnte ja sein, dass – wenn man zum richti-

gen Zeitpunkt auf die Seitenteile der Erfindung Deines Großvaters haut – jedes Mal 3 zusätzliche Münzen ausgespuckt werden und Du auf diese Weise nach einem Jahr 4661 Münzen hast:

```
>>> kopierte_Münzen = 13
>>> gefundene_Münzen + kopierte_Münzen * 365 - gestohlene_Münzen * 52
4661
```

Es ist natürlich so, dass Variablen bei einer solch einfachen Gleichung immer noch nur *ein klein wenig* nützlich sind. Sie sind noch nicht so *richtig* nützlich geworden. Bis jetzt solltest Du Dir nur einfach merken, dass Variablen eine Möglichkeit sind, Dinge zu bezeichnen, die man später wieder braucht.

3.4 Was Du gelernt hast

In diesem Kapitel hast Du gelernt, wie man einfache Gleichungen mit Python-Operatoren erstellt und wie man mit Klammern die Rangfolge von Operationen bestimmt (die Reihenfolge, nach der Python die Teile der Gleichung berechnet). Anschließend haben wir Variablen erzeugt, um Werte zu bezeichnen, und diese Variablen in unseren Berechnungen eingesetzt.

4 Strings, Listen, Tupeln und Maps

In Kapitel 3 haben wir ein paar einfache Berechnungen mit Python vorgenommen, und Du hast etwas über Variablen erfahren. In diesem Kapitel werden wir mit einigen anderen Elementen von Python-Programmen arbeiten: Strings, Listen, Tupeln und Maps. Du wirst Strings benutzen, um Mitteilungen in Deinen Programmen anzuzeigen (z.B. »Mache Dich bereit« und »Das Spiel ist aus« in einem Spiel). Du wirst auch lernen, wie man mit Listen, Tupeln und Maps Sammlungen von Dingen speichert.

4.1 Strings

Unter Programmierern nennt man Text meist einen *String*. Wenn man sich einen String (engl. für »Zeichenfolge«) als Ansammlung von Buchstaben vorstellt, ergibt der Begriff einen Sinn. Alle Buchstaben, Zahlen und Symbole könnten ein String sein, ebenso Deine Adresse. Auch das erste Python-Programm, das wir in Kapitel 2 geschrieben haben, hat einen String benutzt: »Hallo Welt!«

Strings erzeugen

In Python erzeugt man einen String, indem man den Text in Anführungszeichen setzt. Wir können zum Beispiel unsere ansonsten sinnlose Variable `fred` aus Kapitel 3 nutzen, um damit einen String zu bezeichnen:

```
fred = "Warum haben Gorillas große Nasenlöcher? Weil sie große Finger
haben!"
```

Um zu überprüfen, was die Variable `fred` enthält, können wir `print(fred)` eingeben:

```
>>> print (fred)
Warum haben Gorillas große Nasenlöcher? Weil sie große Finger haben!
```

Man kann auch vorne und hinten jeweils ein Apostroph setzen, um einen String zu erzeugen:

```
>>> fred = 'Was ist rot und rosig? Rote Rosen!!!'
>>> print(fred)
Was ist rot und rosig? Rote Rosen!!!
```

Wenn Du aber versuchst, mehr als eine Zeile Text mit Apostrophen oder Anführungszeichen einzuschließen, oder mit dem einen Zeichen anfängst und mit einem anderen aufhörst, bekommst Du in der Python-Shell eine Fehlermeldung. Gib zum Beispiel einmal folgende Zeile ein:

```
>>> fred = "Was ist lauter als ein Dinosaurier?
```

Du wirst dann dieses Ergebnis bekommen:

```
SyntaxError: EOL while scanning string literal
```

Dies ist eine Fehlermeldung, die sich über die Syntax beschwert, weil Du nicht die Regel beachtet hast, nach der ein String mit einem Apostroph oder mit Anführungszeichen beendet werden muss.

Mit *Syntax* ist die Anordung und Reihenfolge von Wörtern in einem Satz oder – wie in diesem Fall – die Anordnung und Reihenfolge von Wörtern und Symbolen in einem Programm gemeint.

`SyntaxError` heißt also,

- dass Du etwas gemacht hast, was Python nicht erwartet hat, oder
- dass Python etwas erwartet, was Du vergessen hast.

EOL steht für *end-of-line* (Ende der Zeile), was bedeutet, dass Python das Ende der Zeile erreicht hat, ohne ein Anführungszeichen für das Beenden des Strings zu finden.

Damit Du mehr als eine Zeile Text in Deinem String verwenden kannst, verwendest Du drei Apostrophe (''') und drückst nach jeder Zeile die Enter-Taste:

```
>>> fred = '''Was ist lauter als ein Dinosaurier? Zwei Dinosaurier!'''
```

Jetzt lassen wir uns den Inhalt von `fred` anzeigen, um zu sehen, ob es funktioniert hat:

```
>>> print(fred)
Was ist lauter als ein Dinosaurier?
Zwei Dinosaurier!
```

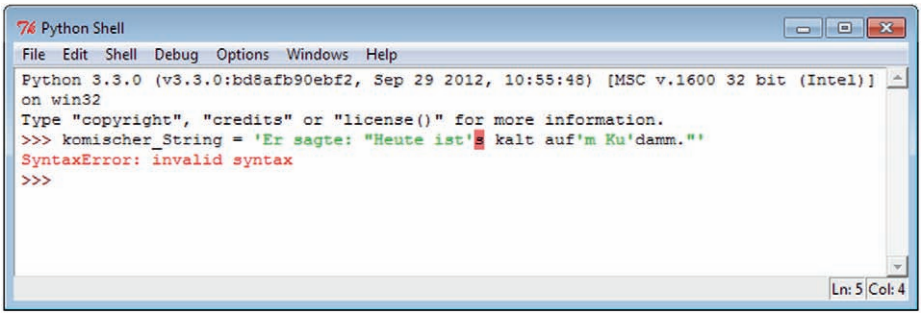
Wie man Probleme mit Strings meistert

Jetzt schaue Dir einmal dieses komische Beispiel für einen String an, der Python zu einer Fehlermeldung bringt:

```
>>> komischer_String = 'Er sagte: "Heute ist's kalt auf'm Ku'damm."'
SyntaxError: invalid syntax
```

In der ersten Zeile wollen wir einen String erzeugen (bezeichnet mit der Variable `komischer_String`), den wir zwischen zwei Apostrophe stellen. Aber im Satz stecken auch Apostrophe in den Wörtern `ist's`, `auf'm` und `Ku'damm`. Zusätzlich gibt es noch Anführungszeichen. Was für ein Chaos!

Du darfst nicht vergessen, dass Python selbst nicht so schlau ist wie ein Mensch. Python fasst `Er sagte: "Heute ist` als String auf, und danach kommen ein paar Schriftzeichen, die es nicht erwartet. Sobald Python Anführungsstriche oder Apostrophe sieht, erwartet es einen String, der nach dem ersten Anführungszeichen bzw. Apostroph beginnt und nach dem nächsten in dieser Zeile aufhört. In diesem Beispiel fängt der String mit einem Apostroph vor dem Wort `Er` an, und für Python ist der String nach dem Apostroph nach dem `t` in `ist's` zu Ende. IDLE markiert die Stelle, ab der die Dinge nicht mehr stimmen:

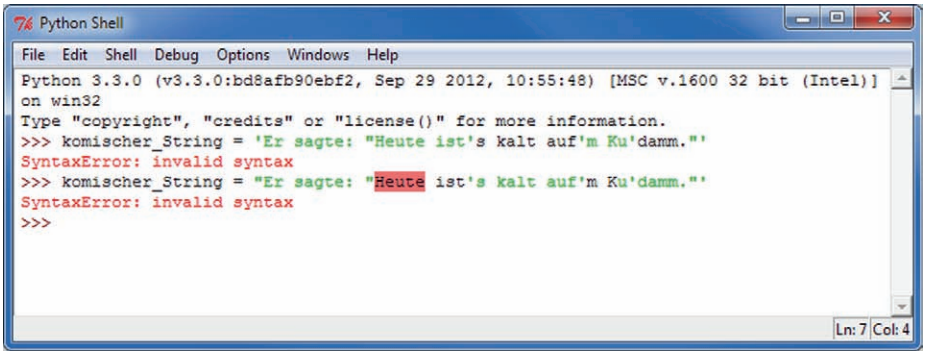


In der letzten Zeile teilt uns IDLE mit, welche Art von Fehler aufgetreten ist – in diesem Fall ein Syntax-Fehler.

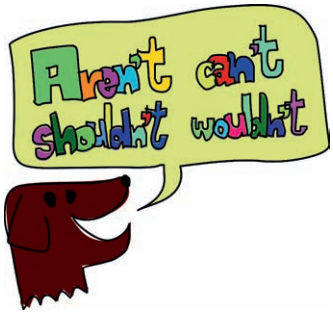
Wenn man statt der Apostrophe Anführungszeichen verwendet, gibt es immer noch eine Fehlermeldung:

```
>>> komischer_String = "Er sagte: "Heute ist's kalt auf'm Ku'damm.""
SyntaxError: invalid syntax
```

Hier sieht Python nun einen String, der in Anführungszeichen eingeschlossen ist und aus den Zeichen `Er sagte:` (und einem Leerzeichen) besteht. Alles, was danach kommt (ab `Heute`), verursacht den Fehler:



Aus der Sicht von Python gehört das ganze Zeug, das danach kommt, nicht dahin. Python sucht nach dem nächsten passenden Zeichen zur Markierung des Strings (Anführungszeichen oder Apostroph) und weiß einfach nicht, was Du mit all dem vorhast, was danach noch in derselben Zeile steht.



Der Ausweg besteht in der Markierung von Strings, die über mehrere Zeilen gehen können. Das sind die drei Apostrophe (`'''`), die wir schon kennengelernt haben. Mit ihnen können wir Anführungsstriche und Apostrophe innerhalb unseres Strings verwenden, ohne dass es Fehlermeldungen gibt. Solange wir in unserem String keine *drei* Apostrophe verwenden, können wir ein oder zwei Apostrophe und Anführungszeichen verwenden, wie es uns gefällt. Die fehlerfreie Version unseres Strings sieht also so aus:

```
>>> komischer_String = '''Er sagte: "Heute ist's kalt auf'm
Ku'damm."'''
```

Aber warte mal, da gibt es noch mehr. Wenn Du unbedingt statt der drei Apostrophe Deinen String in Apostrophe oder Anführungszeichen einschließen willst und trotzdem darin Anführungszeichen und Apostrophe verwenden möchtest, kannst Du vor jedes Anführungszeichen einen Rückwärtsschrägstrich (\) setzen. Dies nennt man *Escaping*. Es ist ein Verfahren, um Python zu sagen »Ja, ich weiß, dass ich Anführungsstriche in meinem String habe, aber ich möchte, dass Du sie ignorierst, bis Du die Anführungsstriche am Ende siehst.«

Strings mit Escapes sind aber manchmal schwerer zu lesen, sodass es wahrscheinlich besser ist, mehrzeilige Strings zu verwenden. Trotzdem kann es ja einmal sein, dass man Code-Schnipsel zu sehen bekommt, in denen diese Rückwärtsschrägstriche enthalten sind. Dann ist es gut zu wissen, wofür sie da sind.

Hier sind ein paar Beispiele, an denen Du siehst, wie das Escaping funktioniert:

```
❶ >>> Apostroph_String =
                        'Er sagte: "Heute ist\'s kalt auf\'m Ku\'damm."'
❷ >>> Anführungszeichen_String =
                        "Er sagte: \"Heute ist's kalt auf'm Ku'damm.\""
  >>> print(Apostroph_String)
  Er sagte: "Heute ist's kalt auf'm Ku'damm."
  >>> print(Anführungszeichen_String)
  Er sagte: "Heute ist's kalt auf'm Ku'damm."
```

In der ersten Zeile ❶ haben wir einen String mit Apostrophen erzeugt und vor jedem Apostroph innerhalb des Strings einen Rückwärtsschrägstrich gesetzt. In der zweiten Zeile ❷ haben wir einen String mit Anführungszeichen erzeugt und den Rückwärtsschrägstrich vor die Anführungszeichen innerhalb des Strings gestellt. In den Zeilen danach haben wir uns die Inhalte der gerade erzeugten Variablen ausgeben lassen. Wie Du siehst, kommen die Rückwärtsschrägstriche in der Ausgabe nicht mehr vor.

Werte in Strings einbetten

Wenn Du eine Nachricht anzeigen lassen möchtest, die den Inhalt einer Variable enthält, kannst Du darin mit `%s` Werte einbetten. Das `%s` funktioniert wie eine Markierung für einen Wert, den Du später noch hinzufügst. Nehmen wir an, Python soll bei einem Spiel die Anzahl der Punkte errechnen oder speichern, und wir wollen diese Punkte dann in einem Satz wie »Ich habe ____ Punkte erzielt« einfügen. Dazu können wir mit `%s` die Stelle für den Wert im Satz markieren und Python dann diesen Wert ausgeben lassen:

```
>>> Mein_Score = 1000
>>> Nachricht = 'Ich habe %s Punkte erreicht'
>>> print(Nachricht % Mein_Score)
Ich habe 1000 Punkte erreicht
```

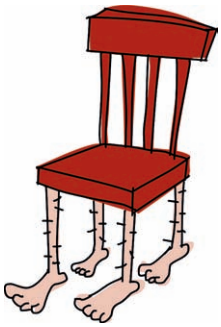
Hier haben wir die Variable `Mein_Score` mit dem Wert 1000 erzeugt sowie die Variable `Nachricht` mit einem String, der die Worte »Ich habe %s Punkte erreicht« enthält. Dabei dient %s als Platzhalter für die Punktzahl.

In der nächsten Zeile rufen wir `print(Nachricht)` mit dem %-Zeichen auf, um mit Python das `%s` gegen den in der Variable `Mein_Score` gespeicherten Wert auszutauschen. Das Ergebnis beim Ausgeben dieser Nachricht ist `Ich habe 1000 Punkte`

erreicht. Für diesen Wert hätte man eigentlich keine Variable benötigt. Wir hätten das gleiche Ergebnis auch mit `print(Nachricht % 1000)` bekommen.

Wir können auch andere Werte für den `%s`-Platzhalter einschleusen, indem wir andere Variablen benutzen, wie in diesem Beispiel hier:

```
>>> Witztext = '%s: eine Vorrichtung zum Auffinden von Möbeln im
               Dunklen'
>>> Körperteil1 = 'Knie'
>>> Körperteil2 = 'Schienbein'
>>> print(Witztext % Körperteil1)
Knie: eine Vorrichtung zum Auffinden von Möbeln im Dunklen
>>> print(Witztext % Körperteil2)
Schienbein: eine Vorrichtung zum Auffinden von Möbeln im Dunklen
```



Hier haben wir drei Variablen erzeugt. Die erste, `Witztext`, steht für den String mit dem Platzhalter `%s`. Die anderen Variablen sind `Körperteil1` und `Körperteil2`. Wir können die Variable `Witztext` ausgeben und wieder den Operator `%` verwenden, um ihn gegen den Inhalt der Variablen `Körperteil1` und `Körperteil2` auszutauschen und unterschiedliche Nachrichten zu erzeugen.

Du kannst auch mehr als einen Platzhalter in einem String verwenden:

```
>>> Zahlen = 'Was sagte die Zahl %s zur Zahl %s? Schicker Gürtel!!'
>>> print(Zahlen % (0, 8))
Was sagte die Zahl 0 zur Zahl 8? Schicker Gürtel!!
```

Wenn Du mehr als einen Platzhalter verwendest, musst Du darauf achten, dass Du die Austauschwerte, wie in diesem Beispiel, in Klammern setzt. Die Reihenfolge der Werte ist dieselbe wie im String.

Strings multiplizieren

Wie viel ergibt 10 multipliziert mit 5? 50 natürlich. Aber wie viel ergibt 10 multipliziert mit *a*? Hier ist die Antwort von Python auf diese Frage:

```
>>> print(10 * 'a')
aaaaaaaaaa
```

Python-Programmierer machen sich dies zunutze, um Strings nach einer bestimmten Anzahl von Leerzeichen auszurichten, wenn sie z. B. Nachrichten in einer Shell anzeigen lassen wollen. Wie wäre es, wenn wir einen Brief in der Shell ausgeben? Wähle dazu **File ▶ New Window**, und gib folgenden Code ein:

```
Leerzeichen = ' ' * 35
print('%s Hinten Raus 12' % Leerzeichen)
print('%s 11156 Ostschnarchheim' % Leerzeichen)
print()
print()
print('Sehr geehrte Damen und Herren,')
print()
print('ich muss Ihnen bedauerlicherweise mitteilen, dass bei meinem')
print('Toilettenhäuschen einige Dachziegeln fehlen.')
print('Ich glaube, dass der Sturm sie letzte Nacht heruntergeweht
hat.')
print()
print('Mit freundlichen Grüßen,')
print('Max Maus')
```

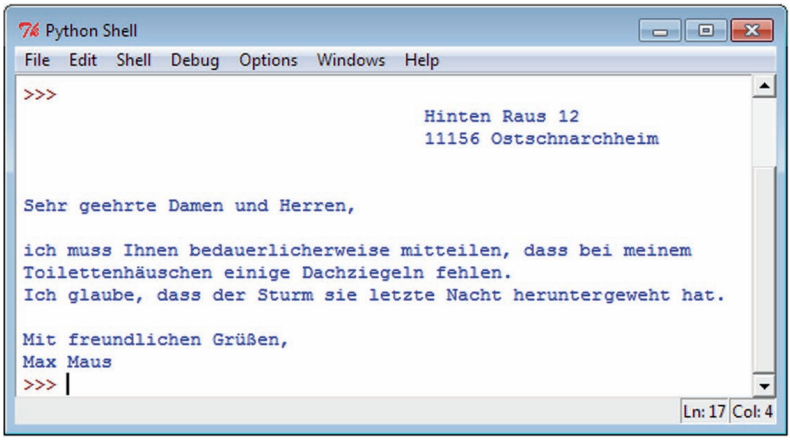
Sobald Du den Code in das Shell-Fenster eingegeben hast, gehst Du auf **File ▶ Save As**. Nenne Deinen Brief *MeinBrief.py*.

Achtung!

Von jetzt an gilt: Wenn Du *Save As: irgendeinDateiname.py* und darüber eine Menge Code siehst, musst Du auf **File ▶ New Window** gehen, den Code in das erscheinende Fenster eingeben und dann anschließend speichern, wie wir es in diesem Beispiel gemacht haben.

In der ersten Zeile dieses Beispiels haben wir die Variable `Leerzeichen` erzeugt, in der wir das Leerzeichen mit 35 multipliziert haben. Anschließend haben wir diese Variable in den nächsten zwei Zeilen verwendet, um den Text an der rechten Seite in der Shell auszurichten.

Das Ergebnis der Anweisung `print` siehst Du unten:



Wir können die Multiplikation von Strings nicht nur zum Ausrichten benutzen, sondern können uns den Bildschirm mit lustigen Mitteilungen vollschreiben lassen. Probier einmal dieses Beispiel aus:

```
>>> print(1000 * 'Matsch ')
```

4.2 Listen können mehr als Strings

Spinnenbeine, Froschzeh, Molchauge, Fledermausflügel, Schneckenschleim und Schlangenhautschuppen sind auf keiner alltäglichen Einkaufsliste (es sei denn, Du wärst ein Zauberer), aber wir nutzen sie als unser erstes Beispiel für die Unterschiede zwischen Strings und Listen. Wir können diese Liste von Elementen mit einem String wie diesem in der Variable `Zaubererliste` speichern:

```
>>> Zaubererliste = 'Spinnenbeine, Froschzeh, Molchauge,
Fledermausflügel, Schneckenschleim, Schlangenhautschuppen'
>>> print(Zaubererliste)
Spinnenbeine, Froschzeh, Molchauge, Fledermausflügel,
Schneckenschleim, Schlangenhautschuppen
```

Wir könnten aber ebenso eine *Liste* erzeugen, so eine Art magisches Python-Objekt, das wir beeinflussen können. So sähen die Elemente als Liste geschrieben aus:

```
>>> Zaubererliste = ['Spinnenbeine', 'Froschzeh', 'Molchauge',
     'Fledermausflügel', 'Schneckenschleim', 'Schlangenhautschuppen']
>>> print(Zaubererliste)
['Spinnenbeine', 'Froschzeh', 'Molchauge', 'Fledermausflügel',
'Schnekkenschleim', 'Schlangenhautschuppen']
```

Eine Liste zu erzeugen erfordert etwas mehr Tippen als bei einem String, aber man kann mit einer Liste mehr anfangen, da man sie beeinflussen kann. Wir könnten zum Beispiel den dritten Posten der `Zaubererliste` (`Molchauge`) ausgeben lassen, indem wir seine Position auf der Liste (die *Indexposition*) in eckige Klammern ([ ]) setzen:

```
>>> print(Zaubererliste[2])
Molchauge
```

Häh? Ist das nicht der dritte Posten auf der Liste? Ja, aber die Liste beginnt mit der Indexposition 0, sodass das erste Element die Position 0 hat, das zweite die Position 1 und das dritte die Position 2. Das mag uns Menschen zwar wenig sinnvoll erscheinen, den Computern dagegen schon.

Wir können ein Element einer Liste auch viel leichter ändern als in einem String. Vielleicht brauchen wir ja anstelle des Molchauges eine Schlangenzunge. So würden wir das mit unserer Liste machen:

```
>>> Zaubererliste[2] = 'Schlangenzunge'
>>> print(Zaubererliste)
['Spinnenbeine', 'Froschzeh', 'Schlangenzunge', 'Fledermausflügel',
'Schneckenschleim', 'Schlangenhautschuppen']
```

So haben wir das Element auf der Indexposition 2 von `Molchauge` in `Schlangenzunge` geändert.

Eine andere Möglichkeit besteht darin, sich eine Auswahl der Elemente auf der Liste anzeigen zu lassen. Das machen wir, indem wir einen Doppelpunkt in eckige Klammern setzen. Im nächsten Beispiel gibst Du Folgendes ein, um Dir das dritte bis fünfte Element der Liste anzeigen zu lassen (alles vorzügliche Zutaten für ein köstliches Sandwich):

```
>>> print(Zaubererliste[2:5])
['Schlangenzunge', 'Fledermausflügel', 'Schneckenschleim']
```

[2:5] zu schreiben, ist wie zu sagen: »Zeige mir das Element der Indexpositionen 2 bis (aber nicht einschließlich) 5 – oder, mit anderen Worten, die Elemente 2, 3 und 4.«

In Listen kann man alle möglichen Posten speichern, auch Zahlen:

```
>>> einige_Zahlen = [1, 2, 5, 10, 20]
```

Sie können genauso gut Strings beinhalten:

```
>>> einige_Strings = ['Wer', 'Wie', 'Wo', 'Warum']
```

Sie können aber auch sowohl Zahlen als auch Strings enthalten:

```
>>> Zahlen_und_Strings = ['Wer', 'hatte', 6, 'Angst', 'vor', 7,
                          'weil', 7, 8, 9]
>>> print(Zahlen_und_Strings)
['Wer', 'hatte', 6, 'Angst', 'vor', 7, 'weil', 7, 8, 9]
```

Und Listen können sogar andere Listen speichern:

```
>>> Zahlen = [1, 2, 3, 4]
>>> Strings = ['Ich', 'stieß', 'meinen', 'Zeh', 'und', 'jetzt', 'tut',
               'er', 'weh']
>>> MeineListe = [Zahlen, Strings]
>>> print(MeineListe)
[[1, 2, 3, 4], ['Ich', 'stieß', 'meinen', 'Zeh', 'und', 'jetzt',
'tut', 'er', 'weh']]
```

Mit dieser Liste innerhalb einer Liste wurden drei Variablen erzeugt: `Zahlen` (mit vier Zahlen darin), `Strings` (mit neun Strings) und `MeineListe`, die Zahlen und Strings enthält. Die dritte Liste (`MeineListe`) enthält nur zwei Elemente, da sie eine Liste von Variablennamen ist und nicht den Inhalt der Variablen direkt enthält.

Einer Liste Elemente hinzufügen

Um einer Liste Elemente hinzuzufügen, benutzen wir die Funktion `append`. Eine *Funktion* ist ein Batzen Code, der Python sagt, dass es etwas Bestimmtes tun soll. In diesem Fall fügt `append` dem Ende einer Liste ein Element hinzu.

Um beispielsweise der Einkaufsliste des Zauberers Bärenrülpser (ich bin mir sicher, dass es so etwas gibt) hinzuzufügen, machen wir Folgendes:

```
>>> Zaubererliste.append('Bärenrülpser')
>>> print(Zaubererliste)
['Spinnenbeine', 'Froschzeh', 'Schlangenzunge', 'Fledermausflügel',
'Schneckenschleim', 'Schlangenhautschuppen', 'Bärenrülpser']
```

Du kannst der Liste des Zauberers auf die gleiche Weise noch mehr magische Elemente hinzufügen:

```
>>> Zaubererliste.append('Alraune')
>>> Zaubererliste.append('Schierling')
>>> Zaubererliste.append('Sumpfgas')
```

Jetzt sieht die Zaubererliste so aus:

```
>>> print(Zaubererliste)
['Spinnenbeine', 'Froschzeh', 'Schlangenzunge', 'Fledermausflügel',
'Schneckenschleim', 'Schlangenhautschuppen', 'Bärenrülpser',
'Alraune', 'Schierling', 'Sumpfgas']
```

Jetzt kann man aus dieser Zutatenliste etwas wirklich Magisches brauen!

Elemente aus einer Liste entfernen

Um Elemente aus einer Liste zu entfernen, benutzt Du den Befehl `del` (Abkürzung für engl. *delete*, »löschen«). Um zum Beispiel das sechste Element von der Zaubererliste zu entfernen (die Schlangenhautschuppen), machst Du Folgendes:

```
>>> del Zaubererliste[5]
>>> print(Zaubererliste)
['Spinnenbeine', 'Froschzeh', 'Schlangenzunge', 'Fledermausflügel',
'Schneckenschleim', 'Bärenrülpser', 'Alraune', 'Schierling',
'Sumpfgas']
```

Achtung!

Denke daran, dass die Positionen einer Liste bei null beginnen und dass `Zauberer-liste[5]` sich in Wirklichkeit auf das sechste Element bezieht.

Und so entfernen wir die zuvor hinzugefügten Elemente (Alraune, Schierling und Sumpfgas) wieder von der Liste:

```
>>> del Zaubererliste[8]
>>> del Zaubererliste[7]
>>> del Zaubererliste[6]
>>> print(Zaubererliste)
['Spinnenbeine', 'Froschzeh', 'Schlangenzunge', 'Fledermausflügel',
'Schneckenschleim', 'Bärenrülpser']
```

Mit Listen rechnen

Wir können Listen durch Addition zusammenfügen. Genauso wie bei Zahlen machen wir das mit einem Pluszeichen. Wir haben zum Beispiel zwei Listen, `Liste1` mit den Zahlen 1 bis 4 und `Liste2` mit einigen Wörtern. Mit dem Befehl `print` und dem Pluszeichen können wir sie addieren:

```
>>> Liste1 = [1, 2, 3, 4]
>>> Liste2 = ['Ich', 'stolperte', 'und', 'fiel', 'zu', 'Boden']
>>> print(Liste1 + Liste2)
[1, 2, 3, 4, 'Ich', 'stolperte', 'und', 'fiel', 'zu', 'Boden']
```

Wir können auch zwei Listen zusammenzählen und das Ergebnis zu einer dritten Variable werden lassen:

```
>>> Liste1 = [1, 2, 3, 4]
>>> Liste2 = ['Ich', 'aß', 'Schokolade', 'und', 'wollte', 'mehr']
>>> Liste3 = Liste1 + Liste2
>>> print(Liste3)
[1, 2, 3, 4, 'Ich', 'aß', 'Schokolade', 'und', 'wollte', 'mehr']
```

Und wir können sogar eine Liste mit einer Zahl multiplizieren. Um zum Beispiel `Liste1` mit 5 zu multiplizieren, geben wir `Liste1 * 5` ein.

```
>>> Liste1 = [1, 2]
>>> print(Liste1 * 5)
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

So wird Python gesagt, es solle `Liste1` fünfmal wiederholen, was dann `1, 2, 1, 2, 1, 2, 1, 2, 1, 2` ergibt.

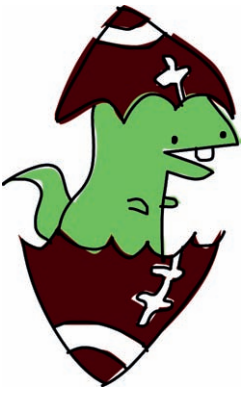
Divisionen (/) und Subtraktionen (–) dagegen führen wie in diesen Beispielen nur zu Fehlermeldungen:

```
>>> Liste1 / 20
Traceback (most recent call last):
  File "<pyshell#59>", line 1, in <module>
    Liste1 / 20
TypeError: unsupported operand type(s) for /: 'list' and 'int'

>>> Liste1 - 20
Traceback (most recent call last):
  File "<pyshell#61>", line 1, in <module>
    Liste1 - 20
TypeError: unsupported operand type(s) for -: 'list' and 'int'
```

Aber warum? Nun, Listen mit + verbinden und mit * zu wiederholen sind recht einfache Operationen. Auch im realen Leben sind sie nachzuvollziehen. Wenn ich Dir zwei Einkaufslisten in die Hand drücken und sagen würde »Addiere diese zwei Listen«, würdest Du vielleicht alle Posten auf ein weiteres Blatt Papier in der gleichen Reihenfolge bis zum Ende aufschreiben. Du kannst Dir auch sicher vorstellen, eine Liste aller Posten dreimal hintereinander auf ein Blatt zu schreiben.

Aber wie würdest Du eine Liste teilen? Stelle Dir einmal vor, Du würdest eine Liste mit sechs Zahlen (1 bis 6) durch zwei teilen. Hier sind nur drei Möglichkeiten aufgezählt:

```
[1, 2, 3]      [4, 5, 6]
[1]            [2, 3, 4, 5, 6]
[1, 2, 3, 4]   [5, 6]
```



Soll man die Liste in der Mitte aufteilen, nach dem ersten Posten oder rein zufällig eine Position wählen und dort teilen? Darauf gibt es eben keine einfache Antwort, und wenn man Python auffordert, eine Liste zu teilen, weiß es auch nicht, was es machen soll. Deshalb antwortet es mit einer Fehlermeldung.

Das Gleiche gilt für das Addieren von irgendetwas anderem als einer Liste zu einer Liste. Das geht auch nicht. Dies passiert zum Beispiel, wenn man zur `Liste1` die Zahl 50 addiert:

```
>>> Liste1 + 50
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in <module>
    Liste1 + 50
TypeError: can only concatenate list (not "int") to list
```

Warum bekommen wir hier eine Fehlermeldung? Was heißt denn das, 50 zu einer Liste zu addieren? Heißt das 50 zu jedem Element? Was aber, wenn keines der Elemente eine Zahl ist? Oder soll es bedeuten, dass man die Zahl 50 am Anfang oder Ende der Liste dazuschreibt?

Beim Schreiben von Computerprogrammen sollten Befehle immer die gleichen Dinge machen, sobald man sie eingibt. Der dumme Computer kennt nur Schwarz oder Weiß. Fordere ihn auf, eine komplizierte Entscheidung zu treffen, und er wirft mit Fehlermeldungen um sich.

4.3 Tupeln

Ein *Tupel* ist eine Liste, die in Klammern gesetzt ist – so wie in diesem Beispiel:

```
>>> fibs = (0, 1, 1, 2, 3)
>>> print(fibs[3])
2
```

Hier definieren wir die Variable `fibs` als die Zahlen 0, 1, 1, 2 und 3. Anschließend geben wir das Element mit der Indexposition 3 im Tupel mit `print(fibs[3])` aus.

Der wesentliche Unterschied zwischen einem Tupel und einer Liste ist der, dass man ein Tupel nicht ändern kann, sobald es erzeugt wurde. Wenn wir beispielsweise den ersten Wert in dem Tupel `fibs` gegen die Zahl 4 austauschen wollen (so wie wir das mit den Werten in unserer `Zaubererliste` gemacht haben), bekommen wir eine Fehlermeldung:

```
>>> fibs[0] = 4
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    fibs[0] = 4
TypeError: 'tuple' object does not support item assignment
```

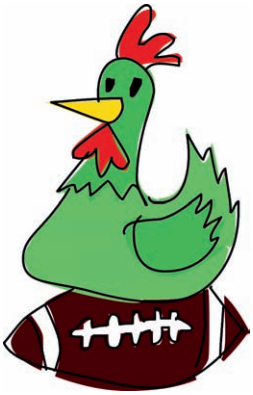
Warum sollten wir dann also überhaupt Tupel statt Listen verwenden? Hauptsächlich, weil es manchmal praktisch ist, wenn man etwas benutzt, bei dem man davon ausgehen kann, dass es sich nicht ändert. Wenn Du ein Tupel mit zwei Elementen darin erzeugst, wird es immer diese zwei Elemente in sich tragen.

4.4 Maps in Python weisen Dir nicht den Weg

Eine *Map* (engl. für »Landkarte«, auch als *dict*, Abkürzung für *Dictionary*, engl. für »Wörterbuch«, bezeichnet) ist eine Sammlung von Dingen, wie etwa von Listen und Tupeln. Der Unterschied zwischen Maps und Listen oder Tupeln besteht darin, dass jeder Posten in einer Map einen *Schlüssel (key)* und einen dazugehörigen *Wert (value)* hat.

Sagen wir, wir hätten zum Beispiel eine Liste von Personen und deren Lieblingssportarten. Wir könnten diese Informationen in eine Python-Liste schreiben, in der auf den Namen der Person ihre liebste Sportart folgt:

```
>>> Lieblingssportarten = ['Rüdiger Werner, Fussball',
                'Michael Tippler, Basketball',
                'Eduard Reichert, Radsport',
                'Renate Kalmert, Volleyball',
                'Elvira Schmidt, Badminton',
                'Frank Rohage, Schwimmen']
```



Wenn man nun nach Renate Kalmerts Lieblingssportart fragen würde, könntest Du die Liste durchsehen und die Antwort Volleyball herausfinden. Wenn aber nun 100 (oder noch viel mehr) Leute auf der Liste stehen würden?

Nun, wenn wir die gleiche Information als Map speichern, in der der Name der Person als Schlüssel und ihr Lieblingssport als Wert gespeichert ist, würde der Code in Python so aussehen:

```
>>> Lieblingssportarten = {'Rüdiger Werner' : 'Fussball',
                'Michael Tippler' : 'Basketball',
                'Eduard Reichert' : 'Radsport',
                'Renate Kalmert' : 'Volleyball',
                'Elvira Schmidt' : 'Badminton',
                'Frank Rohage' : 'Schwimmen'}
```

Wir haben mit den Doppelpunkten jeden Schlüssel von seinem Wert getrennt und jeden Schlüssel und Wert mit Apostrophen umgeben. Achte auch darauf, dass wir die Elemente der Map in geschweifte Klammern ({}) und nicht in runde oder eckige Klammern gesetzt haben.

Das Ergebnis davon ist eine Map (jeder Schlüssel führt zu einem bestimmten Wert), wie in Tabelle 4–1 zu sehen ist.

Schlüssel	Wert
Rüdiger Werner	Fussball
Michael Tippler	Basketball
Eduard Reichert	Radsport
Renate Kalmert	Volleyball
Elvira Schmidt	Badminton
Frank Rohage	Schwimmen

Tab. 4–1 Schlüssel, die auf die Werte in einer Map mit Lieblingssportarten verweisen

Um jetzt die Lieblingssportart von Renate Kalmert herauszufinden, greifen wir auf unsere Map `Lieblingssportarten` zu:

```
>>> print(Lieblingssportarten['Renate Kalmert'])
Volleyball
```

Die Anwort lautet `Volleyball`. Um einen Wert in der Map zu löschen, benutzen wir ihren Schlüssel. So etwa löschen wir Elvira Schmidt:

```
>>> del Lieblingssportarten['Elvira Schmidt']
>>> print(Lieblingssportarten)
{'Renate Kalmert': 'Volleyball', 'Rüdiger Werner': 'Fussball',
'Eduard Reichert': 'Radsport', 'Michael Tippler': 'Basketball',
'Frank Rohage': 'Schwimmen'}
```

Um einen Wert in einer Map auszutauschen, benutzen wir auch seinen Schlüssel:

```
>>> Lieblingssportarten['Rüdiger Werner'] = 'Eishockey'
>>> print(Lieblingssportarten)
{'Renate Kalmert': 'Volleyball', 'Rüdiger Werner': 'Eishockey',
'Eduard Reichert': 'Radsport', 'Michael Tippler': 'Basketball',
'Frank Rohage': 'Schwimmen'}
```

Wir haben die Lieblingssportart `Fussball` durch den Schlüssel `Rüdiger Werner` gegen `Eishockey` ausgetauscht.

Wie Du siehst, ist das Arbeiten mit Maps so ähnlich wie mit den Listen und Tupeln, außer der Tatsache, dass Du Maps nicht mit dem Operator Plus (+) zusammenfügen kannst. Wenn Du das probierst, bekommst Du eine Fehlermeldung:

```
>>> Lieblingssportarten = {'Rüdiger Werner' : 'Eishockey',
                'Michael Tippler' : 'Basketball',
                'Eduard Reichert' : 'Radsport',
                'Renate Kalmert' : 'Volleyball',
                'Frank Rohage' : 'Schwimmen'}
>>> Lieblingsmuster = {'Maximilian Fleischer' : 'rosa Punkte',
                'Johannes Bashagen' : 'orangefarbene Streifen',
                'Susanne Lehmann' : 'lila Karos'}
>>> Lieblingssportarten + Lieblingsmuster
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    Lieblingssportarten + Lieblingsmuster
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

Maps zu verbinden, ergibt für Python keinen Sinn, sodass es wieder eine Fehlermeldung ausgibt.

4.5 Was Du gelernt hast

Ist diesem Kapitel hast Du gelernt, wie Python Strings zum Speichern von Text verwendet und dass es Listen und Tupel zum Umgang mit mehreren Elementen benutzt. Du hast gesehen, dass man die Elemente in Listen verändern kann und dass man eine Liste mit einer anderen verbinden kann, die Werte in einem Tupel aber nicht. Du hast auch gelernt, wie man Maps zum Speichern von Werten benutzt, die von Schlüsseln identifiziert werden.

4.6 Programmier-Puzzles

Die folgenden paar Experimente kannst Du selber ausprobieren. Die Lösungen findest Du unter *www.dpunkt.de/python*.

#1: Lieblingssachen

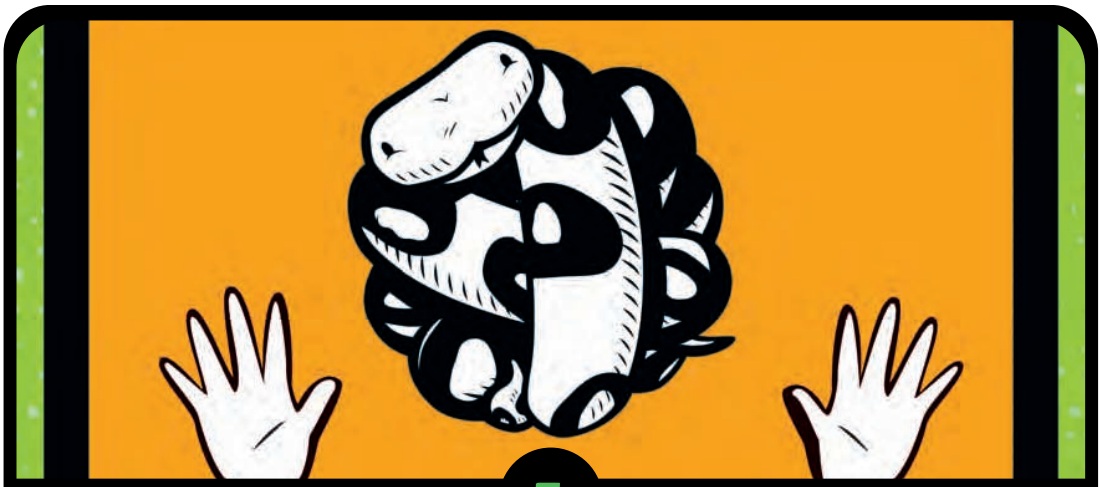
Lege eine Liste Deiner Lieblingshobbies an, und gib der Liste den Variablennamen `Hobbies`. Dann machst Du dir eine Liste Deiner Lieblingsgerichte und nennst die Variable `Essen`. Verbinde die beiden Listen, und nenne das Ergebnis `Lieblingssachen`. Am Ende gibst Du die Variable `Lieblingssachen` aus.

#2: Kämpfer zählen

Wenn es drei Gebäude gibt, auf deren Dächern sich jeweils 25 Ninjas versteckt halten, und es zwei Tunnel gibt, in denen sich jeweils 40 Samurai verkrochen haben, wie viele Ninjas und Samurai treten dann insgesamt in die Schlacht? (Du kannst dies mit einer einzigen Gleichung in der Python-Shell machen.)

#3: Grüße!

Erzeuge zwei Variablen: eine, die für Deinen Vornamen steht, und eine, die für Deinen Nachnamen steht. Erzeuge jetzt einen String, und benutze Platzhalter, um Deinen Namen mit der Nachricht auszugeben, die diese beiden Variablen verwendet, etwa so: »Hallo, Benni Richter!«



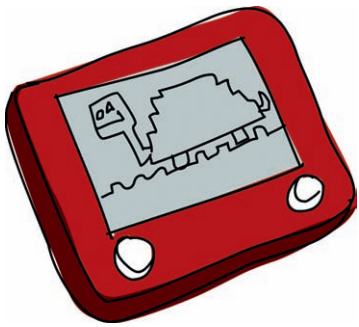
5 Malen mit Turtles

In Python ist *turtle* (engl. für »Schildkröte«) so etwas Ähnliches wie im richtigen Leben. Wir kennen die Schildkröte als ein Reptil, das sich sehr langsam fortbewegt und sein Haus auf dem Rücken mitschleppt. In der Welt von Python ist *turtle* ein kleiner, schwarzer Pfeil, der sich sehr langsam über den Monitor bewegt. Wenn man allerdings bedenkt, dass die Python-*turtle* bei der Fortbewegung auf dem Monitor eine Spur hinterlässt, denkt man viel weniger an eine Schildkröte, sondern an eine Schnecke.

Mit *turtle* kann man sehr schön die Grundlagen der Computergrafik erlernen. Aus diesem Grund benutzen wir jetzt Python-*turtle*, um einige einfache Formen und Linien zu zeichnen.

5.1 Wie man Pythons Modul turtle benutzt

Ein Modul in Python ist eine Möglichkeit, wie man nützlichen Code für ein weiteres Programm zur Verfügung stellen kann (neben anderen Dingen kann ein Modul Funktionen enthalten, die wir nutzen können). In Kapitel 8 wirst Du mehr über Funktionen erfahren. Python enthält ein spezielles Modul, das sich `turtle` nennt und mit dem man Bilder auf dem



Monitor zeichnen kann. Mit dem Modul `turtle` kann man lernen, wie man Vektorgrafiken erzeugt. Im Grunde ist das nichts weiter, als einfache Linien, Punkte und Kurven zu zeichnen.

Lass uns einmal schauen, wie `turtle` funktioniert. Als Erstes starten wir die Python-Shell, indem wir auf das Desktop-Icon klicken (wenn Du Ubuntu benutzt, gehst Du auf **IDLE** in der Programmzeile). Als Nächstes sagst Du Python, dass es das Modul `turtle` importieren soll:

```
>>> import turtle
```

Das Importieren eines Moduls sagt Python, dass Du es benutzen möchtest.

Achtung!

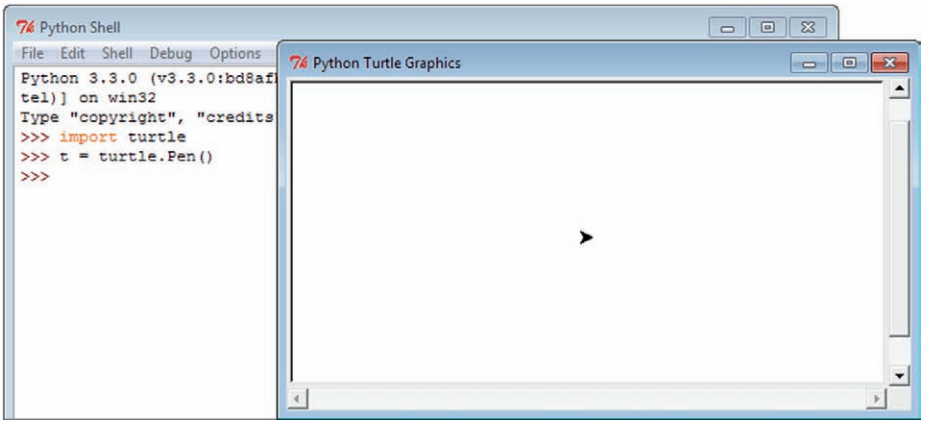
Wenn Du Ubuntu benutzt und an dieser Stelle eine Fehlermeldung bekommst, musst Du eventuell das Modul `tkinter` installieren. Dafür musst Du das Ubuntu-Software-Center öffnen und im Suchfenster *python--tk* eingeben. Im Fenster sollte dann »Tkinter – Writing Tk Applications with Python« erscheinen. Klicke auf **Installieren**, um das Package zu installieren.

Eine Leinwand erzeugen

Jetzt, wo wir das Modul `turtle` importiert haben, müssen wir als Erstes eine Leinwand erzeugen – einen leeren Platz, auf dem wir zeichnen können, so wie auf einer Leinwand eines Malers. Dazu rufen wir im Modul `turtle` die Funktion `Pen` auf, die automatisch für uns eine Leinwand erzeugt. Gib Folgendes in die Shell ein:

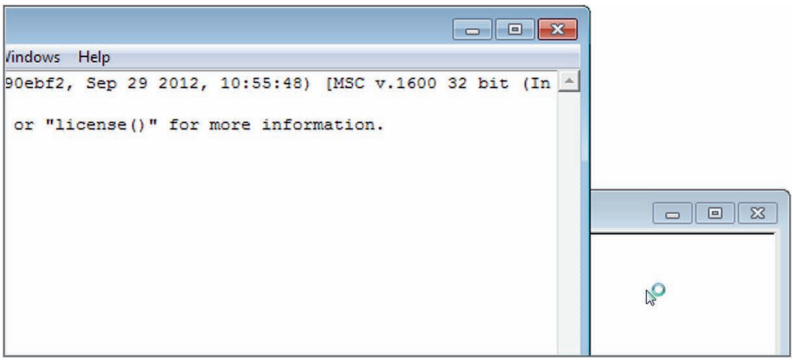
```
>>> t = turtle.Pen()
```

Danach solltest Du ein leeres Fenster (die Leinwand) mit einem Pfeil in der Mitte sehen, etwa so:



Der kleine Pfeil in der Mitte ist die Schildkröte, und ja, er sieht wirklich nicht wie eine Schildkröte aus.

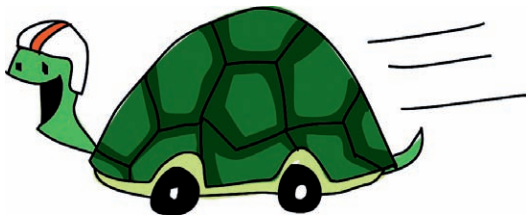
Wenn das Schildkröten-Fenster hinter dem Shell-Fenster erscheint, kann es sein, dass es nicht richtig funktioniert. Sobald Du die Maus über das Schildkröten-Fenster bewegst, taucht dann neben dem Mauszeiger ein Kringel auf:



Dies kann mehrere Ursachen haben: Du hast die Shell nicht über das Icon auf Deinem Desktop gestartet (wenn Du Windows oder Mac benutzt), sondern IDLE (Python GUI) im Windows-Startmenü ausgewählt oder IDLE nicht korrekt installiert. Versuche, die Shell vom Icon auf dem Desktop aus neu zu starten. Wenn dies nicht gelingt, probierst Du es mit der Python-Konsole anstelle der Shell. Das geht so:

- In Windows gehst Du auf **Start ▶ Alle Programme**. Dort gehst Du in den Ordner **Python 3.3** und klickst auf **Python (command line)**.
- In MacOSX klickst Du auf das Spotlight-Icon ganz oben rechts auf dem Monitor und gibst in das Suchfenster *Terminal* ein. Sobald das Fenster erschienen ist, gibst Du dort *python* ein.
- In Ubuntu öffnest Du das Terminal vom **Programme**-Menü aus und gibst dort *python* ein.

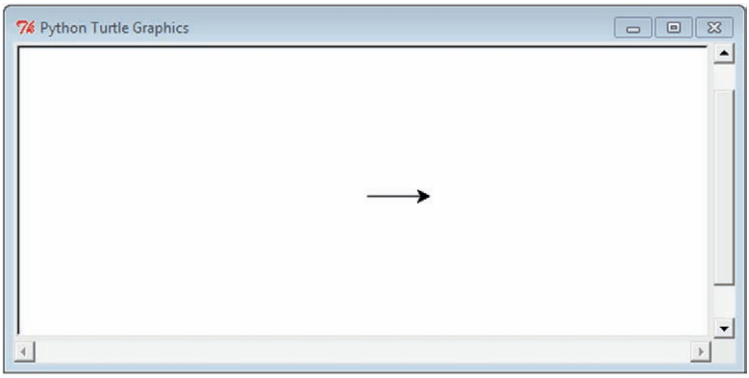
Die Schildkröte bewegen



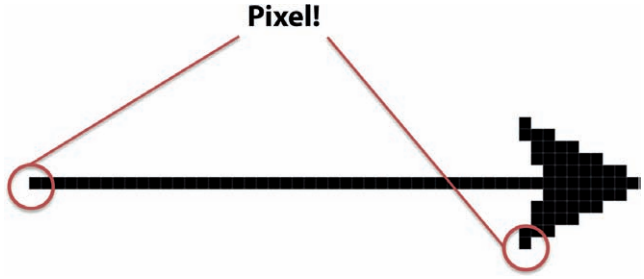
Du schickst der Schildkröte Anweisungen, indem Du Funktionen benutzt, die der Variable `t` (die wir gerade erzeugt haben), zur Verfügung stehen. Das geht genau so wie mit der Funktion `Pen` im Modul `turtle`. Die Anweisung `forward` sagt der Schildkröte, dass sie sich vorwärts bewegen soll. Damit die Schildkröte sich 50 Pixel vorwärts bewegt, gibst Du folgenden Befehl ein:

```
>>> t.forward(50)
```

Das sollte in etwa so aussehen:



Die Schildkröte hat sich um 50 Pixel vorwärts bewegt. Ein *Pixel* ist ein einzelner Punkt auf dem Monitor – das kleinste Element, das dargestellt werden kann. Alles, was Du auf dem Computermonitor siehst, ist aus Pixeln zusammengesetzt. Es sind kleine, quadratische Pünktchen (eben Pixel). Wenn Du in die Leinwand mit der Schildkröte reinzoomen könntest, würdest Du erkennen, dass die Spur der Schildkröte nur ein Haufen Pixel ist. Sie ist nur einfache Computergrafik.



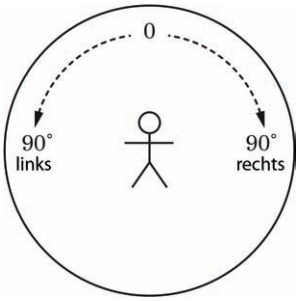
Jetzt sagen wir der Schildkröte, dass sie um 90 Grad nach links abbiegen soll. Wir tun das mit folgendem Befehl:

```
>>> t.left(90)
```

Wenn Du noch nichts von Graden gehört hast, so musst Du Dir sie so vorstellen: Du stehst in der Mitte eines Kreises.

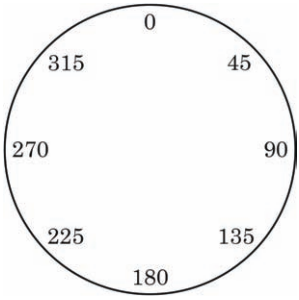
- Die Richtung, in die Du gerade schaust, ist 0 Grad.
- Wenn Du Deinen linken Arm ausstreckst, sind das 90 Grad links.
- Wenn Du Deinen rechten Arm ausstreckst, sind das 90 Grad rechts.

Du kannst die 90-Grad-Drehungen nach links oder rechts hier sehen:



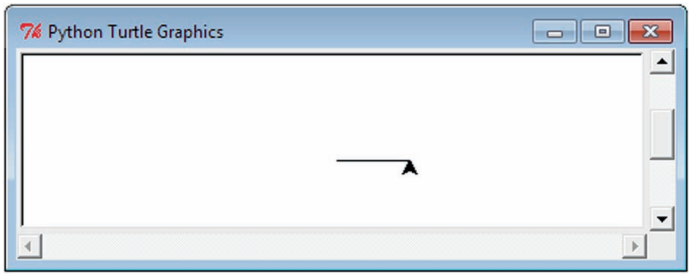
Wenn Du von dort, wo Dein Arm jetzt nach rechts zeigt, im Kreis weiter nach rechts gehst, liegen 180 Grad genau hinter Dir. 270 Grad sind dort, wo Dein linker Arm hinzeigt. 360 Grad sind dort, wo Du gestartet bist. Die Winkelgrade gehen von 0 bis 360.

Die Gradzahlen eines vollen Kreises, in dem man sich rechts herum dreht, sind hier in 45-Grad-Schritten gezeigt:



Wenn sich die Python-Schildkröte links herum dreht, schwenkt sie in eine neue Richtung um (so, als ob Du Deinen Körper dorthin drehen würdest, wo Dein Arm 90 Grad nach links zeigt).

Der Befehl `t.left(90)` richtet deshalb den Pfeil nach oben (da er ja nach rechts zeigend gestartet ist):



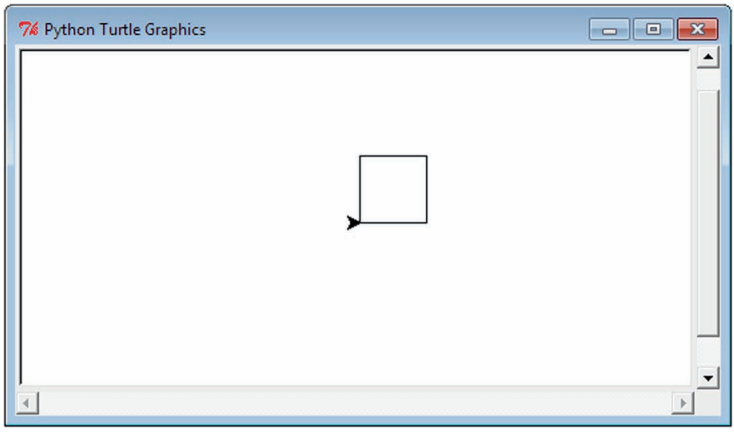
Achtung!

Wenn Du `t.left(90)` schreibst, ist das das Gleiche, als ob Du `t.right(270)` schreibst. Das gilt auch für `t.right(90)`, was dasselbe wie `t.left(270)` ist. Stelle Dir dazu einfach den Kreis vor, und folge den Gradzahlen.

Jetzt werden wir ein Quadrat zeichnen. Gib zu den bereits eingegeben Zeilen folgenden Code ein:

```
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
```

Deine Schildkröte sollte ein Quadrat gezeichnet haben und in die gleiche Richtung zeigen wie am Anfang:



Um die Leinwand zu löschen, gibt man `reset` ein. Dadurch wird die Leinwand leer, und die Schildkröte befindet sich wieder an ihrer Startposition.

```
>>> t.reset()
```

Du kannst auch `clear` eingeben, was die Leinwand löscht, die Schildkröte aber da lässt, wo sie sich gerade befindet.

```
>>> t.clear()
```

Wir können die Schildkröte auch mit `right` nach rechts und mit `backward` rückwärts bewegen. Wir können mit dem Befehl `up` den Zeichenstift von der Lein-

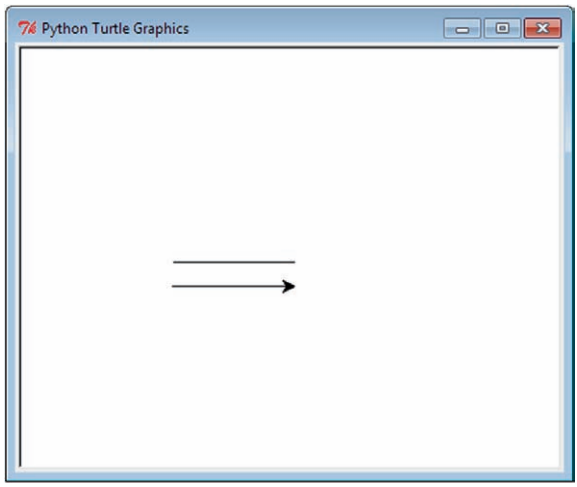
wand nehmen (oder anders gesagt, der Schildkröte sagen, dass sie aufhören soll zu zeichnen).

Wir machen jetzt noch eine Zeichnung, um einige dieser Befehle anzuwenden. Dieses Mal lassen wir die Schildkröte zwei Linien malen. Gib dazu folgenden Code ein:

```
>>> t.reset()
>>> t.backward(100)
>>> t.up()
>>> t.right(90)
>>> t.forward(20)
>>> t.left(90)
>>> t.down()
>>> t.forward(100)
```

Als Erstes löschen wir mit `t.reset()` die Leinwand und setzen dadurch die Schildkröte auf ihre Startposition. Als Nächstes lassen wir die Schildkröte mit `t.backward(100)` 100 Pixel zurücklaufen, und danach nehmen wir den Stift der Schildkröte mit `t.up()` hoch und unterbrechen das Zeichnen.

Mit dem Befehl `t.right(90)` drehen wir die Schildkröte nach rechts, um sie nach unten zeigen zu lassen, und mit `t.forward(20)` lassen wir sie 20 Pixel vorwärts gehen. Weil wir in der dritten Zeile den Befehl `up` verwendet haben, wird gerade nichts gezeichnet. Wir drehen die Schildkröte wieder mit `t.left(90)` um 90 Grad, damit sie nach rechts schaut. Mit dem Befehl `down` sagen wir der Schildkröte, dass sie den Stift wieder aufsetzen und zu malen beginnen soll. Zum Schluss lassen wir sie mit `t.forward(100)` eine Linie geradeaus, also parallel zur ersten Linie, zeichnen. Die beiden Linien, die wir gezeichnet haben, sehen am Ende so aus:



5.2 Was Du gelernt hast

In diesem Kapitel hast Du gelernt, wie man das Python-Modul `turtle` benutzt. Wir haben mit den Befehlen `left` und `right`, `forward` und `backward` einige einfache Linien gezeichnet. Du hast herausgefunden, wie man mit dem Befehl `up` das Zeichnen beendet und mit `down` wieder anfängt. Du hast auch gelernt, dass sich die Schildkröte anhand von Gradzahlen drehen lässt.

5.3 Programmier-Puzzles

Versuche die folgenden Formen mit der Schildkröte zu zeichnen. Die Lösungen findest Du unter *www.dpunkt.de/python*.

#1: Ein Rechteck

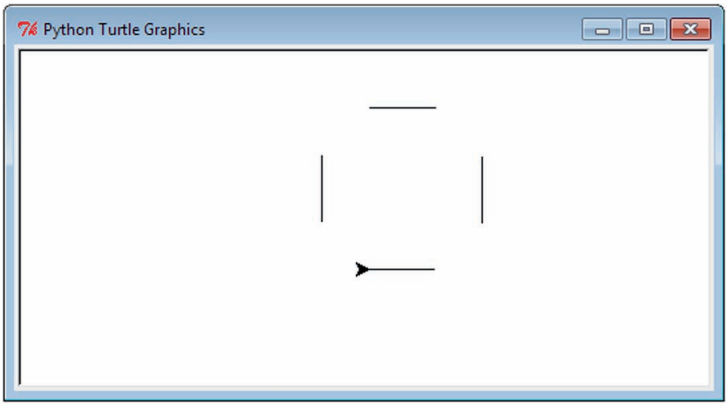
Erzeuge mit der Funktion `Pen` des Moduls `turtle` eine neue Leinwand, und zeichne dann ein Rechteck.

#2: Ein Dreieck

Erzeuge wieder eine Leinwand, und zeichne dieses Mal ein Dreieck. Schaue noch einmal zurück auf das Schema mit dem Kreis und den Gradzahlen (im Abschnitt »Die Schildkröte bewegen« auf S. 45), und vergewissere Dich, in welche Richtung Du die Schildkröte mit den Gradzahlen drehen musst.

#3: Eine Kiste ohne Ecken

Schreibe ein Programm, um diese vier Linien zu zeichnen (die Größe ist nicht so wichtig, nur die Form):



6 Fragen mit if und else stellen

Beim Programmieren stellen wir oft Fragen, die man mit Ja oder Nein beantwortet, und reagieren darauf, je nachdem, wie die Antwort lautete. Wir könnten zum Beispiel fragen: »Bist Du älter als 20?« Wenn die Antwort »ja« lautet, antworten wir »Du bist zu alt!«

Solche Fragen nennt man *Bedingungen*, und wir kombinieren diese Bedingungen und die Reaktionen darauf in sogenannten *if-Anweisungen*. Solche Bedingungen können viel komplizierter aufgebaut sein als nur aus einer Frage. `if`-Anweisungen können mit mehreren Fragen und unterschiedlichen Reaktionen – je nach Antwort auf die Frage – kombiniert werden.

In diesem Kapitel lernst Du, wie man mit `if`-Anweisungen Programme baut.

6.1 if-Anweisungen

Eine `if`-Anweisung (*if* bedeutet im Englischen »falls«) kann in Python so geschrieben werden:

```
>>> Alter = 13
>>> if Alter > 20:
        print('Du bist zu alt!')
```

Eine if-Anweisung besteht aus dem if-Schlüsselwort, auf das eine Bedingung und ein Doppelpunkt (:) folgen, so wie bei `if Alter > 20:`. Die Zeilen, die nach dem Doppelpunkt kommen, müssen in einem Block, dem Anweisungsblock, stehen. Wenn die Antwort auf die Frage »ja« lautet (oder *wahr*, wie wir Python-Programmierer sagen), werden die Befehle in dem Anweisungsblock ausgeführt. Lass uns jetzt schauen, wie man Anweisungsblöcke und Bedingungen schreibt.

Ein Anweisungsblock enthält mehrere Anweisungen

Ein Anweisungsblock ist eine Gruppe von Programmieranweisungen. Wenn zum Beispiel die Bedingung `if Alter > 20` wahr ist, möchtest Du vielleicht mehr tun können, als nur die Meldung »Du bist zu alt!« auszugeben. Vielleicht möchtest Du ein paar alternative Sätze anzeigen lassen:

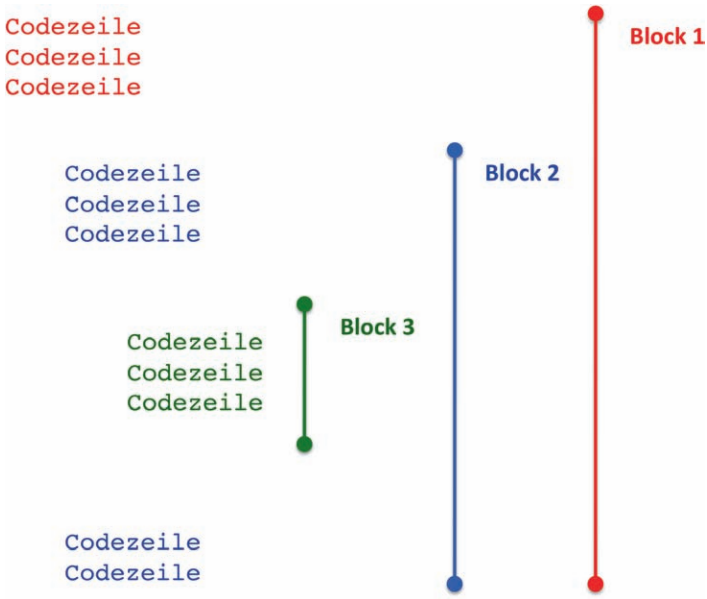
```
>>> Alter = 25
>>> if Alter > 20:
        print('Du bist zu alt!')
        print('Was machst Du hier?')
        print('Warum mähst\'n Du nicht den Rasen oder sortierst Akten?')
```

Dieser Anweisungsblock besteht aus drei print-Anweisungen, die nur ausgeführt werden, wenn die Bedingung `Alter > 20` wahr ist. Damit die print-Befehle in IDLE ausgeführt werden, musst Du die Enter-Taste zweimal drücken: das erste Mal, um den Block zu schließen (der Cursor steht dann im selben Block eine Zeile weiter unten), und das zweite Mal, um den Code auszuführen.

Jede der Zeilen dieses Blocks hat am Anfang vier Leerzeichen, wie Du im Vergleich mit der Zeile aus der if-Anweisung darüber erkennst. Lass uns noch einen Blick auf den Code mit sichtbar gemachten Leerzeichen werfen

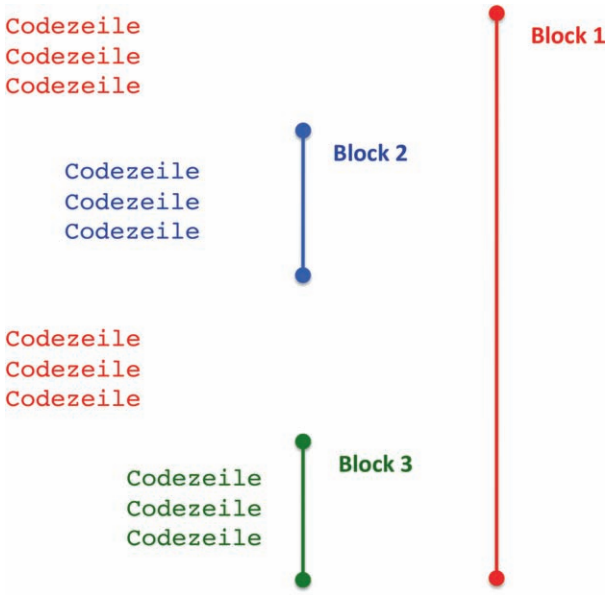
```
>>> Alter = 25
>>> if Alter > 20:
    ····print('Du bist zu alt!')
    ····print('Was machst Du hier?')
    ····print('Warum mähst\'n Du nicht den Rasen oder sortierst Akten?')
```

In Python haben *Leerräume (whitespaces)* durch Tabulatoren (sie entstehen, wenn Du die Tabulatorentaste [→] drückst) oder Leerzeichen (wenn Du die Leertaste drückst) eine Bedeutung. Code, der sich an gleicher Position befindet (vom linken Rand aus gleich weit eingerückt ist), wird in einem Anweisungsblock gruppiert. Immer, wenn Du eine neue Zeile mit mehr Leerzeichen als in der vorherigen beginnst, fängst Du einen neuen Block an, der Teil desjenigen davor ist:



Wir gruppieren Anweisungen in Blöcken, da sie zusammengehören. Diese Anweisungen müssen zusammen ausgeführt werden.

Sobald Du die Einrückungen änderst, erzeugst Du dadurch neue Blöcke. Das folgende Beispiel zeigt drei verschiedene Blöcke, die nur durch Änderung der Einrückung einstanden sind:



Obwohl die Blöcke 2 und 3 die gleiche Einrückung haben, werden sie als unterschiedliche Blöcke behandelt, da zwischen ihnen ein Block mit weniger Einrückung (weniger Leerzeichen, weiter links) steht.

Deshalb produziert ein Block mit vier Leerzeichen und einer nächsten Zeile mit sechs Leerzeichen einen Einrückungsfehler (*indentation error*), wenn er durch Python läuft, da Python davon ausgeht, dass alle Zeilen eines Blocks die gleiche Einrückung haben. Wenn Du also einen Block mit vier Leerzeichen beginnst, solltest Du in diesem Block bis zum Ende vier Leerzeichen verwenden. Hier siehst Du, was damit gemeint ist:

```
>>> if Alter > 20:
    ····print('Du bist zu alt!')
    ····››print('Was machst Du hier?')
```

Die Leerzeichen habe ich sichtbar gemacht, damit Du die Unterschiede erkennst: Die dritte Zeile hat hier sechs Leerzeichen statt der vier in der Zeile darüber.

Wenn wir diesen Code ausführen, markiert IDLE die Zeile, in der es ein Problem erkennt, mit einem roten Block und gibt eine erklärende `SyntaxError`-Meldung:

```
>>> if Alter > 25:
        print('Du bist zu alt!')
        print('Was machst Du hier?')
SyntaxError: unexpected indent
```

Python hat an in der zweiten `print`-Zeile keine zwei zusätzlichen Leerzeichen erwartet .

Achtung!

Mache Deine Einrückungen immer einheitlich, damit Dein Code besser lesbar ist. Wenn Du beginnst, ein Programm zu schreiben, und vier Leerzeichen vor den Anfang eines Anweisungsblocks setzt, solltest Du das bei den anderen Blöcken durchhalten. Achte auch darauf, dass jede Zeile eines Blocks die gleiche Einrückung hat.

Mit Bedingungen können wir Dinge vergleichen

Eine *Bedingung* ist eine Programmanweisung, die Dinge vergleicht und uns sagt, ob die Kriterien in diesem Vergleich wahr (`True`) sind und mit Ja beantwortet werden oder ob sie falsch (`False`) sind und daher mit Nein beantwortet werden. Die Bedingung `Alter > 10` kann man so ausdrücken: »Ist der Wert der Variable `Alter` größer als 10?«

Eine andere Bedingung ist: `Haarfarbe == 'lila'`, was in anderen Worten heißt: »Ist der Wert der Variable `Haarfarbe` `lila`?«

In Python benutzen wir Symbole (sogenannte *Operatoren*), um Bedingungen wie »gleich«, »größer als« und »weniger als« zu erzeugen. In Tabelle 6–1 sind ein paar Symbole für Bedingungen aufgelistet.

Symbol	Bedeutung
==	Ist gleich
!=	Ungleich
>	Größer als
<	Kleiner als
>=	Größer oder gleich
<=	Kleiner oder gleich

Tab. 6–1 Symbole für Bedingungen

Wenn Du zum Beispiel 10 Jahre alt bist, wäre die Bedingung `Dein_Alter == 10` wahr (true); ansonsten käme falsch (`false`) zurück. Wenn Du 12 Jahre alt wärst, wäre die Bedingung `Dein_Alter > 10` wahr (`true`).

Achtung!

Wenn Du eine Bedingung mit »ist gleich« formulierst, musst Du immer doppelte Gleichheitszeichen (==) verwenden.

Wir sollten noch ein paar Beispiele ausprobieren. Hier setzen wir unser Alter auf 10 und formulieren aufgrund einer Bedingung eine Anweisung, die »Du bist zu alt für meine Witze!« ausgibt, falls `Alter` größer als 10 ist.

```
>>> Alter = 10
>>> if Alter > 10:
        print('Du bist zu alt für meine Witze!')
```

Was passiert nun, wenn Du dies in IDLE eingibst und (zweimal) die Enter-Taste drückst?

Nichts.

Da der Wert, der durch die Variable `Alter` gesetzt wurde, nicht größer als 10 war, hat Python den Anweisungsblock mit dem Befehl `print` nicht ausgeführt. Wenn wir dagegen die Variable `Alter` auf 20 gesetzt hätten, wäre die Meldung ausgegeben worden.

Lass uns jetzt das vorige Beispiel ändern und die Bedingung größer als (>=) einsetzen:

```
>>> Alter = 10
>>> if Alter >= 10:
        print('Du bist zu alt für meine Witze!')
```

Jetzt solltest Du die Meldung »Du bist zu alt für meine Witze!« auf dem Monitor sehen, da der Werte der Variable Alter gleich 10 ist.

Als Nächstes probieren wir die Bedingung »ist gleich« (==) aus:

```
>>> Alter = 10
>>> if Alter == 10:
        print('Was ist braun und klebrig und läuft in der Wüste umher?
              Ein Karamel!')
```

Jetzt sollte die Meldung »Was ist braun und klebrig und läuft in der Wüste umher? Ein Karamel!« auf dem Monitor erscheinen.

6.2 If-Then-Else-Anweisungen

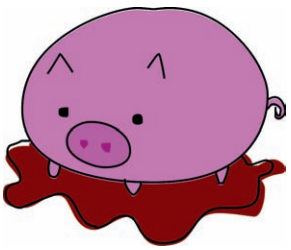
Bei den `if`-Anweisungen können wir nicht nur etwas machen, wenn die Bedingung zutrifft (wahr, `True`), sondern auch, wenn sie nicht zutrifft (falsch, `False`).

Der Trick besteht hier darin, eine `if-then-else`-Anweisung zu verwenden, die im Prinzip sagt: »*Wenn* (`if`) etwas wahr (`True`) ist, *dann* (`then`) tue dies oder tue *sonst* (`else`) das.«

Wenn Du die folgenden Codes in die Shell eingibst, setzt IDLE automatisch Einrückungen nach den `if`- und `else`-Anweisungen. Nachdem Du die `print`-Befehle eingegeben hast, musst Du daher in IDLE mit der Rückschritt- oder Entfernen-Taste nach dem letzten `print`-Befehl den Cursor an den Anfang der Zeile (ganz links) setzen. Das ist die gleiche Position, in der die `if`-Anweisung wäre, wenn kein Prompt (`>>>`) da wäre.

Wir erstellen jetzt eine solche `if-then-else`-Anweisung. Gib dazu Folgendes in die Shell ein:

```
>>> print('Möchtest Du einen schmutzigen Witz hören?')
Möchtest Du einen schmutzigen Witz hören?
>>> Alter = 12
>>> if Alter == 12:
        print('Ein Schwein fiel in den Matsch!')
else:
        print('Psst. Geheim.')

Ein Schwein fiel in den Matsch!
```



Da wir die Variable auf 12 gesetzt haben und die Bedingung fragt, ob das `Alter` gleich 12 ist, solltest Du die erste `print`-Meldung auf dem Monitor sehen. Jetzt ändern wir den Wert von `Alter` in eine andere Zahl als 12:

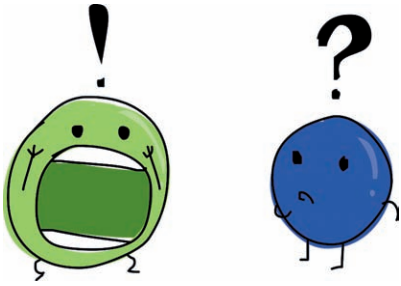
```
>>> print('Möchtest Du einen schmutzigen Witz hören?')
Möchtest Du einen schmutzigen Witz hören?
>>> Alter = 8
>>> if Alter == 12:
        print('Ein Schwein fiel in den Matsch!')
else:
        print('Psst. Geheim.')

Psst. Geheim.
```

Dieses Mal sollte die zweite `print`-Meldung kommen.

6.3 if- und elif-Anweisungen

Wir können eine `if`-Anweisung mit `elif` (einer Abkürzung für `else-if`) noch mehr erweitern. Wir können zum Beispiel abfragen, ob eine Person 10, 11 oder 12 usw. Jahre alt ist, und unser Programm je nach ihrem Alter etwas Unterschiedliches machen lassen. Diese Anweisungen unterscheiden sich von den `if-then`-Anweisungen dadurch, dass es mehr als ein `elif` in derselben Anweisung geben kann:

```
>>> Alter = 12
❶ >>> if Alter == 10:
❷         print("Wie nennt man einen Bumerang, der nicht zurückkommt?")
          print("Stock!")
❸ elif Alter == 11:
          print("Was sagt die grüne Traube zur blauen Traube?")
          print("Du musst atmen! Atme endlich!")
❹ elif Alter == 12:
❺         print("Was sagt die 0 zur 8?")
          print("Hallo Jungs!")
elif Alter == 13:
          print( "Wo wohnen Katzen?")
          print("Im Mietzhaus.")
else:
          print("Häh?")

Was sagt die 0 zur 8? Hallo Jungs!
```



In diesem Beispiel prüft die Anweisung in der zweiten Zeile ❶, ob der Wert der Variable `Alter` gleich 10 ist. Die `print`-Anweisung, die dann in ❷ folgt, wird ausgeführt, wenn das `Alter` gleich 10 ist. Da wir jedoch das `Alter` gleich 12 gesetzt haben, springt der Computer zur nächsten `if`-Anweisung in ❸ und prüft, ob der Wert von `Alter` gleich 11 ist. Da er es nicht ist, springt der Computer zur nächsten `if`-Anweisung in ❹ und schaut, ob `Alter` gleich 12 ist. Es ist 12, und deshalb führt der Computer den `print`-Befehl in ❺ aus.

6.4 Bedingungen kombinieren

Mit den Schlüsselwörtern `and` (und) und `or` (oder) kannst Du Bedingungen kombinieren und auf diese Weise kürzeren und einfacheren Code schreiben. Hier ist ein Beispiel für `or`:

```
>>> if Alter == 10 or Alter == 11 or Alter == 12 or Alter == 13:
        print('Was ergeben 13 + 49 + 84 + 155 + 97? Kopfschmerzen!')
else:
        print('Häh?')
```

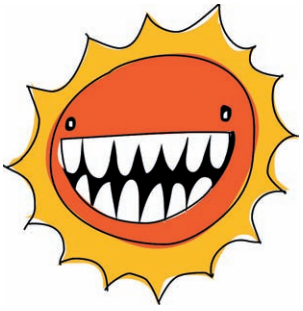
Sobald eine der Bedingungen in der ersten Zeile wahr ist (oder anders gesagt: sobald `Alter` gleich 10, 11, 12 oder 13 ist), wird der Anweisungsblock in der nächsten Zeile, der mit `print` beginnt, ausgeführt.

Wenn die Bedingungen in der ersten Zeile alle falsch sind, springt der Computer zu `else`, führt den Anweisungsblock darunter aus und zeigt `Häh?` an.

Um das Beispiel noch weiter zu kürzen, könnten wir das Schlüsselwort `and` in Kombination mit den Operatoren Größer-oder-gleich (>=) und Kleiner-oder-gleich (<=) verwenden.

```
>>> if Alter >= 10 and Alter <= 13:
        print('Was ergeben 13 + 49 + 84 + 155 + 97? Kopfschmerzen!')
else:
        print('Häh?')
```

Wenn das `Alter` jetzt größer oder gleich 10 *und* kleiner oder gleich 13 ist (in der ersten Zeile wird das als `if Alter >= 10 and Alter <= 13:` ausgedrückt), wird der Anweisungsblock in der nächsten Zeile, der mit `print` beginnt, ausgeführt. Wenn das Alter beispielsweise 12 beträgt, wird `Was ergeben 13 + 49 + 84 + 155 + 97? Kopfschmerzen!` angezeigt, da 12 mehr ist als 10, aber weniger als 13.



6.5 Variablen ohne Wert – None

Genau so, wie wir einer Variable Zahlen, Strings und Listen zuordnen können, können wir ihr auch nichts oder einen Leerwert zuordnen. In Python nennt man diesen leeren Wert `None`, und er steht für die Abwesenheit eines Inhalts. Es ist wichtig zu wissen, dass der Wert `None` sich von dem Wert 0 unterscheidet, da er etwas anderes bedeutet, als eine Zahl mit dem Wert 0. Der einzige Wert, den eine Variable hat, die den leeren Wert `None` erhalten hat, ist Nichts. Hier ein Beispiel:

```
>>> ein_Wert = None
>>> print(ein_Wert)
None
```

Den Wert None einer Variablen zuzuordnen ist eine Möglichkeit, um eine Variable in ihren leeren Ausgangszustand zu versetzen. Mit None kann man eine Variable auch definieren, ohne ihr einen Wert zuzuweisen. Das kannst Du immer dann machen, wenn Du die Variable später im Programm zwar noch brauchen wirst, Deine Variablen aber schon am Anfang alle definieren möchtest. Programmierer definieren ihre Variablen häufig am Anfang eines Programms, da man die Namen der Variablen dort leichter findet als mitten im Code.

Du kannst None auch in einer if-Anweisung abfragen, wie das folgende Beispiel zeigt:

```
>>> ein_Wert = None
>>> if ein_Wert == None:
        print('Die Variable ein_Wert hat keinen Wert')

Die Variable ein_Wert hat keinen Wert
```

Das ist immer dann nützlich, wenn man nur dann einen Wert für eine Variable berechnen möchte, falls er nicht schon berechnet wurde.

6.6 Der Unterschied zwischen Strings und Zahlen

Benutzereingaben sind das, was eine Person mit der Tastatur eingibt – egal ob es sich nun um einen Buchstaben, eine Pfeiltaste, die Enter-Taste oder sonst etwas handelt. Benutzereingaben gelangen als Strings in Python. Das bedeutet: Wenn Du mit der Tastatur die Zahl 10 eingibst, speichert Python die 10 als String in einer Variable und nicht als Zahl.

Worin besteht nun aber der Unterschied zwischen der Zahl 10 und dem String '10'? Für uns sehen sie beide gleich aus, nur dass Apostrophe um die eine 10 sind. Für den Computer sind sie aber grundverschieden.

Nehmen wir zum Beispiel einmal an, dass wir den Wert der Variable Alter mit einer Zahl in einer if-Anweisung vergleichen wollen:

```
>>> if Alter == 10:
        print('Wie spricht man am besten mit einem Monster?')
        print('Von so weit weg wie möglich!')
```

Dann weisen wir der Variable Alter die Zahl 10 zu:

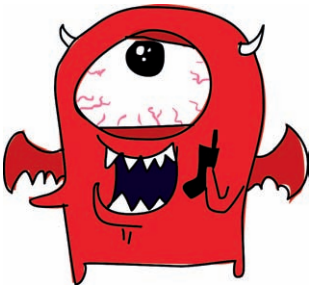
```
>>> Alter = 10
>>> if Alter == 10:
        print('Wie spricht man am besten mit einem Monster?')
        print('Von so weit weg wie möglich!')

Wie spricht man am besten mit einem Monster?
Von so weit weg wie möglich!
```

Wie Du siehst, wird jetzt die print-Anweisung ausgeführt.

Jetzt schauen wir, was passiert, wenn man der Variable `Alter` den String `'10'` (mit Apostrophen) zuweist:

```
>>> Alter = '10'
>>> if Alter == 10:
        print('Wie spricht man am besten mit einem Monster?')
        print('Von so weit weg wie möglich!')
```

In diesem Fall wird der Code der `print`-Anweisung nicht ausgeführt, da Python keine Zahl zwischen den Apostrophen erkennt, sondern sie als String ansieht.



Zum Glück hat Python magische Funktionen, mit denen man Strings in Zahlen und Zahlen in Strings verwandeln kann. Mit `int` zum Beispiel kannst Du den String '10' in eine Zahl umwandeln:

```
>>> Alter = '10'
>>> umgewandeltes_Alter = int(Alter)
```

Die Variable `umgewandeltes_Alter` enthält jetzt die Zahl 10.

Um eine Zahl in einen String umzuwandeln, benutzt Du `str`:

```
>>> Alter = 10
>>> umgewandeltes_Alter = str(Alter)
```

In diesem Fall enthält die Variable `umgewandeltes_Alter` den String `10` statt der Zahl 10.

Erinnerst Du Dich an die Anweisung `if Alter == 10`, die nichts ausgegeben hat, solange die Variable auf den String (`Alter = '10'`) gesetzt war? Wenn wir zuvor die Variable umwandeln, bekommen wir ein ganz anderes Ergebnis:

```
>>> Alter = '10'
>>> umgewandeltes_Alter = int(Alter)
>>> if umgewandeltes_Alter == 10:
        print('Wie spricht man am besten mit einem Monster?')
        print('Von so weit weg wie möglich!')

Wie spricht man am besten mit einem Monster?
Von so weit weg wie möglich!
```

Aber Achtung: Sobald Du eine Zahl mit Dezimalpunkt eingibst, bekommst Du eine Fehlermeldung, da die Funktion `int` einen ganzzahligen Wert (engl. *integer*) erwartet.

```
>>> Alter = '10.5'
>>> umgewandeltes_Alter = int(Alter)
Traceback (most recent call last):
    File "<pyshell#7>", line 1, in <module>
    umgewandeltes_Alter = int(Alter)
ValueError: invalid literal for int() with base 10: '10.5'
```

Mit dem `ValueError` sagt Python, dass der Wert, den Du ausprobiert hast, ungeeignet ist. Damit es funktioniert, nimmst Du die Funktion `float` statt `int`. Die Funktion `float` kann mit Zahlen umgehen, die nicht ganzzahlig sind.

```
>>> Alter = '10.5'
>>> umgewandeltes_Alter = float(Alter)
>>> print(umgewandeltes_Alter)
10.5
```

Python beschwert sich auch mit einem `ValueError`, wenn Du versuchst, einen String umzuwandeln, der keine Zahlen als Ziffern enthält:

```
>>> Alter = 'zehn'
>>> umgewandeltes_Alter = int(Alter)
Traceback (most recent call last):
    File "<pyshell#13>", line 1, in <module>
    umgewandeltes_Alter = int(Alter)
ValueError: invalid literal for int() with base 10: 'zehn'
```

6.7 Was Du gelernt hast

In diesem Kapitel hast Du gelernt, wie man mit `if`-Anweisungen arbeitet und Anweisungsblöcke erzeugt, die nur ausgeführt werden, wenn eine bestimmte Bedingung wahr ist. Du hast gesehen, wie man die `if`-Anweisungen mit `elif` erweitert, sodass als Reaktion auf bestimmte Bedingungen unterschiedliche Anweisungsblöcke ausgeführt werden. Du hast auch gesehen, dass man mit dem Schlüsselwort `else` Code ausführen lassen kann, falls keine der Bedingungen wahr ist. Du hast gelernt, wie man mit den Schlüsselwörtern `and` und `or` Bedingungen kombiniert und so prüfen kann, ob Zahlen in einen bestimmten Bereich fallen. Dann hast Du noch gesehen, wie man Strings und Zahlen mit `int`, `str` und `float` ineinander umwandelt. Du hast auch erfahren, dass Nichts (`None`) in Python eine Bedeutung hat und dass man damit Variablen in deren leeren Ausgangszustand zurückversetzen kann.

6.8 Programmier-Puzzles

Versuche die folgenden Puzzles mit `if`-Anweisungen und Bedingungen zu lösen. Die Lösungen finden sich unter *www.dpunkt.de/python*.

#1: Bist Du reich?

Was glaubst Du, macht der Code unten? Versuche die Antwort zu finden, ohne dass Du ihn in die Shell eingibst, und überprüfe erst danach die Anwort.

```
>>> Geld = 2000
>>> if Geld > 1000:
        print ('Ich bin reich!!')
else:
        print('Ich bin nicht reich.')
          print('Aber vielleicht später…')
```

#2: Kekse!

Erzeuge eine `if`-Anweisung, die prüft, ob eine Anzahl von Keksen (in der Variable `Kekse`) weniger als 100 oder mehr als 500 beträgt. Dein Programm sollte die Meldung »Zu wenige oder zu viele« ausgeben, falls die Bedingung wahr ist.

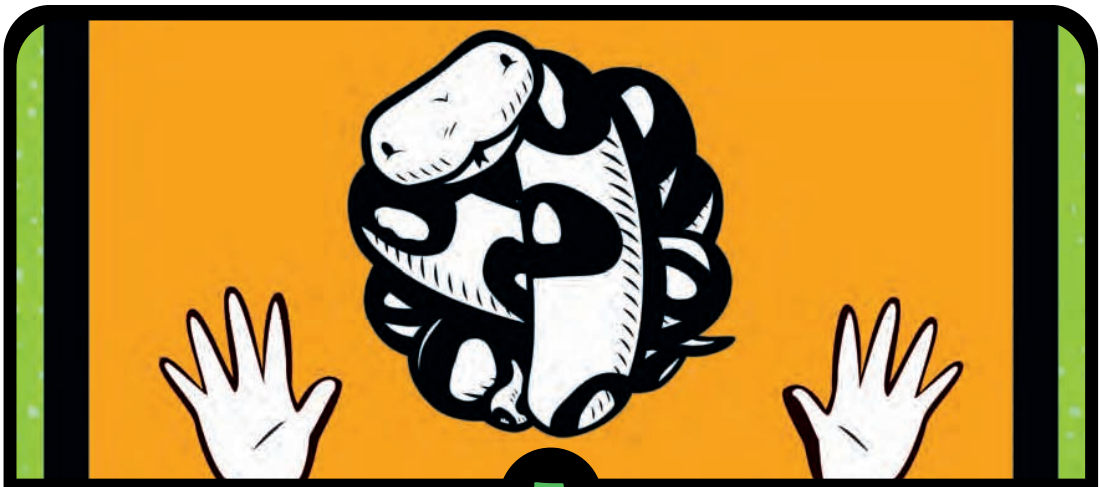
#3: Einfach die richtige Zahl

Erzeuge eine `if`-Anweisung, die prüft, ob der in der Variable `Geld` enthaltene Wert zwischen 100 und 500 oder zwischen 1000 und 5000 beträgt.

#4: Ich kann die Ninjas bezwingen

Erzeuge eine `if`-Anweisung, die den String »Das sind zu viele« ausgibt, falls die Variable `Ninjas` eine Zahl enthält, die unter 50 liegt. »Es wird hart, aber ich kann das schaffen« soll erscheinen, wenn die Zahl unter 30 liegt, und »Ich kann die Ninjas bezwingen!« soll erscheinen, wenn die Zahl unter 10 liegt. Du kannst Deinen Code so beginnen:

```
>>> Ninjas = 5
```



7 Schleifen drehen

Nichts ist schlimmer, als dieselbe Sache dauernd wiederholen zu müssen. Nicht ohne Grund zählen manche Leute Schafe, wenn sie nicht einschlafen können. Das hat nichts mit wundersamen Fähigkeiten dieser wolligen Vierbeiner zu tun. Endloses Wiederholen ist einfach langweilig, sodass Du leichter in den Schlaf kommst, da Du Dich dabei auf nichts Interessantes konzentrierst.

Programmierern macht es auch keinen Spaß, sich dauernd zu wiederholen, solange sie nicht versuchen, davon einzuschlafen. Zum Glück kennen die meisten Programmiersprachen etwas, das man `for`-Schleife nennt. Eine Schleife wiederholt automatisch Sachen wie andere Programmieranweisungen und Code-Blöcke.

In diesem Kapitel werden wir uns `for`-Schleifen und einen weiteren Schleifentyp anschauen, den Python bietet: die `while`-Schleife.

7.1 Wie man for-Schleifen benutzt

Um Hallo in Python fünfmal anzuzeigen, könntest Du Folgendes machen:

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

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

Das ist aber sehr aufwendig. Stattdessen kannst Du eine `for`-Schleife benutzen, um das viele Tippen und die Wiederholungen zu reduzieren:

```
❶ >>> for x in range(0, 5):
❷         print('Hallo')

Hallo
Hallo
Hallo
Hallo
Hallo
```

Mit der Funktion `range` in ❶ kann man eine Reihe von Zahlen erzeugen, die von der ersten bis zur Zahl *vor* der letzten Zahl reicht. Das klingt jetzt vielleicht etwas verwirrend. Lass uns einmal die Funktion `range` mit der Funktion `list` kombinieren, um genau zu sehen, wie das funktioniert. Die Funktion `range` erzeugt nämlich in Wirklichkeit keine Reihe von Zahlen, sondern gibt einen sogenannten *Iterator* zurück, ein bestimmtes Python-Objekt, das speziell für den Umgang mit Schleifen geschaffen wurde. Wenn wir `range` also mit `list` kombinieren, bekommen wir eine Zahlenreihe:

```
>>> print(list(range(10, 20)))
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

In diesem Fall einer `for`-Schleife sagt der Code in ❶ Python, dass es Folgendes tun soll:

- Beginne von 0 an zu zählen, und höre damit auf, bevor Du die 5 erreichst.
- Bei jeder Zahl, die wir zählen, speichere den Wert in der Variable `x`.

Python führt anschließend den Code-Block in ❷ aus. Achte darauf, dass es vier zusätzliche Leerzeichen am Anfang der Zeile ❷ gibt (wenn man sie mit Zeile ❶ vergleicht). IDLE setzt diese Einrückung für Dich automatisch.

Wenn wir nach Eingabe der zweiten Zeile die Enter-Taste zweimal drücken, zeigt Python fünfmal hintereinander »Hallo« an.

Wir können auch das `x` in unserer `print`-Anweisung verwenden, um die Hallos zu zählen:

```
>>> for x in range(0, 5):
        print('Hallo %s' % x)

Hallo 0
Hallo 1
Hallo 2
Hallo 3
Hallo 4
```

Wenn wir die for-Schleife wieder herausnehmen, sieht unser Code in etwa so aus:

```
>>> x = 0
>>> print('Hallo %s' % x)
Hallo 0
>>> x = 1
>>> print('Hallo %s' % x)
Hallo 1
>>> x = 2
>>> print('Hallo %s' % x)
Hallo 2
>>> x = 3
>>> print('Hallo %s' % x)
Hallo 3
>>> x = 4
>>> print('Hallo %s' % x)
Hallo 4
```

Der Einsatz einer Schleife hat uns also acht Zeilen zusätzlichen Code erspart. Gute Programmierer haben es gar nicht gern, wenn sie Dinge mehr als einmal tun müssen. Die for-Schleife ist also eine der beliebteren Anweisungen einer Programmiersprache.

Du musst nicht unbedingt die Funktionen `range` und `list` verwenden, wenn Du for-Schleifen schreibst. Du kannst auch eine Liste verwenden, die Du bereits erzeugt hast – zum Beispiel die Einkaufsliste aus Kapitel 4:

```
>>> Zaubererliste = ['Spinnenbeine', 'Froschzeh', 'Schlangenzunge',
                     'Fledermausflügel', 'Schneckenschleim', 'Bärenrülpser']
>>> for i in Zaubererliste:
        print(i)

Spinnenbeine
Froschzeh
Schlangenzunge
Fledermausflügel
Schneckenschleim
Bärenrülpser
```

Dieser Code sagt quasi: »Für jeden Posten in der `Zaubererliste` speicherst Du den Wert in der Variable `i` und gibst dann den Inhalt der Variable aus.« Wenn wir wieder die `for`-Schleife herausnehmen würden, müssten wir so etwas machen:

```
>>> Zaubererliste = ['Spinnenbeine', 'Froschzeh', 'Schlangenzunge',
                     'Fledermausflügel', 'Schneckenschleim', 'Bärenrülpser']
>>> print(Zaubererliste[0])
Spinnenbeine
>>> print(Zaubererliste[1])
Froschzeh
>>> print(Zaubererliste[2])
Schlangenzunge
>>> print(Zaubererliste[3])
Fledermausflügel
>>> print(Zaubererliste[4])
Schneckenschleim
>>> print(Zaubererliste[5])
Bärenrülpser
```

Und schon wieder hat uns die Schleife viel Tipperei erspart.

Jetzt erzeugen wir noch eine Schleife. Gib folgenden Code in die Shell ein. Die Einrückungen im Code sollte sie automatisch für Dich machen.

```
❶ >>> RiesigeflauschigeHose = ['Riesige', 'flauschige', 'Hose']
❷ >>> for i in RiesigeflauschigeHose:
❸         print(i)
❹         print(i)
❺
❻ Riesige
  Riesige
  flauschige
  flauschige
  Hose
  Hose
```

In der ersten Zeile ❶ erzeugen wir eine Liste aus `'Riesige'`, `'flauschige'` und `'Hose'`. In der nächsten Zeile ❷ schleifen wir die Posten dieser Liste durch, wobei jeder Posten der Variablen `i` zugewiesen wird. Wir geben die Inhalte der Variablen in den nächsten zwei Zeilen (❸ und ❹) aus. Durch Drücken der Enter-Taste in der nächsten Leerzeile ❺ wird Python mitgeteilt, dass der Block zu Ende ist. Der Code läuft durch, und jedes Element der Liste wird zweimal angezeigt ❻.

Denke daran, dass Du eine Fehlermeldung bekommst, wenn Du die falsche Anzahl an Leerzeichen eingibst. Wenn Du im Code oben ein zusätzliches Leerzei-

chen in Zeile ❹ eingeben würdest, würde Python einen Einrückungsfehler anzeigen:

```
>>> RiesigeflauschigeHose = ['Riesige', 'flauschige', 'Hose']
>>> for i in RiesigeflauschigeHose:
        print(i)
        print(i)

SyntaxError: unexpected indent
```

Wie Du in Kapitel 6 gelernt hast, erwartet Python, dass die Anzahl an Leerzeichen in einem Block einheitlich ist. Es spielt keine Rolle, wie viele Leerzeichen Du eingibst, solange sie nur von Zeile zu Zeile immer gleich sind. (Außerdem macht es den Code für das menschliche Auge übersichtlicher.)

Hier ist nun ein komplizierteres Beispiel einer `for`-Schleife mit zwei Anweisungsblöcken:

```
>>> RiesigeflauschigeHose = ['Riesige', 'flauschige', 'Hose']
>>> for i in RiesigeflauschigeHose:
        print(i)
        for j in RiesigeflauschigeHose:
            print(j)
```

Woraus bestehen hier die Anweisungsblöcke? Der erste Block ist die for-Schleife:

```
RiesigeflauschigeHose = ['Riesige', 'flauschige', 'Hose']
for i in RiesigeflauschigeHose:
        print(i)                        #  Diese Zeilen bilden
        for j in RiesigeflauschigeHose: #  den ERSTEN Block.
            print(j)                    #
```

Der zweite Block besteht aus der `print`-Zeile in der zweiten `for`-Schleife:

```
❶ RiesigeflauschigeHose = ['Riesige', 'flauschige', 'Hose']
  for i in RiesigeflauschigeHose:
        print(i)
❷       for j in RiesigeflauschigeHose:
❸           print(j)                    #  Diese Zeile bildet auch
                                        #  noch den ZWEITEN Block.
```

Kannst Du erkennen, was dieses kleine Stückchen Code tun wird?

Nachdem in ❶ eine Liste namens `RiesigeflauschigeHose` erzeugt wurde, können wir anhand der nächsten beiden Zeilen sagen, dass sie durch die ersten beiden Elemente der Liste wandern und jedes davon anzeigen. In ❷ allerdings wird wieder eine Schleife durch die Liste gelegt, wobei dieses Mal der Wert der Variable `j` zugewiesen wird. In ❸ wird er dann wieder angezeigt. Der Code in ❷ und ❸ ist immer noch Teil der `for`-Schleife. Dies bedeutet, dass diese Anweisungen bei jedem Element ausgeführt werden, während die `for`-Schleife die Liste durchgeht.

Wenn der Code also durchläuft, sollten wir `Riesige` gefolgt von `Riesige`, `flauschige`, `Hose` und dann `flauschige` gefolgt von `Riesige`, `flauschige`, `Hose` usw. sehen.

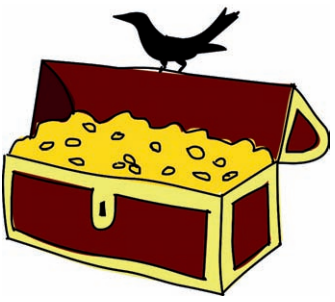
Gib den Code in die Shell von Python ein, und sieh selbst:

```
>>> RiesigeflauschigeHose = ['Riesige', 'flauschige', 'Hose']
>>> for i in RiesigeflauschigeHose:
❶       print(i)
        for j in RiesigeflauschigeHose:
❷           print(j)

◈ Riesige
  Riesige
  flauschige
  Hose
◈ flauschige
  Riesige
  flauschige
  Hose
◈ Hose
  Riesige
  flauschige
  Hose
```

Python geht in die erste Schleife und gibt ein Element aus der Liste in ❶ aus. Jetzt geht es in die zweite Schleife und gibt alle Elemente der Liste in ❷ aus. Dann geht weiter mit dem Befehl `print(i)`, wodurch das nächste Element der Liste angezeigt wird. Danach wird wieder die komplette Liste durch den Befehl `print(j)` angezeigt. In der Ausgabe sind die vom Befehl `print(i)` erzeugten Zeilen mit ◈ markiert. Die unmarkierten Zeilen wurden vom Befehl `print(j)` erzeugt.

Wie wäre es jetzt mit etwas Praktischerem als immer nur komischen Wörtern? Erinnerst Du Dich an die Berechnungen, mit denen wir in Kapitel 3 herausgefunden haben, wie viele Goldmünzen Du nach einem Jahr hättest, wenn Du mit der verrückten Erfindung Deines Großvaters die Münzen kopiert hättest? Es sah so aus:

```
>>> 20 + 10 * 365 - 3 * 52
```



Dies steht für 20 gefundene Münzen plus 10 kopierte Münzen multipliziert mit 365 Tagen eines Jahres minus 3 durch die Elster gestohlene Münzen pro Woche.

Es könnte ganz nützlich sein, wenn Du sehen könntest, wie Dein Haufen Goldmünzen jede Woche größer wird. Wir können das mit einer

weiteren for-Schleife machen. Zuerst müssen wir aber den Wert der Variable kopierte_Münzen ändern, sodass er die Gesamtzahl aller Münzen pro Woche darstellt. Das sind 10 kopierte Münzen pro Tag, bei 7 Tagen pro Woche, also beträgt kopierte_Münzen 70:

```
>>> gefundene_Münzen = 20
>>> kopierte_Münzen = 70
>>> gestohlene_Münzen = 3
```

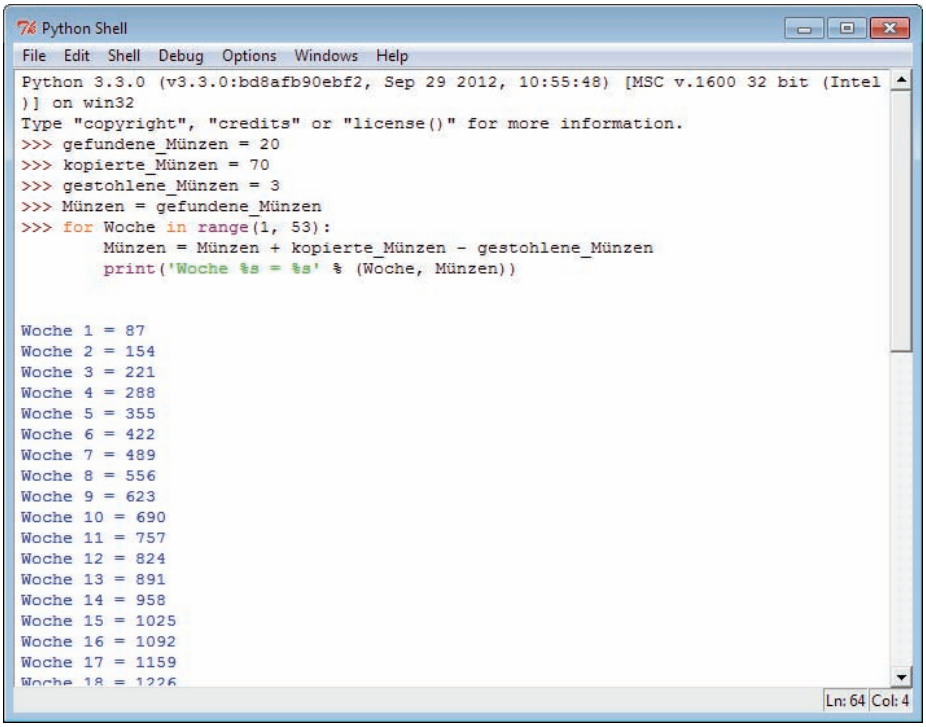
Indem wir eine weitere Variable, Münzen genannt, einführen und eine Schleife benutzen, können wir sehen, wie unser Schatz jede Woche größer wird:

```
>>> gefundene_Münzen = 20
>>> kopierte_Münzen = 70
>>> gestohlene_Münzen = 3
❶ >>> Münzen = gefundene_Münzen
❷ >>> for Woche in range(1, 53):
❸         Münzen = Münzen + kopierte_Münzen - gestohlene_Münzen
❹         print('Woche %s = %s' % (Woche, Münzen))
```

In ❶ wird die Variable Münzen mit dem Wert der Variable gefundene_Münzen besetzt; dies ist unsere Ausgangszahl. In Zeile ❷ wird die for-Schleife aufgesetzt, die durch die Befehle im Block führt. (Der Block besteht aus den Zeilen ❸ und ❹.) Jedes Mal, wenn die Schleife durchlaufen wird, wird die Variable Woche mit der nächsten Zahl in der Reihe von 1 bis 52 beladen.

Die Zeile bei ❸ ist etwas komplizierter. Im Prinzip wollen wir jede Woche die Münzen, die wir kopiert haben, hinzuzählen und die Münzen abziehen, die von der Elster gestohlen werden. Du kannst Dir die Variable Münzen als so etwas wie eine Schatzkiste vorstellen. Jede Woche werden neue Münzen in die Schatzkiste gelegt. Was die Zeile also wirklich macht, ist: »Ersetze den Inhalt der Variable Münzen durch die Anzahl der momentan vorhandenen Münzen, und zähle die Münzen dazu, die diese Woche entstanden sind.« Das Gleichheitszeichen (=) macht Folgendes klar: »Rechne erst das Zeug auf der rechten Seite aus, speichere es für später, und benutze dafür den Namen auf der linken Seite.«

Die Zeile in ❹ besteht aus einer print-Anweisung, die Platzhalter benutzt, die die Wochennummer und die Gesamtzahl der Münzen (in dieser Woche) auf dem Monitor ausgibt. (Wenn das für Dich keinen Sinn ergibt, liest Du am besten noch einmal Abschnitt »Werte in Strings einbetten« auf S. 31 nach). Wenn Du das Programm laufen lässt, bekommst Du so etwas:



7.2 Wo wir gerade von Schleifen sprechen...

Die `for`-Schleifen sind nicht die einzige Sorte Schleifen, die Du in Python nutzen kannst. Es gibt auch noch die `while`-Schleife. Die `for`-Schleife ist eine Schleife von bestimmter Länge. Die `while`-Schleife nimmt man, wenn man vorher nicht weiß, wann sie mit dem Durchschleifen aufhören soll.

Stelle Dir dazu eine Treppe mit 20 Stufen vor. Die Treppe ist drinnen, und Du weißt, dass Du 20 Stufen gut schaffst. So ist das bei einer `for`-Schleife.

```
>>> for Stufe in range(0, 20):
        print(Stufe)
```

Jetzt stellst Du Dir eine Treppe vor, die einen Berg hinaufführt. Der Berg ist richtig hoch, und vielleicht bist Du schon erschöpft, bevor Du den Gipfel erreicht hast; oder das Wetter wird schlecht und Du musst Deine Tour abbrechen. So ist das bei einer `while`-Schleife.

```
Stufe = 0
while Stufe < 10000:
        print(Stufe)
        if erschöpft == True:
                break
        elif Schlechtwetter == True:
                break
        else:
                Stufe = Stufe + 1
```

Wenn Du diesen Code eingibst und versuchst, ihn durchlaufen zu lassen, bekommst Du eine Fehlermeldung. Warum? Einfach, weil wir die Variablen `erschöpft` und `Schlechtwetter` nicht definiert haben. Auch wenn hier noch nicht genug Code steht, um daraus ein funktionierendes Programm zu machen, kann man doch sehen, wie eine `while`-Schleife funktioniert.

Wir beginnen mit der Erzeugung der Variable `Stufe` und sagen `Stufe = 0`. Als Nächstes erzeugen wir eine `while`-Schleife, die prüft, ob der Wert der Variable `Stufe` weniger als 10000 beträgt (`Stufe < 10000`), was der Gesamtzahl der Stufen bis zum Gipfel des Berges entspricht. Solange `Stufe` unter 10000 liegt, führt Python den Rest des Codes aus.

Mit `print(Stufe)` zeigen wir den Wert der Variable an und prüfen danach, ob der Wert der Variable `erschöpft` wahr (`True`) ist: `if erschöpft == True:`. (`True` ist ein sogenannter logischer Ausdruck, über den Du in Kapitel 9 mehr erfahren wirst.) Wenn der Wert wahr (`True`) ist, benutzen wir das Schlüsselwort `break`, um die Schleife zu verlassen. Mit dem Schlüsselwort `break` kann man sofort aus der Schleife heraus springen (oder, anders gesagt, sie beenden). Das funktioniert sowohl bei `for`- als auch bei `while`-Schleifen. In unserem Beispiel hat das zur Folge, dass aus dem Block in die Zeile mit `Stufe = Stufe + 1` gesprungen wird.

Die Zeile `elif Schlechtwetter == True:` prüft, ob die Variable `Schlechtwetter` auf `True` (wahr) gestellt ist. Falls ja, wird durch das Schlüsselwort `break` die Schleife verlassen. Falls weder `erschöpft` noch `Schlechtwetter` wahr (`True`) sind, geht es über `else` in die nächste Zeile, in der wir zur Variable `Stufe` 1 addieren, und wir setzen die Schleife fort: `Stufe = Stufe + 1`.

Die Schritte in einer `while`-Schleife sind also folgende:

1. Prüfe die Bedingung.
2. Führe den Code im Block aus.
3. Wiederhole das Ganze.

Meistens werden `while`-Schleifen mit mehreren Bedingungen auf einmal erzeugt, statt nur mit einer:

```
❶ >>> x = 45
❷ >>> y = 80
❸ >>> while x < 50 and y < 100:
        x = x + 1
        y = y + 1
        print(x, y)
```

Hier erzeugen wir eine Variable x mit dem Wert 45 ❶ und eine Variable y mit dem Wert 80 ❷. Die Schleife prüft in ❸ zwei Bedingungen: ob x weniger als 50 und ob y weniger als 100 beträgt.

Solange beide Bedingungen wahr sind, werden die Zeilen danach ausgeführt, und dabei wird zu beiden Werten der Variablen 1 addiert. Hier ist die Ausgabe des Codes:

```
46 81
47 82
48 83
49 84
50 85
```

Kommst Du dahinter, wie das funktioniert?

Wir fangen bei der Variable x bei 45 und der Variable y bei 80 an zu zählen (durch das Addieren von 1 zu jeder Variable) und durchlaufen dabei die Schleife. Die Schleife läuft so lange, wie x weniger als 50 und y weniger als 100 beträgt. Nachdem die Schleife fünfmal durchlaufen wurde (und jedes Mal 1 zu jeder Variable hinzugezählt wurde), erreicht der Wert von x 50. Nun ist die Bedingung (x < 50) nicht mehr wahr, und Python weiß, dass es die Schleife beenden soll.

Die `while`-Schleifen benutzt man häufig, um sogenannte *halbunendliche Schleifen* zu erzeugen. Diese Schleifen könnten sich unendlich fortsetzen, tun dies aber nur so lange, bis etwas im Code passiert, wodurch sie beendet werden. Hier ein Beispiel:

```
while True:
        Jede Menge Code hier
        Jede Menge Code hier
        Jede Menge Code hier
        if irgendein_Wert == True:
                break
```

Die Bedingung für die `while`-Schleife ist einfach wahr (`True`) und bleibt es immer, sodass der Code im Block für immer durchlaufen wird (die Schleife läuft also unendlich). Nur wenn die Variable `irgendein_Wert` wahr wird, bricht Python aus der Schleife aus. Auf Seite 123 wirst Du in »Mit `randint` eine Zufallszahl bestimmen lassen« noch ein besseres Beispiel kennenlernen. Warte aber lieber, bis Du Kapitel 8 gelesen hast, bevor Du Dir das ansiehst.

7.3 Was Du gelernt hast

In diesem Kapitel haben wir mit Schleifen sich wiederholende Aufgaben erledigt, ohne dass wir sie dauernd wiederholen mussten. Wir haben Python gesagt, was wir wiederholt haben wollten, indem wir es in Code-Blöcke geschrieben haben, die wir in die Schleifen gelegt haben. Wir haben zwei Arten von Schleifen benutzt: die `for`-Schleifen und die `while`-Schleifen, die sich zwar ähneln, aber unterschiedlich genutzt werden können. Wir haben auch das Keyword `break` benutzt, um Schleifen zu beenden – also, um aus ihnen auszusteigen.

7.4 Programmier-Puzzles

Hier sind nun einige Beispiele für Schleifen, die Du selbst ausprobieren kannst. Die Lösungen findest Du unter *www.dpunkt.de/python*.

#1: Die Hallo-Schleife

Was glaubst Du, macht der folgende Code? Überlege zuerst selbst, was passiert, und gib erst danach den Code in Python ein, um zu sehen, ob Du recht hattest.

```
>>> for x in range(0, 20):
        print('Hallo %s' % x)
        if x < 9:
                break
```

#2: Gerade Zahlen

Erzeuge eine Schleife, die gerade Zahlen ausgibt, bis sie Dein Alter erreicht. Sie könnte zum Beispiel Folgendes ausgeben:

```
2
4
6
8
10
12
14
```

#3: Meine fünf Lieblingszutaten

Erzeuge eine Liste mit fünf verschiedenen Sandwich-Zutaten wie den folgenden:

```
>>> Zutaten = ['Schnecken', 'Blutegel', 'Gorilla-Ohrenschmalz',
               'Raupen-Augenbrauen', 'Hundertfüßler-Zehen']
```

Jetzt erzeugst Du eine Schleife, die folgende Liste (mit den Zahlen dabei) ausgibt:

```
1 Schnecken
2 Blutegel
3 Gorilla-Ohrenschmalz
4 Raupen-Augenbrauen
5 Hundertfüßler-Zehen
```

#4 Wie viel wiegst Du auf dem Mond?

Wenn Du jetzt auf dem Mond stehen würdest, würde Dein Gewicht dort nur 16,5 % von dem auf der Erde betragen. Du kannst es also berechnen, indem Du Dein Gewicht auf der Erde mit 0,165 multiplizierst.

Wenn Du nun in den nächsten 15 Jahren jedes Jahr ein Kilo zunehmen würdest, wie hoch wäre dann bei Deiner jährlichen Mondreise Dein Gewicht dort? Schreibe ein Programm mit einer `for`-Schleife, das Dein Gewicht auf dem Mond für jedes der kommenden 15 Jahre ausgibt.

8
Wiederverwertung Deines Codes mit Funktionen und Modulen

Überlege einmal, wie viel Zeug Du jeden Tag wegwirfst: Wasserflaschen, Getränkedosen, Kartoffelchips-Tüten, Frischhaltefolie, Zeitungen, Zeitschriften etc. Jetzt stelle Dir einmal vor, all dieser Müll würde auf einen großen Haufen auf Eurer Einfahrt gekippt, ohne dass Papier, Plastik und Dosen getrennt würden.

Natürlich trennst Du Deinen Müll so gut wie möglich, denn niemand möchte auf dem Weg zur Schule über eine Müllhalde klettern. Stattdessen werden Deine aussortierten Glasflaschen zu neuen Gläsern und Flaschen eingeschmolzen, das Papier wird zu Recyclingpapier eingestampft, und aus Plastik werden meist größere Plastikgegenstände gefertigt. Die Dinge werden also wiederverwertet.

In der Welt des Programmierens ist Wiederverwertung genauso wichtig. Natürlich erstickt Dein Programm nicht unter einem Haufen Müll, aber wenn Du nicht einiges von dem wiederverwertest, was Du schon geschrieben hast, würdest Du Dir die Finger wund tippen. Die Wiederverwertung macht Deinen Code auch noch kürzer und leichter zu lesen.

Python bietet eine ganze Reihe von Möglichkeiten, Code wiederzuverwerten. In diesem Kapitel erfährst Du, wie das geht.

8.1 Funktionen benutzen

Eine Möglichkeit, um Code in Python wiederzuverwerten, hast Du schon gesehen: Im letzten Kapitel haben wir mit den Funktionen `range` und `list` Python zählen lassen.

```
>>> list(range(0, 5))
[0, 1, 2, 3, 4]
```

Wenn Du weißt, wie man zählt, ist es nicht sehr schwer, eine Liste mit aufsteigenden Zahlen selbst einzugeben, aber je länger die Liste wird, umso mehr musst Du tippen. Mit Funktionen jedoch, kannst Du ganz leicht eine Liste mit tausend Zahlen erzeugen.

```
>>> list(range(0, 1000))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 …, 997, 998, 999]
```

Funktionen sind Einheiten von Code, die Python befehlen, etwas Bestimmtes zu tun. Sie stellen also ein Verfahren dar, um Code wiederzuverwerten – und Du kannst Funktionen in Deinen Programmen wieder und wieder benutzen.

Wenn Du kleine Programme schreibst, sind Funktionen schon sehr praktisch. Sobald Du lange und kompliziertere Programme (wie etwa Spiele) schreibst, sind Funktionen absolut *notwendig* (falls das Programm noch in diesem Jahrhundert fertig werden soll).

Teile einer Funktion

Eine Funktion besteht aus drei Teilen: einem *Namen*, den *Parametern* und dem *Funktionskörper*. Hier siehst Du ein Beispiel für eine einfache Funktion:

```
>>> def Testfunktion(MeinName):
        print('Hallo %s' % MeinName)
```

Der Name der Funktion ist `Testfunktion`. Sie hat einen einzigen Parameter, `MeinName`, und der Funktionskörper ist der Code-Block direkt im Anschluss an die Zeile mit dem `def` (Abkürzung für engl. *define*, »definiere«). Ein *Parameter* ist eine Variable, die nur so lange existiert, wie die Funktion benutzt wird. Du kannst eine Funktion laufen lassen, indem Du ihren Namen aufrufst und den Wert des Parameters dahinter in Klammern setzt:

```
>>> Testfunktion('Marie')
Hallo Marie
```

Die Funktion kann zwei, drei oder jede andere Anzahl von Parametern haben und nicht nur einen:

```
>>> def Testfunktion(Vorname, Nachname):
        print('Hallo %s %s ' % (Vorname, Nachname))
```

Die Werte für diese beiden Parameter werden durch ein Komma getrennt:

```
>>> Testfunktion('Marie', 'Schmidt')
Hallo Marie Schmidt
```

Wir können auch zuerst einige Variablen erzeugen und dann die Funktion mit ihnen aufrufen:

```
>>> Vorname = 'Julian'
>>> Nachname = 'Reichling'
>>> Testfunktion(Vorname, Nachname)
Hallo Julian Reichling
```

Eine Funktion wird oft mit einer `return`-Anweisung benutzt, um einen Wert zurückzugeben. Du könntest zum Beispiel eine Funktion schreiben, mit der Du ausrechnen kannst, wie viel Geld Du gespart hast:

```
>>> def Erspartes(Taschengeld, Zeitung_austragen, Ausgaben):
        return Taschengeld + Zeitung_austragen - Ausgaben
```

Diese Funktion nimmt drei Parameter auf. Sie zählt die ersten beiden Parameter zusammen (`Taschengeld` und `Zeitung_austragen`) und zieht den letzten ab (`Ausgaben`). Das Ergebnis wird zurückgegeben und kann einer Variable zugewiesen werden (so wie wir andere Werte auch Variablen zuordnen) oder angezeigt werden:

```
>>> print(Erspartes(10, 10, 5))
15
```

8.2 Variablen und ihr Gültigkeitsbereich

Eine Variable, die in einem Funktionskörper steckt, kann nicht erneut verwendet werden, wenn die Funktion durchgelaufen ist, da sie nur innerhalb der Funktion existiert. In der Welt des Programmierens nennt man das einen *Gültigkeitsbereich*.

Lass uns dazu eine einfache Funktion betrachten, die ein paar Variablen benutzt, aber keine Parameter besitzt:

```
❶ >>> def Variablentest():
          erste_Variable = 10
          zweite_Variable = 20
❷         return erste_Variable * zweite_Variable
```

Bei diesem Beispiel erzeugen wir in ❶ eine Funktion `Variablentest`, die zwei Variablen miteinander multipliziert (`erste_Variable` und `zweite_Variable`) und das Ergebnis in ❷ zurückgibt.

```
>>> print(Variablentest)
200
```

Wenn wir diese Funktion mit `print` aufrufen, bekommen wir das Ergebnis 200. Wenn wir allerdings den Inhalt von `erste_Variable` ausgeben wollen (oder auch den von `zweite_Variable`) und dies nicht innerhalb des Anweisungsblocks der Funktion tun, bekommen wir eine Fehlermeldung:

```
>>> print(erste_Variable)
Traceback (most recent call last):
  File "<pyshell#145>", line 1, in <module>
    print(erste_Variable)
NameError: name 'erste_Variable' is not defined
```

Sobald dagegen eine Variable außerhalb der Funktion definiert wird, hat sie einen anderen Gültigkeitsbereich. Lass uns zum Beispiel eine Variable definieren, bevor wir unsere Funktion definieren, und dann versuchen, sie innerhalb der Funktion zu verwenden:

```
❶ >>> weitere_Variable = 100
  >>> def Variablentest2():
          erste_Variable = 10
          zweite_Variable = 20
❷         return erste_Variable * zweite_Variable * weitere_Variable
```

Obwohl in diesem Code die Variablen `erste_Variable` und `zweite_Variable` nicht außerhalb der Funktion verwendet werden können, kann die Funktion `weitere_Variable` (die in ❶ außerhalb der Funktion erzeugt wurde) in ❷ innerhalb von ihr benutzt werden.

Wenn diese Funktion aufgerufen wird, erhältst Du folgendes Ergebnis:

```
>>> print(Variablentest2())
20000
```

Jetzt nimm einmal an, Du wolltest ein Raumschiff bauen, und weil Du ein sparsamer Konstrukteur bist, verwendest Du gebrauchte Konservendosen. Du gehst davon aus, dass Du 2 Dosen pro Woche plattdrücken kannst, um die gebogenen Wände Deines Raumschiffs zu bauen. Du brauchst aber ungefähr 500 Dosen, um den Rumpf fertigzustellen. Wir können uns leicht eine Funktion schreiben,

die uns dabei hilft, herauszufinden, wie lange man zum Plätten von 500 Dosen bräuchte, wenn man das 2-mal pro Woche macht.

Lass uns eine Funktion erzeugen, die zeigt, wie viele Dosen wir in jeder Woche bis zu einer Dauer eines Jahres plattgedrückt haben. Unsere Funktion nimmt die Anzahl der Dosen als Parameter:

```
>>> def Raumschiffbau(Dosen):
        Dosen_gesamt = 0
        for Woche in range(1, 53):
                Dosen_gesamt = Dosen_gesamt + Dosen
                print('Woche %s = %s Dosen' % (Woche, Dosen_gesamt))
```

In der ersten Zeile unserer Funktion erzeugen wir die Variable `Dosen_gesamt` und setzen deren Wert auf 0. Danach erzeugen wir eine Schleife für die Wochen eines Jahres und fügen die Anzahl der geplätteten Dosen pro Woche hinzu. Dieser Anweisungsblock bildet den Inhalt unserer Funktion. Aber es gibt in dieser Funktion noch einen weiteren Codeblock: die beiden letzten Zeilen, die den Block für die `for`-Schleife darstellen.

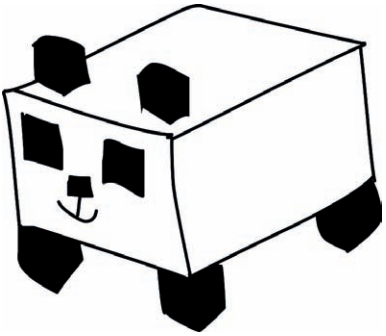
Jetzt versuchen wir, die Funktion in die Shell einzugeben, und rufen sie mit verschiedenen Werten für die Anzahl der `Dosen` auf:

```
>>> Raumschiffbau(2)
Woche 1 = 2 Dosen
Woche 2 = 4 Dosen
Woche 3 = 6 Dosen
Woche 4 = 8 Dosen
Woche 5 = 10 Dosen
Woche 6 = 12 Dosen
Woche 7 = 14 Dosen
Woche 8 = 16 Dosen
Woche 9 = 18 Dosen
Woche 10 = 20 Dosen
(setzt sich fort …)

>>> Raumschiffbau(13)
Woche 1 = 13 Dosen
Woche 2 = 26 Dosen
Woche 3 = 39 Dosen
Woche 4 = 52 Dosen
Woche 5 = 65 Dosen
(setzt sich fort …)
```

Diese Funktion kann mit unterschiedlichen Werten für die Anzahl der Dosen pro Woche wiederverwertet werden. Das ist doch etwas effizienter, als jede `for`-Schleife neu einzutippen, wenn man eine andere Zahl ausprobiert.

Funktionen lassen sich zu Modulen zusammenfassen, was Python so richtig nützlich macht (und nicht nur »so ein bisschen«).

8.3 Einsatz von Modulen



Mit *Modulen* fasst man Funktionen, Variablen und andere Dinge zu größeren, leistungsfähigeren Programmen zusammen. Manche Module sind in Python bereits enthalten, andere wiederum kannst Du separat herunterladen. Du kannst Module finden, die Dir dabei helfen, Spiele zu schreiben (so wie `tkinter`, das eingebaut ist, oder `PyGame`, das heruntergeladen werden muss), Module zur Bildbearbeitung (wie `PIL`, die *Python Imaging Library*) oder Module zum Zeichnen von 3D-Grafiken (wie `Panda3D`).

Module kann man für alle möglichen nützlichen Dinge verwenden. Wenn Du zum Beispiel einen Simulator bauen willst und möchtest, dass die Spielewelt sich realistisch ändert, kannst Du das aktuelle Datum und die Uhrzeit mit dem eingebauten Modul `time` berechnen:

```
>>> import time
```

Hier wird der Befehl `import` verwendet, um Python zu sagen, dass wir das Modul `time` verwenden möchten.

Danach können wir die in diesem Modul verfügbaren Funktionen aufrufen, indem wir einen Punkt setzen. (Du erinnerst Dich, dass wir Funktionen wie diese im Modul `turtle` verwendet haben, und zwar in Kapitel 5, etwa `t.forward(50)`.) Wir könnten zum Beispiel die Funktion `asctime` im Modul `time` aufrufen:

```
>>> print(time.asctime())
Mon Jan 14 22:16:09 2013
```

Die Funktion `asctime` ist Teil des Moduls `time` und gibt das aktuelle Datum und die Uhrzeit als String zurück.

Jetzt stell Dir vor, Du würdest jemanden, der Dein Programm benutzt, bitten wollen, einen Wert einzugeben – zum Beispiel den Geburtstag oder das Alter. Du kannst das bewerkstelligen, indem Du diese Bitte mit einer `print`-Anweisung anzeigen lässt, und das Modul `sys` (Abkürzung für *System*) nutzt, das Hilfsprogramme enthält, die mit dem Python-System selbst kommunizieren. Als Erstes importieren wir dazu das Modul `sys`:

```
>>> import sys
```

Innerhalb des Moduls `sys` gibt es ein Objekt namens `stdin` (*st*andard *in*put), dass die ziemlich nützliche Funktion `readline` enthält. Die Funktion `readline` (engl.

für »lies Zeile«) macht genau das, was ihr Name sagt: Sie liest eine Zeile Text ein, die über die Tastatur eingegeben wird, bis man die Enter-Taste drückt. (Wie Objekte funktionieren, klären wir in Kapitel 9). Um `readline` einmal zu testen, gibst Du folgenden Code in die Shell ein:

```
>>> import sys
>>> print(sys.stdin.readline())
```

Wenn Du danach ein paar Wörter eintippst und die Enter-Taste drückst, werden die Wörter in der Shell angezeigt. Jetzt denke noch einmal an den Code zurück, den wir in Kapitel 6 mit der `if`-Anweisung geschrieben haben:

```
>>> if Alter >= 10 and Alter <= 13:
        print('Was ergeben 13 + 49 + 84 + 155 + 97? Kopfschmerzen!')
else:
        print('Häh?')
```

Statt nun die Variable `Alter` zuvor zu erzeugen und ihr vor der `if`-Anweisung einen Wert zuzuordnen, können wir jetzt jemanden den Wert (das Alter) eingeben lassen. Dafür machen wir aber erst aus dem Code eine Funktion:

```
>>> def Blöder_Alterwitz(Alter):
        if Alter >= 10 and Alter <= 13:
                print('Was ergeben 13 + 49 + 84 + 155 + 97?
                      Kopfschmerzen!')
        else:
                print('Häh?')
```

Jetzt kannst Du die Funktion aufrufen, indem Du ihren Namen eingibst, und anschließend sagen, welche Zahl sie benutzen soll, indem Du die Zahl in Klammern setzt. Klappt das?

```
>>> Blöder_Alterwitz(9)
Häh?
>>> Blöder_Alterwitz(10)
Was ergeben 13 + 49 + 84 + 155 + 97? Kopfschmerzen!
```

Es klappt! Jetzt lass uns die Funktion nach dem Alter einer Person fragen. (Du kannst eine Funktion so oft ändern oder erweitern, wie Du willst.)

```
>>> def Blöder_Alterwitz():
        print('Wie alt bist Du?')
        Alter = int(sys.stdin.readline())
❶       if Alter >= 10 and Alter <= 13:
❷               print('Was ergeben 13 + 49 + 84 + 155 + 97?
                      Kopfschmerzen!')
        else:
                print('Häh?')
```

Hast Du die Funktion `int` in ❶ erkannt, die aus dem String eine Zahl macht? Wir haben diese Funktion mit eingebaut, weil `readline()` aus jeder Eingabe, die jemand macht, einen String erzeugt. Wir brauchen nun aber eine Zahl, um die Eingabe mit den Zahlen 10 und 13 in ❷ zu vergleichen. Um das jetzt selbst einmal auszuprobieren, gibst Du die Funktion ohne irgendwelche Parameter ein; und wenn die Frage `Wie alt bist Du?` erscheint, gibst Du eine Zahl ein:

```
>>> Blöder_Alterwitz()
Wie alt bist Du?
10
Was ergeben 13 + 49 + 84 + 155 + 97? Kopfschmerzen!
>>> Blöder_Alterwitz()
Wie alt bist Du?
15
Häh?
```

8.4 Was Du gelernt hast

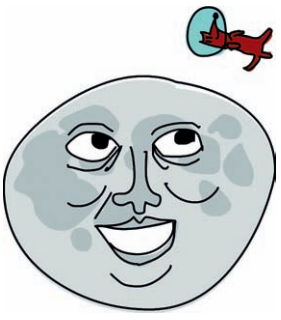
In diesem Kapitel hast Du gesehen, wie man in Python wiederverwertbare Code-Abschnitte mit Funktionen erstellt und wie man Funktionen verwendet, die von Modulen bereitgestellt werden. Du hast gelernt, wie der Gültigkeitsbereich von Variablen bestimmt, ob sie innerhalb oder außerhalb einer Funktion liegen, und wie man das Schlüsselwort `def` benutzt. Du hast auch herausgefunden, wie man Module importiert, damit Du ihren Inhalt verwenden kannst.

8.5 Programmier-Puzzles

Probiere einmal die folgenden Beispiele aus, um mit Deinen selbst erzeugten Funktionen zu experimentieren. Die Lösungen findest Du unter

www.dpunkt.de/python.

#1: Einfache Funktion für Dein Gewicht auf dem Mond



In einem der Puzzles zu Kapitel 7 hast Du eine `for`-Schleife erzeugt, um Dein Gewicht auf dem Mond über einen Zeitraum von 15 Jahren zu bestimmen. Diese `for`-Schleife kannst Du ganz einfach in eine Funktion verwandeln. Versuche eine Funktion zu erzeugen, die ein Startgewicht aufnimmt und das Gewicht jedes Jahr steigen lässt. Du könntest die neue Funktion etwa mit diesem Code aufrufen:

```
>>> Mondgewicht(30, 0.25)
```

#2: Was wiegst Du auf dem Mond nach x Jahren?

Nimm die Funktion, die Du eben erzeugt hast, und ändere sie so, dass sie Dein Gewicht über verschiedene Zeitabschnitte (über 5 oder 20 Jahre) bestimmt. Achte darauf, dass sie drei Argumente aufnimmt: das Startgewicht, die Gewichtszunahme innerhalb eines Jahres und die Gesamtzahl der Jahre:

```
>>> Mondgewicht(40, 0.25, 5)
```

#3: Ein Programm für Dein Gewicht auf dem Mond

Statt einer einfachen Funktion, bei der Du die Werte als Parameter weitergibst, kannst Du ein Mini-Programm schreiben, das die Werte mit der Funktion `sys.stin.readline()` abfragt. Danach kannst Du die Funktionen ohne Parameter aufrufen:

```
>>> Mondgewicht()
```

Die Funktion zeigt dann eine Mitteilung, die erst nach dem Startgewicht fragt, nach dessen Eingabe nach der Gewichtszunahme in einem Jahr und abschließend nach der Gesamtzahl der Jahre. Das sollte dann so aussehen:

```
Bitte gibt Dein momentanes Gewicht auf der Erde ein
45
Bitte gib die jährliche Gewichtszunahme ein
0.4
Gib nun die Gesamtzahl der Jahre ein
12
```

Vergiss nicht, vor der Erzeugung Deiner Funktion das Modul `sys` zu importieren:

```
>>> import sys
```

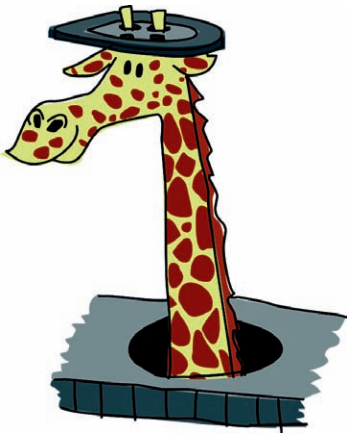
9
Wie man Klassen und Objekte benutzt

Was haben eine Giraffe und ein Bürgersteig gemeinsam? Beide sind *Dinge*, die man im Deutschen als *Substantive* und in Python als *Objekte* bezeichnet.

Das Konzept von *Objekten* ist in der Computerwelt sehr wichtig. Mit Objekten wird der Code in einem Programm strukturiert und Dinge in kleinere Stücke aufgeteilt, damit es leichter wird, kompliziertere Ideen zu verwirklichen. (In Kapitel 5 haben wir schon mit einem Objekt gearbeitet, dem Schildkröten-Zeichenstift `Pen`).

Um wirklich zu verstehen, wie Objekte in Python funktionieren, müssen wir einmal kurz über unterschiedliche Typen von Objekten nachdenken. Gehen wir einmal von Giraffen und Bürgersteigen aus.

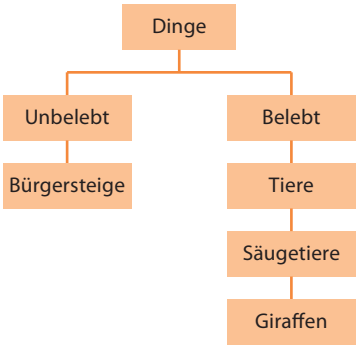
Eine Giraffe ist ein bestimmter Typ von Säugetier, das wiederum einen Typ von Tier darstellt. Eine Giraffe ist auch ein belebtes Objekt – es lebt.



Jetzt denken wir einmal an einen Bürgersteig. Außer, dass er kein lebendiges Ding ist, gibt es kaum etwas über ihn zu sagen. Nennen wir ihn ein unbelebtes Objekt (schließlich lebt er nicht). Die Begriffe *Säugetier*, *Tier*, *belebt* und *unbelebt* sind alles Möglichkeiten, um Dinge zu klassifizieren.

9.1 Dinge in Klassen aufteilen

In Python werden Objekte durch sogenannte *Klassen* definiert. Wir können uns das so vorstellen, dass wir Objekte in Gruppen klassifizieren. Hier siehst Du ein Baumdiagramm der Klassen, in die Giraffen und Bürgersteige aufgrund unserer bisherigen Definitionen passen würden:



Die Hauptklasse ist `Dinge`. Unter der Klasse `Dinge` haben wir die Klassen `Belebt` und `Unbelebt` angelegt. Diese beiden werden weiter aufgeteilt. In der Klasse `Unbelebt` gibt es einfach nur noch die `Bürgersteige`. In der Klasse `Belebt` geht es aber noch weiter: erst mit `Tiere`, dann mit `Säugetiere` und schließlich mit `Giraffen`.

Mit Klassen können wir Abschnitte des Python-Codes strukturieren. Alle Dinge, die Pythons Modul `turtle` kann (also sich vorwärts, rückwärts, nach links und rechts bewegen), sind Funktionen der Klasse `Pen`. Ein Objekt kann man sich als Angehörigen einer Klasse vorstellen, und man kann so viele Objekte in eine Klasse packen, wie man möchte – dazu folgt bald mehr.

Jetzt erzeugen wir den gleichen Satz von Klassen von oben nach unten, wie wir ihn im Baumdiagramm oben sehen. Die Klassen definieren wir durch das Schlüsselwort `class`, hinter das wir den Klassennamen schreiben. Da `Dinge` die allgemeinste Klasse darstellt, erzeugen wir sie als erste:

```
>>> class Dinge:
    pass
```

Wir nennen die Klasse `Dinge` und verwenden die Anweisung `pass`, um Python dadurch mitzuteilen, dass wir keine weiteren Informationen eingeben.

Als Nächstes fügen wir die anderen Klassen hinzu und bauen einige Verbindungen zwischen ihnen auf.

Kinder und Eltern

Sobald eine Klasse Teil einer anderen Klasse wird, ist sie ein *Kind* dieser Klasse, die wiederum zur *Elternklasse* wird. Klassen können gleichzeitig sowohl Kinder- als auch Elternklassen von anderen Klassen sein. In unserem Baumdiagramm ist die Klasse über einer Klasse deren Elternklasse und die Klasse darunter ihr Kind. `Unbelebt` und `Belebt` sind jeweils Kinder der Klasse `Dinge`, `Dinge` ist also deren Elternklasse. Damit Python weiß, dass eine Klasse Kind einer anderen Klasse ist, schreiben wir den Namen der Elternklasse in Klammern hinter den Namen der neuen Klasse:

```
>>> class Unbelebt(Dinge):
    pass

>>> class Belebt(Dinge):
    pass
```

Hier haben wir zuerst eine Klasse namens `Unbelebt` erzeugt und Python mit dem Code `class Unbelebt(Dinge)` mitgeteilt, dass deren Elternklasse `Dinge` ist. Als Nächstes erzeugen wir die Klasse `Belebt` und sagen Python, dass ihre Elternklasse ebenso `Dinge` ist, indem wir `class Belebt(Dinge)` schreiben.

Probieren wir jetzt das Gleiche mit der Klasse `Bürgersteig`. Wir erzeugen die Klasse `Bürgersteig` mit der Elternklasse `Unbelebt`:

```
>>> class Bürgersteig(Unbelebt):
    pass
```

Und die Klassen Tiere, Säugetiere und Giraffen können wir durch deren Elternklassen ebenfalls strukturieren:

```
>>> class Tiere(Belebt):
    pass

>>> class Säugetiere(Tiere):
    pass

>>> class Giraffen(Säugetiere):
    pass
```

9.2 Klassen Objekte hinzufügen

Jetzt haben wir einen Haufen Klassen, aber wie wäre es, wir könnten ein paar Dinge in diese Klassen tun? Sagen wir einmal, wir hätten eine Giraffe, die Wiegand heißt. Wir selbst wissen zwar, dass Wiegand in die Klasse der Giraffen gehört, aber wie sagen wir es dem Programm, dass es sich bei Wiegand um eine einzelne Giraffe handelt? Einfach, indem wir Wiegand ein *Objekt* (manchmal

wird auch der Begriff *Instanz* gebraucht) der Klasse `Giraffen` nennen. Um nun Wiegand in Python »einzuführen«, benutzen wir diese Programmzeile:

```
>>> wiegand = Giraffen()
```

Mit diesem Code sagst Du Python, dass es ein Objekt in der Klasse `Giraffen` anlegen soll und dieses Objekt der Variable `wiegand` zuweisen soll. Wie bei einer Funktion stehen auch hier nach dem Klassennamen zwei Klammern. Später werden wir in diesem Kapitel noch sehen, wie man Objekte erzeugt und dabei in den Klammern Parameter verwendet.

Aber was macht jetzt das Objekt `wiegand`? Nun, im Moment eigentlich nichts. Um etwas mit unseren Objekten anfangen zu können, müssen wir auch Funktionen definieren, die mit den Objekten in dieser Klasse benutzt werden können. Anstatt einfach nur das Schlüsselwort `pass` direkt nach der Definition der Klasse einzugeben, können wir der Klasse Funktionen hinzufügen.

9.3 Funktionen von Klassen definieren

In Kapitel 8 haben wir die Funktionen eingeführt, mit denen wir Code wiederverwenden können. Wenn wir eine Funktion definieren, die zu einer Klasse gehört, machen wir das genau so wie bei jeder anderen Funktion. Der einzige Unterschied besteht darin, dass wir sie unter der Klassendefinition einrücken. Hier ist zum Beispiel eine Funktion, die *nicht* zu einer Klasse gehört:

```
>>> def Dies_ist_eine_normale_Funktion():
        print('Ich bin eine normale Funktion')
```

Und hier sind ein paar Funktionen, die zu einer Klasse *gehören*:

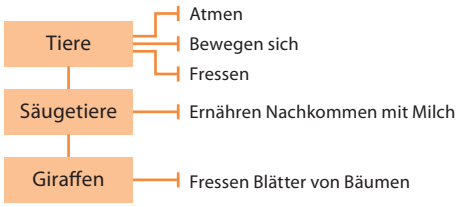
```
>>> class DiesIstMeineKomischeKlasse:
        def Dies_ist_eine_Klassenfunktion():
                print('Ich bin eine Klassenfunktion')
        def Dies_ist_auch_eine_Klassenfunktion():
                print('Ich bin auch eine Klassenfunktion. Siehste?')
```

Klasseneigenschaften als Funktionen hinzufügen

Denke noch einmal an die Kinderklassen der Klasse `Belebt`, die wir auf Seite 87 definiert haben. Wir können jeder Klasse *Eigenschaften* zuweisen, die beschreiben, was die Klasse ist und was sie tun kann. Eine solche Eigenschaft haben alle Angehörigen dieser Klasse (und deren Kinder) gemeinsam.

Was haben zum Beispiel alle Tiere gemeinsam? Nun, da wäre zunächst einmal das Atmen. Sie bewegen sich auch noch und fressen. Wie ist das jetzt mit den Säugetieren? Säugetiere ernähren ihre Nachkommen mit Milch. Und auch sie

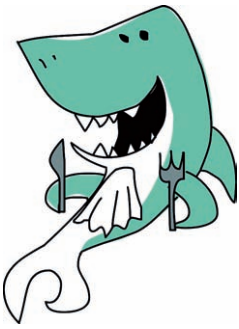
atmen, bewegen sich und fressen. Von Giraffen wissen wir, dass sie Blätter hoch oben von Bäumen fressen und, wie alle Säugetiere, ihre Nachkommen mit Milch ernähren, atmen, sich bewegen und fressen. Wenn wir alle diese Eigenschaften in ein Baumdiagramm zeichnen, sieht das so aus:



Diese Eigenschaften kann man sich als Tätigkeiten oder *Funktionen* vorstellen – Dinge, die ein Objekt einer Klasse tun kann.

Um einer Klasse eine Funktion zuzuweisen, verwenden wir das Schlüsselwort `def`. Die Klasse `Tiere` sieht demnach so aus:

```
>>> class Tiere(Belebt):
        def atmen(self):
                pass
        def bewegen(self):
                pass
        def fressen(self):
                pass
```



In der ersten Zeile dieses Listings beschreiben wir die Klasse wie schon zuvor. Anstatt nun aber in der nächsten Zeile direkt das Schlüsselwort `pass` zu verwenden, definieren wir die Funktion `atmen` und geben ihr einen Parameter: `self`. Mit dem Parameter `self` kann eine Funktion innerhalb der Klasse eine andere in dieser Klasse (und ihrer Elternklasse) aufrufen. Wir werden diesen Parameter später noch in Aktion sehen.

In der nächsten Zeile teilt das Schlüsselwort `pass` Python mit, dass wir keine weiteren Informationen zu der Funktion `atmen` liefern, da diese Funktion jetzt noch nichts tun soll. Genauso fügen wir die Funktionen `bewegen` und `fressen` hinzu, da auch diese noch nichts machen. Wir werden die Klassen bald neu erstellen und ordentlichen Code in die Funktionen schreiben. Programme werden häufig auf diese Weise entwickelt. Programmierer erzeugen oft Klassen, die nichts tun, um zunächst herauszufinden, was die Klassen tun sollen, und fügen die Details der einzelnen Funktionen dann später ein.

Wir können den anderen beiden Klassen, `Säugetiere` und `Giraffen`, auch Funktionen zuweisen. Jede Klasse wird dann die Eigenschaften (die Funktionen)

ihrer Elternklasse nutzen können. Das bedeutet, dass Du nicht *eine* komplizierte Klasse erzeugen musst, sondern diejenigen Funktionen in die höchste Elternklasse legst, für die die Eigenschaften zutreffen sollen. (So kannst Du Deine Klassen einfacher machen, und sie sind dadurch leichter zu verstehen.)

```
>>> class Säugetiere(Tiere):
        def ernähren_Nachkommen_mit_Milch(self):
                pass

>>> class Giraffen(Säugetiere):
        def fressen_Blätter_von_Bäumen(self):
                pass
```

9.4 Wozu braucht man Klassen und Objekte?

Jetzt haben wir unseren Klassen Funktionen hinzugefügt, aber wofür brauchen wir denn eigentlich Klassen und Objekte, wenn man auch einfach so Funktionen wie `atme`, `bewegen` und `fressen` schreiben kann?

Um diese Frage zu beantworten, kommt wieder unsere Giraffe Wiegand zum Einsatz, die wir schon vorher als Objekt der Klasse `Giraffen` erzeugt haben:

```
>>> wiegand = Giraffen()
```

Da `wiegand` ein Objekt ist, können wir Funktionen aufrufen (oder ausführen), die zu seiner Klasse (der Klasse `Giraffen`) gehören, und auch die Funktionen seiner Elternklassen. Wir können die Funktionen auf ein Objekt anwenden, indem wir den Operator *Punkt* und den Namen der Funktionen benutzen. Um Wiegand zu sagen, dass er sich bewegen oder fressen soll, rufen wir die Funktionen folgendermaßen auf:

```
>>> wiegand = Giraffen()
>>> wiegand.bewegen()
>>> wiegand.fressen_Blätter_von_Bäumen()
```

Nehmen wir an, Wiegand hätte einen Artgenossen zum Freund, Heribert. Wir erzeugen ein anderes `Giraffen`-Objekt mit dem Namen `heribert`:

```
>>> heribert = Giraffen()
```

Weil wir Objekte und Klassen benutzen, können wir Python genau sagen, welche Giraffe gemeint ist, wenn wir die Funktion `bewegen` auf sie anwenden. Wenn wir beispielsweise möchten, dass Heribert sich bewegt und Wiegand stehen bleiben soll, rufen wir die Funktion `bewegen` mit unserem Objekt `heribert` auf:

```
>>> heribert.bewegen()
```

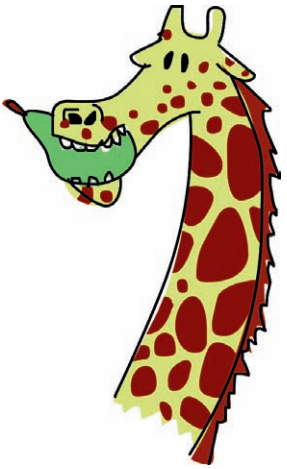
In diesem Fall würde sich nur Heribert bewegen.

Wir ändern jetzt die Klassen ein wenig, um das noch klarer zu machen. Wir fügen anstelle von `pass` jeder Funktion eine `print`-Anweisung hinzu:

```
>>> class Tiere(Belebt):
        def atmen(self):
                print('atmen')
        def bewegen(self):
                print('bewegen')
        def fressen(self):
                print('fressen')

>>> class Säugetiere(Tiere):
        def ernähren_Nachkommen_mit_Milch(self):
                print('Nachkommen ernähren')

>>> class Giraffen(Säugetiere):
        def fressen_Blätter_von_Bäumen(self):
                print('Blätter fressen')
```

Wenn wir jetzt unsere Objekte `wiegand` und `heribert` erzeugen und auf sie Funktionen anwenden, können wir sehen, dass etwas passiert:

```
>>> wiegand = Giraffen()
>>> heribert = Giraffen()
>>> wiegand.bewegen()
bewegen
>>> heribert.fressen_Blätter_von_Bäumen()
Blätter fressen
```



In den ersten beiden Zeilen erzeugen wir die Variablen `wiegand` und `heribert`, die nun Objekte der Klasse Giraffen sind. Als Nächstes rufen wir die Funktion `bewegen` für `wiegand` auf, und Python gibt in der nächsten Zeile `bewegen` aus. Auf die gleiche Weise rufen wir für `heribert` die Funktion `fressen_Blätter_von_Bäumen` auf, und Pyton zeigt `Blätter fressen` an. Wenn es echte Giraffen und nicht nur Objekte in Python wären, würde die eine jetzt laufen und die andere fressen.

9.5 Objekte und Klassen bei Bildern

Wie wäre es mit einem grafischen Ansatz bei Objekten und Klassen?

Lass uns dazu zum Modul `turtle` zurückkehren, mit dem wir in Kapitel 5 herumgespielt haben. Wenn wir `turtle.Pen()` eingeben, erzeugt Python ein Objekt in der Klasse `Pen`, die wiederum im Modul `turtle` enthalten ist. Wir können – genau wie bei den beiden Giraffen – zwei Schildkröten-Objekte erzeugen (mit Namen Amanda und Käte):

```
>>> import turtle
>>> amanda = turtle.Pen()
>>> käte = turtle.Pen()
```

Jedes Schildkröten-(`turtle`-)Objekt gehört jetzt der Klasse `Pen` an.

Jetzt werden die Objekte so richtig nützlich. Dadurch, dass wir unsere Schildkröten-Objekte erzeugt haben und für jedes von ihnen Funktionen aufrufen können, können sie unabhängig voneinander zeichnen. Probiere es aus:

```
>>> amanda.forward(50)
>>> amanda.right(90)
>>> amanda.forward(20)
```

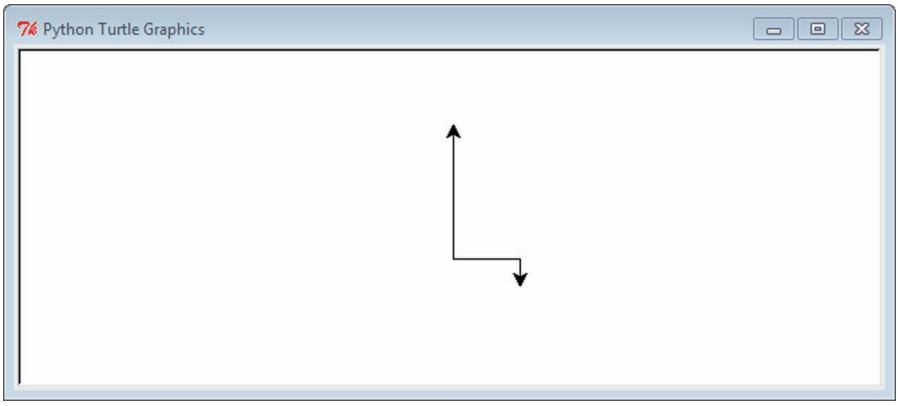
Mit dieser Reihe von Anweisungen sagen wir Amanda, dass sie sich 50 Pixel vorwärts bewegen soll, dann um 90 Grad nach rechts abbiegen und sich anschließend 20 Pixel nach unten bewegen soll. Denk daran, dass sich die Schildkröten immer nach rechts schauend in Bewegung setzen.

Jetzt ist Käte mit der Bewegung dran:

```
>>> käte.left(90)
>>> käte.forward(100)
```

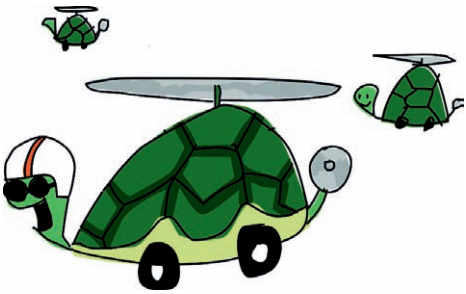
Wir sagen Käte also, dass sie um 90 Grad nach links abbiegen und dann 100 Pixel vorwärts gehen soll, sodass sie am Ende nach oben schaut.

Bis jetzt haben wir eine Linie mit Pfeilspitzen, die in zwei verschiedene Richtungen zeigen, wobei jede Pfeilspitze für ein unterschiedliches Schildkröten-Objekt steht: Amanda zeigt nach unten, Käte nach oben.

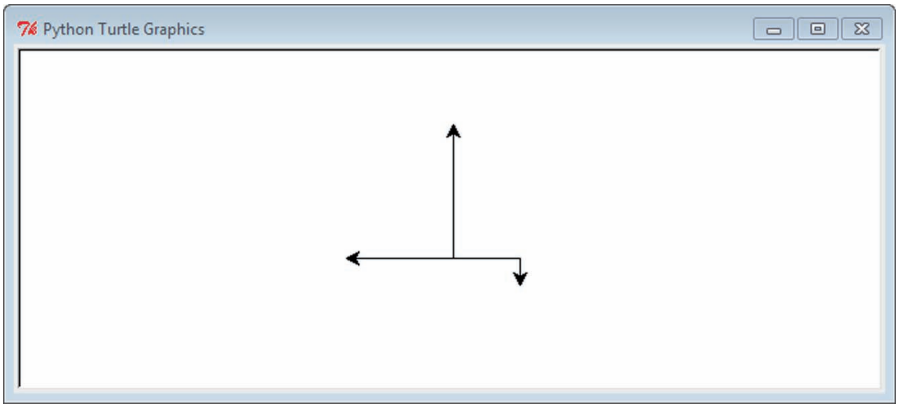


Jetzt tun wir noch eine dritte Schildkröte, Jakob, dazu und bewegen auch ihn, ohne Käte und Amanda zu stören:

```
>>> jakob = turtle.Pen()
>>> jakob.left(180)
>>> jakob.forward(80)
```



Als Erstes erzeugen wir wieder ein `Pen`-Objekt namens `jakob` und drehen es 180 Grad nach links. Dann bewegen wir ihn 80 Pixel vorwärts. Mit drei Schildkröten sieht unsere Zeichnung so aus:



Beachte, dass jedes Mal, wenn wir `turtle.Pen()` aufrufen, um eine Schildkröte zu erzeugen, ein neues, unabhängiges Objekt entsteht. Jedes dieser Objekte gehört noch zu der Klasse `Pen`, sodass wir dieselben Funktionen auf jedes Objekt anwenden können. Weil wir jedoch Objekte verwenden, können wir jede Schildkröte einzeln bewegen. Wie auch unsere unabhängigen Giraffen-Objekte (Wiegand und Heribert) sind Amanda, Käte und Jakob unabhängige Schildkröten-Objekte. Wenn wir ein neues Objekt mit dem einem Objektnamen erzeugen würden, den wir schon zuvor verwendet haben, verschwindet das alte Objekt nicht unbedingt. Probiere es selbst aus: Erzeuge eine zweite Käte, und versuche, sie zu bewegen.

9.6 Weitere nützliche Eigenschaften von Objekten und Klassen

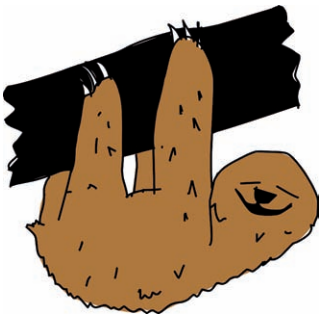
Klassen und Objekte erleichtern das Zusammenfassen von Funktionen. Sie sind auch sehr nützlich, wenn wir ein Programm in kleineren Abschnitten angehen wollen.

Stellen wir uns zum Beispiel eine richtig große Software vor, so etwas wie eine Textverarbeitung oder ein 3D-Computerspiel. Für die allermeisten Menschen ist es einfach unmöglich, große Programme wie diese zu verstehen, da sie aus so viel

Code bestehen. Aber sobald man diese Monsterprogramme in kleine Stücke unterteilt, kann man die Stücke verstehen. Natürlich nur, wenn man die Programmiersprache versteht.

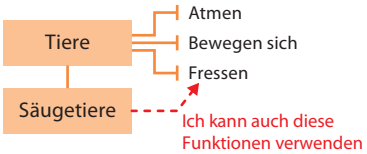
Wenn Du ein großes Programm schreibst, hilft Dir die Unterteilung auch dabei, Dir die Arbeit mit anderen Programmierern zu teilen. Die kompliziertesten Programme, die Du benutzt (wie etwa Dein Webbrowser), wurden von vielen Leuten oder ganzen Teams gemeinsam geschrieben, die jeweils an unterschiedlichen Teilen gleichzeitig gearbeitet haben – und das auf der ganzen Welt verteilt.

Jetzt stell Dir einmal vor, wir wollten einige unserer Klassen, die wir in diesem Kapitel erzeugt haben (`Tiere`, `Säugetiere` und `Giraffen`) erweitern. Wir sind aber so beschäftigt, dass wir unsere Freunde bitten, uns dabei zu helfen. Wir können uns die Arbeit am Code so aufteilen, dass der eine Freund an der Klasse `Tiere`, ein anderer an der Klasse `Säugetiere` und ein weiterer an der Klasse `Giraffen` arbeitet.



9.7 Geerbte Funktionen

Denjenigen von Euch, die gut aufgepasst haben, ist vielleicht aufgefallen, dass der Freund, der mit der Klasse `Giraffen` arbeitet, Glück gehabt hat. Jede Funktion, die die Leute für die Klassen `Tiere` und `Säugetiere` erzeugt haben, kann nämlich auch von der Klasse `Giraffen` verwendet werden. Die Klasse `Giraffen` *erbt* also die Funktionen der Klasse `Säugetiere`, die wiederum die Funktionen der Klasse `Tiere` erbt. Anders gesagt: Wenn wir ein `Giraffen`-Objekt erzeugen, können wir sowohl die Funktionen der Klasse `Giraffen` nutzen als auch die Funktionen, die in den Klassen `Säugetiere` und `Tiere` definiert worden sind. Und wenn wir ein `Säugetier`-Objekt erzeugen, können wir genauso die Funktionen in der `Säugetier`-Klasse sowie in deren `Elternklasse` Tiere verwenden.



Obwohl Wiegand ein Objekt der Klasse `Giraffen` ist, können wir die Funktion `bewegen` aufrufen, die wir in der Klasse `Tiere` definiert haben, da jede Funktion der Elternklasse den Kinderklassen zur Verfügung steht:

```
>>> wiegand = Giraffen()
```

```
>>> wiegand.bewegen()
bewegen
```

Es ist sogar so, dass alle Funktionen, die wir in den Klassen `Tiere` und `Säugetiere` definiert haben, von unserem Objekt `wiegand` aufgerufen werden können, da die Funktionen vererbt werden:

```
>>> wiegand = Giraffen()
>>> wiegand.atmen()
atmen
>>> wiegand.fressen()
fressen
>>> wiegand.ernährt_Nachkommen_mit_Milch()
Nachkommen ernähren
```

9.8 Funktionen, die andere Funktionen aufrufen

Wenn wir für ein Objekt Funktionen aufrufen, schreiben wir dazu den Variablennamen des Objekts. Hier rufen wir zum Beispiel die Funktion `bewegen` für die Giraffe Wiegand auf:

```
>>> wiegand.bewegen()
```

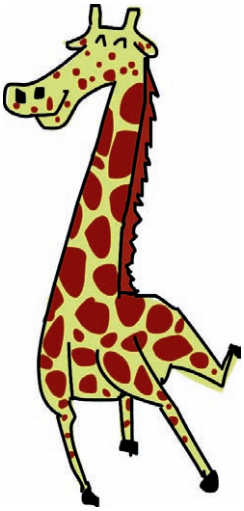
Damit eine Funktion in der Klasse `Giraffen` die Funktion `bewegen` aufruft, würden wir stattdessen den Parameter `self` einsetzen. Mit dem `self`-Parameter kann eine Funktion in der Klasse eine andere aufrufen. Wenn wir zum Beispiel der Klasse `Giraffen` die Funktion `finde_Futter` hinzufügen, schreiben wir:

```
>>> class Giraffen(Säugetiere):
        def finde_Futter(self):
                self.bewegen()
                print('Ich habe Futter gefunden!')
                self.fressen()
```

Jetzt haben wir eine Funktion erzeugt, die zwei weitere Funktionen kombiniert. Das wird beim Programmieren häufig gemacht. Dabei schreibt man oft eine Funktion, die etwas Nützliches tut, um sie dann innerhalb einer anderen Funktion verwenden zu können. (Wir machen das in Kapitel 14, wo wir noch kompliziertere Funktionen erzeugen werden, um ein Spiel zu schreiben.)

Wir benutzen jetzt `self`, um der Klasse `Giraffen` einige Funktionen hinzuzufügen:

```
>>> class Giraffen(Säugetiere):
        def finde_Futter(self):
                self.bewegen()
                print('Ich habe Futter gefunden!')
                self.fressen()
        def fressen_Blätter_von_Bäumen(self):
                self.fressen()
        def mach_ein_Tänzchen(self):
                self.bewegen()
                self.bewegen()
                self.bewegen()
                self.bewegen()
```



Wir benutzen die Funktionen `fressen` und `bewegen` der Elternklasse `Tiere`, um `fressen_Blätter_von_Bäumen` und `mach_ein_Tänzchen` für die Klasse `Giraffen` zu definieren, weil sie geerbte Funktionen sind. Dadurch, dass wir Funktionen hinzufügen, die auf diese Weise weitere Funktionen aufrufen, können wir eine einzelne Funktion aufrufen, die mehr als nur eine Sache tun kann. Wenn wir jetzt die Funktion `mach_ein_Tänzchen` aufrufen, siehst Du, was passiert – unsere Giraffe bewegt sich viermal (das heißt, der Text »bewegen« wird viermal angezeigt):

```
>>> wiegand = Giraffen()
>>> wiegand.mach_ein_Tänzchen()
bewegen
bewegen
bewegen
bewegen
```

9.9 Ein Objekt initialisieren

Manchmal wollen wir bei der Erstellung eines Objekts einige Werte (auch *Attribute* genannt) für später festlegen. Sobald wir ein Objekt *initialisiert* haben, ist es bereit für den Einsatz.

Stell Dir zum Beispiel vor, Du willst die Anzahl der Fellflecken auf unseren Giraffen-Objekten festlegen, während diese Objekte erzeugt werden – wenn sie also initialisiert werden. Dafür erzeugen wir eine `__init__`-Funktion (achte darauf, dass es zwei Unterstrich-Zeichen auf jeder Seite gibt, also insgesamt vier). Die `init`-Funktion ist eine besondere Art von Funktionen in Python-Klassen und

muss daher genau diesen Namen tragen. Mit der Funktion `init` legt man die Attribute eines Objekts bei dessen Erzeugung fest. Python ruft dann diese Funktion automatisch auf, sobald wir ein neues Objekt erzeugen. Und das geht so:

```
>>> class Giraffen:
        def __init__(self, Flecken):
                self.Giraffenflecken = Flecken
```

Als Erstes definieren wir die Funktion `init` mit den beiden Parametern `self` und `Flecken`: `def __init__(self, Flecken):`. Wie bei den anderen Funktionen, die wir in der Klasse definiert haben, braucht auch die `init`-Klasse `self` als ersten Parameter. Als Nächstes legen wir den Parameter `Flecken` als Variable eines Objekts (also seines Attributs) namens `Giraffenflecken` mit dem Parameter `self` fest: `self.Giraffenflecken = Flecken`. Du kannst Dir diese Zeile Code auch vorstellen als: »Nimm den Wert des Parameters `Flecken`, und speichere ihn für später (mit der Objekt-Variable `Giraffenflecken`.« Genauso, wie eine Funktion in einer Klasse eine andere mit dem Parameter `self` aufrufen kann, kann man mit `self` auch Parameter und Variablen in der Klasse aufrufen.

Wenn wir nun als Nächstes ein paar neue Giraffen-Objekte (Oswald und Gertrud) erzeugen und deren Anzahl von Flecken anzeigen lassen wollen, siehst Du die Initialisierungsfunktion in Aktion:

```
>>> oswald = Giraffen(100)
>>> gertrud = Giraffen(150)
>>> print(oswald.Giraffenflecken)
100
>>> print(gertrud.Giraffenflecken)
150
```

Als Erstes erzeugen wir ein Objekt in der `Giraffen`-Klasse mit dem Parameter-Wert 100. Dadurch wird die `__init__`-Funktion aufgerufen und 100 als Wert für den Parameter `Flecken` festgelegt. Als Nächstes erzeugen wir noch ein Objekt in der Klasse `Giraffen`, diesmal mit 150. Als Letztes geben wir die Objektvariable `Giraffenflecken` für jedes der Giraffen-Objekte aus. Wir sehen, dass die Ergebnisse 100 und 150 sind: Es hat also geklappt!

Achte aber auf folgenden Unterschied: Wenn man ein Objekt einer Klasse erzeugt, wie etwa oben `oswald`, können wir uns auf dessen Variablen oder Funktionen beziehen, indem wir den Punkt-Operator und den Namen der Variable oder Funktion hinschreiben, die wir benutzen wollen (z.B. `oswald.Giraffenpunkte`). Wenn wir dagegen Funktionen innerhalb einer Klasse erzeugen, beziehen wir uns auf die gleichen Variablen (und andere Funktionen) mit dem Parameter `self` (`self.Giraffenpunkte`).

9.10 Was Du gelernt hast

In diesem Kapitel haben wir Klassen benutzt, um Kategorien von Dingen zu erzeugen und haben Objekte (oder Instanzen) dieser Klassen erzeugt. Du hast gelernt, wie die Kinder einer Klasse die Funktionen der Elternklasse erben und dass zwei Objekte, obwohl sie den gleichen Namen haben, nicht unbedingt Klone (also genau gleich) sein müssen. Ein Giraffen-Objekt zum Beispiel kann seine eigene Anzahl von Flecken haben. Du hast auch gelernt, wie man Funktionen für ein Objekt aufruft (oder ausführen lässt) und wie man Objektvariablen nutzt, um Werte in diesen Objekten zu speichern. Am Ende haben wir den Parameter `self` in Funktionen genutzt, um uns auf andere Funktionen und Variablen zu beziehen. Dies sind grundlegende Konzepte in Python, denen Du immer wieder in diesem Buch begegnen wirst.

9.11 Programmier-Puzzles

Den Sinn hinter einigen Konzepten in diesem Kapitel wirst Du immer besser verstehen, je häufiger Du sie anwendest. Probiere sie mit den folgenden Beispielen aus, und überprüfe die Antworten auf *www.dpunkt.de/python*.

#1: Der Giraffen-Schiebetanz

Füge der Klasse `Giraffen` Funktionen hinzu, um die linken und rechten Hufe der Giraffe vorwärts und rückwärts zu bewegen. Eine Funktion zum Bewegen der linken Hufe nach vorn könnte so aussehen:

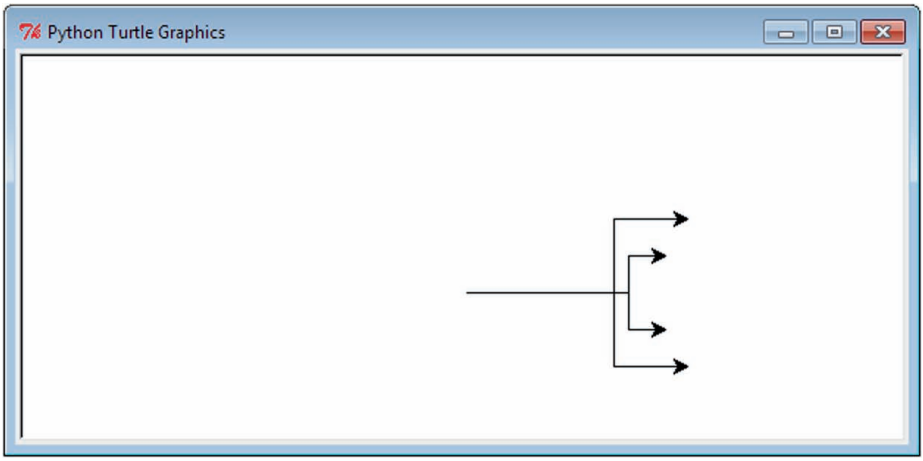
```
>>> def linke_Hufe_vor(self):
        print('linke Hufe vor')
```

Erstelle dann eine Funktion namens `tanzen`, um Wiegand das Tanzen beizubringen (die Funktion wird die vier Hufe-Funktionen aufrufen, die Du gerade erzeugt hast). Nach dem Aufrufen dieser neuen Funktion ist das Ergebnis ein kleiner Tanz:

```
>>> wiegand = Giraffen()
>>> wiegand.tanzen()
linke Hufe vor
linke Hufe zurück
rechte Hufe vor
rechte Hufe zurück
linke Hufe zurück
rechte Hufe zurück
rechte Hufe vor
linke Hufe zurück
```

#2: Schildkröten-Heugabel

Erzeuge das folgende Bild einer Heugabel mit vier Schildkröten-`Pen`-Objekten (die genaue Länge der Linien ist unwichtig). Denke daran, dass Du dazu erst das Modul `turtle` importieren musst!



10 Pythons eingebaute Funktionen

Mit Python installierst Du einen gut ausgestatteten Programmier-Werkzeugkasten, in dem viele Funktionen und Module enthalten sind, die Du nutzen kannst. Wie ein zuverlässiger Hammer oder Schraubenschlüssel können Dir diese eingebauten Werkzeuge – in Wirklichkeit sind es natürlich Einheiten von Code – das Schreiben von Programmen sehr erleichtern.

Wie Du schon in Kapitel 8 gelernt hast, müssen Module importiert werden, bevor Du sie benutzen kannst. Die in Python *eingebauten* Module müssen dagegen nicht erst importiert werden. Sie sind sofort verfügbar, sobald die Python-Shell startet. In diesem Kapitel schauen wir uns einige nützliche eingebaute Funktionen an und werden uns dann auf eine von ihnen konzentrieren: auf die Funktion `open`, mit der Du Dateien öffnen kannst, um in ihnen zu lesen und zu schreiben.

10.1 Eingebaute Funktionen verwenden

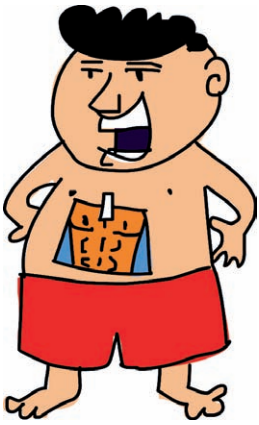
Wir werden uns jetzt 12 eingebaute Funktionen ansehen, die häufig von Python-Programmierern verwendet werden. Ich werde beschreiben, was sie tun und wie man sie benutzt. Anschließend zeige ich in Beispielen, wie sie uns in unseren Programmen helfen können.

Die abs-Funktion

Die Funktion `abs` gibt den *Betrag* einer Zahl zurück. Der Betrag ist der Wert einer Zahl ohne Vorzeichen. Der Betrag von 10 ist 10, und der Betrag von -10 ist ebenfalls 10.

Um die Funktion `abs` zu benutzen, rufst Du sie einfach mit einer Zahl oder Variable als ihr Parameter auf:

```
>>> print(abs(10))
10
>>> print(abs(-10))
10
```



Mit der Funktion `abs` kannst Du so etwas wie die absolute Bewegung einer Figur in einem Spiel berechnen, egal, in welche Richtung sich diese Figur bewegt. Nehmen wir zum Beispiel an, eine Figur bewegt sich drei Schritte nach rechts (+3) und tritt dann zehn Schritte nach links (-10). Wenn wir uns nicht um die Richtung kümmern würden, wären die absoluten Werte dieser Zahlen 3 und 10. Das könntest Du zum Beispiel in einem Brettspiel gebrauchen, bei dem Du zweimal würfelst und dann je nach Gesamtpunktzahl die Figur eine Gesamtzahl an Schritten auf dem Brett machen lässt. Wenn wir jetzt die Anzahl der Schritte in einer Variablen speichern, können wir bestimmen, ob die Figur mit dem Code unten bewegt wird. Falls sich der Spieler dazu entschlossen hat, sich zu bewegen (in diesem Fall wird nur die Mitteilung »Figur bewegt sich« angezeigt), soll eine Nachricht erscheinen:

```
>>> Schritte = -3
>>> if abs(Schritte) > 0:
        print('Figur bewegt sich')
```

Hätten wir nicht die Funktion `abs` verwendet, würde die `if`-Anweisung so aussehen:

```
>>> Schritte = -3
>>> if Schritte < 0 or Schritte > 0:
        print('Figur bewegt sich')
```

Wie Du siehst, wird durch die Funktion `abs` die `if`-Anweisung ein wenig kürzer und verständlicher.

Die boolesche Funktion

Der Name der Funktion `bool` steht für »boolesch«, also für eine logische Funktion, mit der Programmierer einen Datentyp beschreiben, der einen von zwei möglichen Werten annehmen kann – meist entweder *wahr* oder *falsch*.

Die Funktion `bool` nimmt einen Parameter auf und gibt je nach dessen Wert `True` (wahr) oder `False` (falsch) zurück. Wenn man `bool` bei Zahlen benutzt, ergibt 0 `False` und alle anderen Zahlen ergeben `True`. So kann man `bool` mit einigen Zahlen einsetzen:

```
>>> print(bool(0))
False
>>> print(bool(1))
True
>>> print(bool(1123.23))
True
>>> print(bool(-500))
True
```

Wenn Du `bool` für andere Werte, wie etwa Strings, einsetzt, gibt sie `False` zurück, falls der String keinen Wert enthält (oder anders gesagt, das Schlüsselwort `None` oder einen leeren String). Ansonsten gibt sie `True` zurück:

```
>>> print(bool(None))
False
>>> print(bool('a'))
True
>>> print(bool(' '))
True
>>> print(bool('Was macht ein Clown im Büro? Faxen!'))
True
```

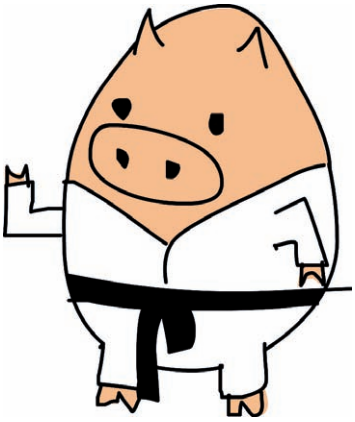
Die Funktion `bool` gibt `False` auch bei Listen, Tupeln und Maps zurück, die keine Werte enthalten, oder `True`, wenn doch Werte vorhanden sind:

```
>>> meine_komische_Liste = []
>>> print(bool(meine_komische_Liste))
False
>>> meine_komische_Liste = ['k', 'o', 'm', 'i', 's', 'c', 'h']
>>> print(bool(meine_komische_Liste))
True
```

Wozu ist die Funktion `bool` also gut? Du könntest sie zum Beispiel einsetzen, wenn Du entscheiden lassen musst, ob ein Wert gesetzt wurde oder nicht. Wenn wir etwa Leute, die unser Programm benutzen, bitten, ihr Geburtsjahr einzugeben, dann könnte unsere `if`-Anweisung mit `bool` den eingegebenen Wert prüfen:

```
>>> Jahr = input('Geburtsjahr: ')
Geburtsjahr:
>>> if not bool(Jahr.rstrip()):
        print('Du sollst einen Wert bei Deinem Geburtsjahr eintragen')
Du sollst einen Wert bei Deinem Geburtsjahr eintragen
```

Die erste Zeile dieses Beispiels benutzt `input`, um das zu speichern, was jemand mit der Tastatur eingibt, wie etwa die Variable `Jahr`. Nach dem Drücken der Enter-Taste (ohne noch etwas einzutippen) wird der Wert der Enter-Taste in der Variablen gespeichert. (In Kapitel 8 haben wir dazu `sys.stdin.readline()` benutzt, was im Grunde das Gleiche macht.)



In der folgenden Zeile überprüft die `if`-Anweisung den booleschen Wert der Variable nach dem Einsatz der Funktion `rstrip`, die alle Leerzeichen und Enter-Zeichen vom Ende des Strings entfernt. Da der Benutzer in diesem Beispiel nichts eingegeben hat, gibt die Funktion `bool` falsch (False) zurück. Weil die `if`-Anweisung das Schlüsselwort `not` benutzt, ist das wie zu sagen »tue dies, falls die Funktion nicht wahr zurückgibt«, sodass der Code in der nächsten Zeile `Du sollst einen Wert bei Deinem Geburtsjahr eintragen` ausgibt.

Die Funktion dir

Die Funktion `dir` (Abkürzung von *directory*, engl. für »Verzeichnis«) gibt Informationen über jeden Wert zurück. Im Wesentlichen zeigt sie Dir die Funktionen, die mit einem Wert benutzt werden können, in alphabetischer Reihenfolge an.

Wenn Du zum Beispiel alle Funktionen sehen willst, die für einen Listenwert zur Verfügung stehen, gibst Du Folgendes ein:

```
>>> dir(['eine', 'kurze', 'Liste'])
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
```

Die Funktion `dir` funktioniert bei so gut wie allem, zum Beispiel bei Strings, Zahlen, Funktionen, Modulen, Objekten und Klassen. Manchmal jedoch kann man mit den zurückgegebenen Funktionen nicht viel anfangen. Wenn Du zum Beispiel `dir` für die Zahl 1 aufrufst, werden jede Menge Sonderfunktionen angezeigt (solche, die mit Unterstrichen anfangen und enden), die nur Python selbst verwendet

und die uns nicht viel nützen. (Du kannst die meisten von ihnen in der Regel ignorieren):

```
>>> dir(1)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__',
'__float__', '__floor__', '__floordiv__', '__format__', '__ge__',
'__getattribute__', '__getnewargs__', '__gt__', '__hash__',
'__index__', '__init__', '__int__', '__invert__', '__le__',
'__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__',
'__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__',
'__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__',
'__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
'__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
'__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag',
'numerator', 'real', 'to_bytes']
```

Die Funktion `dir` kann auch sehr nützlich sein, wenn Du eine Variable hast und schnell herausfinden möchtest, was Du mit ihr tun kannst. Du kannst zum Beispiel `dir` mit der Variablen `Popcorn`, die einen String-Wert enthält, aufrufen und Dir so alle Funktionen anzeigen lassen, die die Klasse `string` enthält (alle Strings gehören zur Klasse `string`):

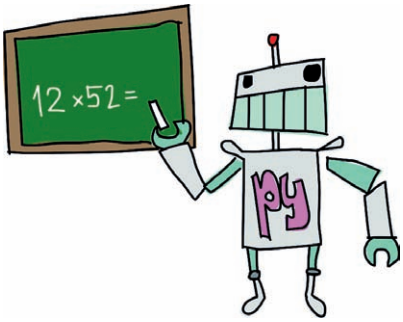
```
>>> Popcorn = 'Ich mag Popcorn!'
>>> dir(Popcorn)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

An dieser Stelle könntest Du jetzt mit `help` (engl. für »Hilfe«) eine kurze Beschreibung der Funktionen in der Liste aufrufen. Hier siehst Du ein Beispiel für die Funktion `upper`:

```
>>> help(Popcorn.upper)
Help on built-in function upper:

upper(...)
        S.upper() -> str
        Return a copy of S converted to uppercase.
```

Kommt Dir die zurückgegebene Information verwirrend vor? Sehen wir sie uns näher an: Die Auslassungspunkte (…) bedeuten, dass `upper` eine eingebaute Funktion der `string`-Klasse ist und dass sie keine Parameter aufnimmt. Der Pfeil (`->`) in der nächsten Zeile bedeutet, dass diese Funktion einen String (`str`) zurückgibt. Die letzte Zeile liefert eine kurze Beschreibung, was die Funktion macht (`upper` erzeugt eine Kopie des Strings in Großbuchstaben).

Die Funktion eval



Die Funktion `eval` (Abkürzung für *evaluation*, engl. für »Auswertung«) nimmt als Parameter einen String und führt ihn aus, als wäre es ein Python-Ausdruck. So wird zum Beispiel durch `eval('print("wow")')` der Befehl `print("wow")` ausgeführt.

Die Funktion `eval` funktioniert bei einfachen Ausdrücken wie diesem hier:

```
>>> eval('10*5')
50
```

Ausdrücke, die über mehr als eine Zeile gehen (wie etwa `if`-Anweisungen), werden nicht ausgewertet:

```
>>> eval('''if True:
         print("das funktioniert überhaupt nicht")''')
Traceback (most recent call last):
  File "<pyshell#1>", line 2, in <module>
    print("das funktioniert überhaupt nicht")''')
  File "<string>", line 1
    if True:
     ^
SyntaxError: invalid syntax
```

Die Funktion `eval` wird häufig genutzt, um Benutzereingaben in Python-Ausdrücke umzuwandeln. Du könntest zum Beispiel ein einfaches Rechenprogramm schreiben, das Aufgaben, die in Python eingegeben werden, liest und dann die Antworten ausrechnet (auswertet).

Da die Benutzereingaben immer als String eingelesen werden, muss Python sie erst in Zahlen und Operatoren umwandeln, bevor es mit ihnen rechnen kann. Die Funktion `eval` erleichtert diese Umwandlung:

```
>>> Deine_Aufgabe = input('Gib eine Rechenaufgabe ein: ')
Gib eine Rechenaufgabe ein: 12*52
>>> eval(Deine_Aufgabe)
624
```

In diesem Beispiel haben wir `input` benutzt, um einzulesen, was der Benutzer in die Variable `Deine_Aufgabe` eingibt. In der nächsten Zeile geben wir den Ausdruck `12*52` ein (vielleicht Dein Alter mit der Anzahl der Wochen eines Jahres multipliziert). Mit `eval` lassen wir die Aufgabe rechnen und das Ergebnis anschließend in der letzten Zeile anzeigen.

Die Funktion exec

Die Funktion `exec` ist wie `eval`, nur dass Du sie in komplexeren Programmen benutzen kannst. Der Unterschied zwischen ihnen besteht darin, dass `eval` einen Wert zurückgibt (etwas, das man in einer Variablen speichern kann) und `exec` das nicht tut. Hier ist ein Beispiel:

```
>>> mein_kleines_Programm = '''print('Schinken')
print('Sandwich')'''
>>> exec(mein_kleines_Programm)
Schinken
Sandwich
```

In den ersten beiden Zeilen erzeugen wir eine Variable mit einem mehrzeiligen String, der zwei `print`-Anweisungen enthält und dann mit `exec` den String ausführt.

Mit `exec` kannst Du Mini-Programme ausführen, die Python aus Dateien einliest – das sind dann Programme innerhalb eines Programms! Das ist vor allem dann praktisch, wenn man lange, komplizierte Anwendungen schreibt. Du könntest zum Beispiel ein Spiel »Roboter-Zweikampf« schreiben, in dem sich zwei Roboter auf dem Monitor bewegen und versuchen, sich gegenseitig anzugreifen. Die Spieler des Spiels würden dann ihre Befehle an die Roboter über kleine Python-Programme geben. Roboter-Zweikampf würde diese Scripts einlesen und sie mit `exec` laufen lassen.

Die Funktion float

Die Funktion `float` wandelt einen String oder eine Zahl in eine *Fließkommazahl* um, also in eine Zahl mit Dezimaltrennzeichen (auch *reelle* Zahl genannt). Die Zahl 10 ist zum Beispiel eine *ganze Zahl*, aber 10.1 und 10.253 sind Fließkom-

mazahlen. Normalerweise schreiben wir solche Zahlen, wie der Name schon sagt, mit einem Komma (10, 1 und 10,253). Python versteht dieses Komma jedoch als String, und deshalb musst Du statt des Kommas den Punkt verwenden.

Du kannst einen String in eine Fließkommazahl umwandeln, indem Du `float` aufrufst:

```
>>> float('12')
12.0
```

Du kannst auch eine Nachkommastelle in dem String verwenden:

```
>>> float('123.456789')
123.456789
```

Mit `float` kannst Du auch Werte, die in Dein Programm eingegeben werden, in richtige Zahlen umwandeln. Dies ist besonders nützlich, wenn Du einen Wert, den ein Benutzer eingibt, mit anderen Werten vergleichen musst. Um zum Beispiel zu prüfen, ob das Alter der Person über einer bestimmten Zahl liegt, könnten wir Folgendes machen:

```
>>> Dein_Alter = input('Gib Dein Alter ein: ')
Gib Dein Alter ein: 20
>>> Alter = float(Dein_Alter)
>>> if Alter > 13:
        print('Du bist %s Jahre zu alt' % (Alter - 13))

Du bist 7.0 Jahre zu alt
```

Die Funktion int

Die Funktion `int` wandelt einen String oder eine Zahl in eine ganze Zahl (engl. *integer*) um. Im Prinzip wird dabei alles nach dem Dezimaltrennzeichen weggelassen. So wandelt man zum Beispiel eine Fließkommazahl in eine Ganzzahl um:

```
>>> int(123.456)
123
```

Bei diesem Beispiel wird ein String in eine Ganzzahl umgewandelt:

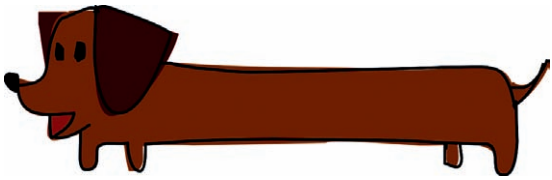
```
>>> int('123')
123
```

Sobald Du aber versuchst, einen String mit einer Fließkommazahl in eine ganze Zahl umzuwandeln, bekommst Du eine Fehlermeldung:

```
>>> int('123.456')
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    int('123.456')
ValueError: invalid literal for int() with base 10: '123.456'
```

Wie Du siehst, kommt eine `ValueError`-Fehlermeldung dabei heraus.

Die Funktion len



Die Funktion `len` gibt die Länge eines Objekts oder die Anzahl der Zeichen bei einem String zurück. Um zum Beispiel die Länge von `Dies ist ein Test-String` zu ermitteln, tust Du Folgendes:

```
>>> len('Dies ist ein Test-String')
24
```

Wenn man diese Funktion bei einer Liste oder einem Tupel verwendet, gibt `len` die Anzahl der Elemente darin zurück:

```
>>> Kreaturenliste = ['Einhorn', 'Zyklop', 'Fee', 'Elfe', 'Drachen',
                      'Troll']
>>> print(len(Kreaturenliste))
6
```

Wenn man sie bei einer Map verwendet, gibt `len` ebenfalls die Anzahl der Elemente zurück:

```
>>> Feindesliste = {'Batman' : 'Joker',
                    'Superman' : 'Lex Luthor',
                    'Spiderman' : 'Green Goblin'}
>>> print(len(Feindesliste))
3
```

Die Funktion `len` ist vor allem bei Schleifen nützlich. Wenn wir zum Beispiel die Indexposition der Elemente einer Liste anzeigen lassen wollen, machen wir das so:

```
>>> Frucht = ['Apfel', 'Banane', 'Mandarine', 'Birne']
❶ >>> Länge = len(Frucht)
❷ >>> for x in range(0, Länge):
❸         print('Die Frucht an Position %s ist %s' % (x, Frucht[x]))

Die Frucht an Position 0 ist Apfel
Die Frucht an Position 1 ist Banane
Die Frucht an Position 2 ist Mandarine
Die Frucht an Position 3 ist Birne
```

Hier speichern wir die Länge der Liste in der Variable `Länge` in ❶ und verwenden diese Variable dann in der Funktion `range`, um unsere Schleife in ❷ zu erzeugen. Wenn wir dann in ❸ durch jedes Element der Liste durchschleifen, geben wir für jedes Element der Liste dessen Position und Wert aus. Du könntest bei einer Liste von Strings mit der Funktion `len` auch einfach jedes zweite oder dritte Element der Liste anzeigen lassen.

Die Funktionen max und min

Die Funktion `max` gibt das größte Element einer Liste, eines Tupels oder eines Strings zurück. Hier tut sie das bei einer Liste von Zahlen:

```
>>> Zahlen = [5, 4, 10, 30, 22]
>>> print(max(Zahlen))
30
```

Bei einem String, der durch Kommata oder Leerzeichen getrennt ist, funktioniert das auch:

```
>>> strings = 's,t,r,i,n,g,S,T,R,I,N,G'
>>> print(max(strings))
t
```

Wie dieses Beispiel zeigt, werden die Buchstaben nach ihrer alphabetischen Reihenfolge sortiert. Kleingeschriebene Buchstaben kommen nach den Großbuchstaben, *t* ist demnach mehr als *T* (d.h. es kommt nach *T*).

Aber Du musst dazu nicht unbedingt Listen, Tupeln oder Strings nehmen. Du kannst die Funktion `max` auch direkt aufrufen und die Elemente, die Du vergleichen möchtest, als Parameter in Klammern dahinter setzen:

```
>>> print(max(10, 300, 450, 50, 90))
450
```

Die Funktion `min` arbeitet genauso wie `max`, nur dass sie das kleinste Element einer Liste, eines Tupels oder eines Strings zurückgibt. Wenn wir wieder unsere Liste mit Zahlen nehmen und `min` statt `max` einsetzen, passiert Folgendes:

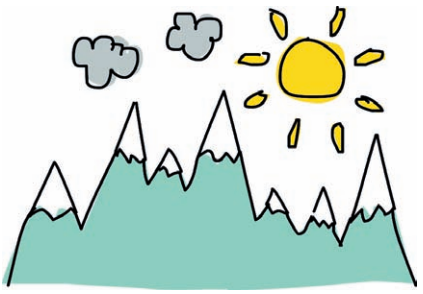
```
>>> Zahlen = [5, 4, 10, 30, 22]
>>> print(min(Zahlen))
4
```

Jetzt stell Dir vor, Du würdest ein Ratespiel mit vier Spielern spielen, bei dem jeder eine Zahl schätzen soll, die unter Deiner Zahl liegt. Sobald nur einer der Spieler eine Zahl darüber schätzt, haben alle Spieler verloren. Wenn sie aber alle eine niedrigere Zahl schätzen, haben sie gewonnen. Wir können mit `max` auch schnell herausfinden, ob alle Schätzungen unter Deiner Zahl liegen:

```
>>> rate_diese_Zahl = 61
>>> Spielerschätzungen = [12, 15, 70, 45]
>>> if max(Spielerschätzungen) > rate_diese_Zahl:
        print('Ups! Ihr habt alle verloren')
else:
        print('Ihr habt gewonnen')

Ups! Ihr habt alle verloren
```

In diesem Beispiel speichern wir die zu ratende Zahl in der Variable `rate_diese_Zahl`. Die Schätzungen der Mitspieler werden in der Liste `Spielerschätzungen` gespeichert. Die `if`-Anweisung vergleicht die höchste Schätzung mit der Zahl in `rate_diese_Zahl`, und sobald einer der Spiele eine Zahl darüber schätzt, zeigen wir die Mitteilung »Ups! Ihr habt alle verloren« an.

Die Funktion range



Die Funktion `range` wird, wie wir schon zuvor gesehen haben, hauptsächlich in `for`-Schleifen verwendet, um einen Codeabschnitt so oft wie gewünscht durchlaufen zu lassen. Die ersten beiden Parameter, die man `range` gibt, sind *Start* und *Stop*. In dem zurückliegenden Beispiel mit der Funktion `range` hast Du diese beiden Parameter in einer Schleife mit der Funktion `len` in Aktion gesehen.

Die Zahlen, die `range` erzeugt, beginnen mit der Zahl, die als erster Parameter angegeben wird, und enden mit der Zahl, die als zweiter Parameter angegeben wird. Hier siehst Du zum Beispiel, was passiert, wenn Du `range` die Zahlen zwischen 0 und 5 erzeugen lässt:

```
>>> for x in range(0, 5):
        print(x)

0
1
2
3
4
```

Die Funktion `range` gibt also ein bestimmtes Objekt, das man *Iterator* nennt, zurück, das dann eine Aktion eine bestimmte Anzahl von Malen wiederholt. In diesem Fall gibt sie bei jedem Aufruf die nächsthöhere Zahl zurück.

Du kannst einen Iterator in eine Liste umwandeln (mit der Funktion `list`). Wenn Du dann den zurückgegebenen Wert nach dem Aufruf von `range` ausgibst, werden Dir auch die Zahlen darin angezeigt:

```
>>> print(list(range(0, 5)))
[0, 1, 2, 3, 4]
```

Du kannst der Funktion `range` noch einen dritten Parameter hinzufügen: `step` (Schrittweite). Wenn der Wert für `step` nicht mit angegeben wird, wird automatisch die Zahl 1 verwendet. Was passiert nun, wenn wir die Zahl 2 als Schrittweite einfügen? Hier ist das Ergebnis:

```
>>> zähle_in_Zweierschritten = list(range(0, 30, 2))
>>> print(zähle_in_Zweierschritten)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

Die Zahlen in der Liste steigen immer um den Betrag von 2 an, und die Liste endet bei 28, was 2 weniger als 30 ist. Du kannst auch negative Schrittweiten verwenden:

```
>>> zähle_in_Zweierschritten_abwärts = list(range(40, 10, -2))
>>> print(zähle_in_Zweierschritten_abwärts)
[40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14, 12]
```

Die Funktion sum

Die Funktion `sum` zählt die Elemente einer Liste zusammen und gibt die Summe zurück. Hier siehst Du ein Beispiel:

```
>>> meine_Zahlenliste = list(range(0, 500, 50))
>>> print(meine_Zahlenliste)
[0, 50, 100, 150, 200, 250, 300, 350, 400, 450]
>>> print(sum(meine_Zahlenliste))
2250
```

In der ersten Zeile haben wir mit der Funktion `range` eine Liste von Zahlen zwischen 0 und 500 bei einer Schrittweite von 50 erzeugt. Danach haben wir uns die Liste mit `print` anzeigen lassen, um uns das Ergebnis anzuschauen. Zum Schluss haben wir die Variable `meine_Zahlenliste` mit `print(sum(meine_Zahlenliste))` an die Funktion `sum` weitergereicht, wodurch alle Elemente der Liste zusammengezählt wurden und die Summe von 2250 dabei herauskam.

10.2 Umgang mit Dateien

Python-Dateien sind die gleichen, die Du von Deinem Computer kennst: Dokumente, Bilder, Musik, Spiele … im Prinzip ist alles auf Deinem Computer in Form von Dateien gespeichert.

Jetzt schauen wir uns an, wie wir in Python Dateien öffnen und mit ihnen arbeiten, indem wir die eingebaute Funktion `open` benutzen. Aber zunächst müssen wir eine neue Datei erzeugen, um mit ihr etwas herumspielen zu können.

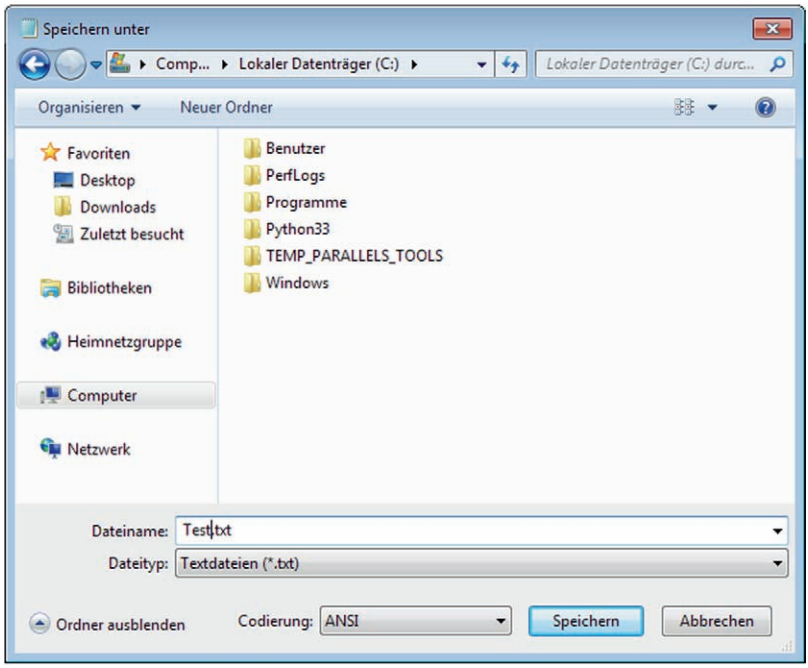
Erzeugen einer Test-Datei

Wir werden mit einer Textdatei experimentieren, die wir *Test.txt* nennen. Folge den Anleitungsschritten für das Betriebssystem, das Du benutzt.

Eine neue Datei unter Windows erzeugen

Wenn Du Windows benutzt, führe die folgenden Schritte aus, um die Datei *Test.txt* zu erzeugen:

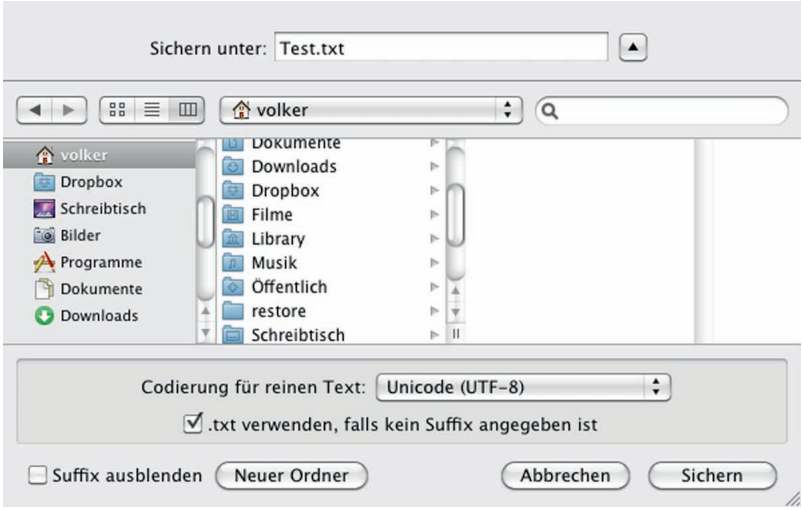
1. Gehe auf **Start** ▶ **Alle** ▶ **Programme** ▶ **Zubehör** ▶ **Editor**.
2. Gibt ein paar Zeilen in die leere Datei ein.
3. Gehe auf **Datei** ▶ **Speichern**.
4. Wenn das Dialogfenster erscheint, wählst Du das Laufwerk *C:* aus, indem Du auf **Computer** klickst und dann auf **Lokaler Datenträger (C:)** einen Doppelklick machst.
5. Gib unten im Fenster bei **Dateiname** *Test.txt* ein.
6. Zum Schluss klickst Du auf den Button **Speichern**.



Eine neue Datei unter MacOS X erzeugen

Wenn Du einen Mac benutzt, folge diesen Schritten, um die Datei *Test.txt* zu erzeugen:

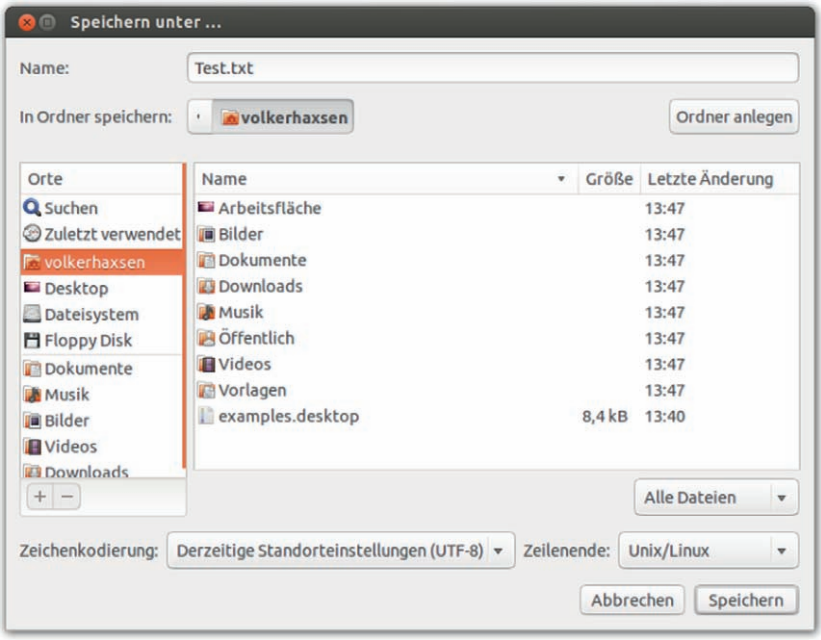
1. Klicke auf das Icon **Spotlight** in der Menüleiste ganz oben auf dem Bildschirm.
2. Ein Suchfenster erscheint. Gib hier **TextEdit** ein.
3. TextEdit sollte daraufhin in dem Bereich *Programme* erscheinen. Klicke darauf, um den Editor zu öffnen. (Du findest TextEdit auch im Ordner *Programme* im Finder.)
4. Gib ein paar Zeilen Text in die leere Datei ein.
5. Gehe auf **Format ▶ In reinen Text umwandeln**.
6. Gehe auf **Ablage ▶ Sichern**.
7. Im Dialogfenster **Sichern unter** gibst Du *Test.txt* ein.
8. In der Liste **Favoriten** klickst Du auf Deinen Benutzernamen – oder den Namen dessen, dem der Computer gehört.
9. Zum Schluss klickst Du auf den Button **Sichern**.



Eine neue Datei unter Ubuntu erzeugen

Wenn Du Ubuntu benutzt, folge diesen Schritten, um die Datei *Test.txt* zu erzeugen:

1. Öffne Deinen Editor, der normalerweise *Textverarbeitung* oder *gedit* heißt. Falls Du ihn zuvor noch nicht benutzt hast, kannst Du nach ihm im Menü **Anwendungen** suchen.
2. Gib ein paar Zeilen Text in den Editor ein.
3. Klicke auf den Button **Speichern**.
4. Gib in der Eingabezeile **Name** *Test.txt* als Dateinamen an. Unter **In Ordner speichern** ist vielleicht schon Dein persönlicher Ordner ausgewählt. Falls nicht, klickst Du links auf ihn in der Liste **Orte**. (Dein persönlicher Ordner ist mit dem Benutzernamen beschriftet, mit dem Du eingeloggt bist.)
5. Klicke auf den Button **Speichern**.



Eine Datei in Python öffnen

Pythons eingebaute Funktion `open` öffnet eine Datei in der Python-Shell und zeigt deren Inhalt an. Wie Du der Funktion sagst, welche Datei sie öffnen soll, hängt von Deinem Betriebssystem ab. Sieh Dir das Beispiel für eine Windows-Datei an, oder lies die Mac- oder Ubuntu-spezifischen Abschnitte, falls Du eines dieser Betriebssysteme benutzt.

Eine Windows-Datei öffnen

Falls Du Windows benutzt, gib folgenden Code ein, um *Test.txt* zu öffnen:

```
>>> Testdatei = open('c:\\Test.txt')
>>> Text = Testdatei.read()
>>> print(Text)
Ein furchtsames Fräulein aus Kassel
sah plötzlich im Raum eine Assel.
Da fiel es vor Schreck
sogleich in den Dreck.
Da haben wir nun den Schlamassel.
```

In der ersten Zeile benutzen wir `open`, was ein Datei-Objekt zurückgibt, das Funktionen zum Arbeiten mit Dateien enthält. Der Parameter, den wir in der Funktion `open` benutzen, ist ein String, der Python sagt, wo es die Datei findet. Falls Du Windows benutzt, hast Du *Test.txt* auf der lokalen Festplatte *C:* gespeichert, sodass Du den Speicherort Deiner Datei mit `c:\\Text.txt` angibst.

Die beiden Rückwärtsschrägstriche im Windows-Dateinamen teilen Python mit, dass die Rückwärtsschrägstriche nur für sich stehen und keine Befehle darstellen. (Wie Du in Kapitel 4 gelernt hast, haben Rückwärtsschrägstriche eine besondere Bedeutung in Python, vor allem in Strings.) Wir speichern das Datei-Objekt in der Variablen `Testdatei`.

In der zweiten Zeile benutzen wir die Funktion `read`, die von dem Datei-Objekt zur Verfügung gestellt wird, um die Inhalte der Datei auszulesen und sie in der Variable `Text` zu speichern. Wir geben die Variable in der letzten Zeile aus, um die Inhalte der Datei anzuzeigen.

Eine MacOSX-Datei öffnen

Falls Du MacOSX benutzt, musst du in der ersten Zeile des Windows-Beispiels einen anderen Pfad angeben, um *Test.txt* zu öffnen. Verwende dabei den Benutzernamen, auf den Du beim Abspeichern der Textdatei geklickt hast, und füge ihn in den String ein. Falls Dein Benutzername zum Beispiel *susannesommer* ist, sollte der Parameter für `open` so aussehen:

```
>>> Testdatei = open('/Users/susannesommer/Test.txt')
```

Eine Ubuntu-Datei öffnen

Falls Du Ubuntu benutzt, musst Du in der ersten Zeile des Windows-Beispiels einen anderen Pfad angeben, um *Test.txt* zu öffnen. Verwende dabei den Benutzernamen, auf den Du beim Abspeichern der Textdatei geklickt hast. Falls Dein Benutzername zum Beispiel *maximilian* ist, sollte der Parameter für `open` so aussehen:

```
>>> Testdatei = open('/home/maximilian/Test.txt')
```

In Dateien schreiben

Das Datei-Objekt, das `open` zurückgegeben hat, hat neben der Funktion `read` noch weitere Funktionen. Durch einen zweiten Parameter, den String `'w'`, können wir eine neue leere Datei erzeugen, wenn wir die Funktion aufrufen:

```
>>> Testdatei = open('c:\\MeineDatei.txt', 'w')
```

Der Parameter `'w'` teilt Python mit, dass wir in das Datei-Objekt schreiben wollen, statt aus ihm zu lesen.

Wir können nun mit der Funktion `write` dieser neuen Datei Informationen hinzufügen:

```
>>> Testdatei = open('c:\\MeineDatei.txt', 'w')
>>> Testdatei.write('Dies ist meine Testdatei')
```

Zum Schluss müssen wir mit der Funktion `close` Python sagen, wann wir mit dem Schreiben in die Datei fertig sind:

```
>>> Testdatei = open('c:\\MeineDatei.txt', 'w')
>>> Testdatei.write('Was ist grün und fährt hüpfend rauf und runter?
                    Eine Erbse im Fahrstuhl!')
>>> Testdatei.close()
```

Wenn Du jetzt diese Datei mit Deinem Editor öffnest, solltest Du sehen, dass sie den Text `'Was ist grün und fährt hüpfend rauf und runter? Eine Erbse im Fahrstuhl!'` enthält. Oder Du kannst Python bitten, ihn Dir auszugeben:

```
>>> Testdatei = open('
c:\\MeineDatei.txt')
>>> print(Testdatei.read())
Was ist grün und fährt hüpfend rauf und runter?
Eine Erbse im Fahrstuhl!
```

10.3 Was Du gelernt hast

In diesem Kapitel hast Du etwas über die eingebauten Funktionen von Python gelernt, wie etwa `float` und `int`, mit denen man Dezimalzahlen in ganze Zahlen verwandeln kann und umgekehrt. Du hast auch gesehen, wie man mit der Funktion `len` Schleifen einfacher gestalten kann und wie man mit Python Dateien öffnen kann, um aus ihnen zu lesen und in sie zu schreiben.

10.4 Programmier-Puzzles

Versuche Dich an den folgenden Beispielen, um mit einigen der in Python eingebauten Funktionen zu experimentieren. Die Lösungen findest Du unter *www.dpunkt.de/python*.

#1: Geheimnisvoller Code

Was kommt dabei heraus, wenn man folgenden Code ausführen lässt? Rate zuerst selbst, und lass den Code erst dann durchlaufen, um zu schauen, ob Du recht hast.

```
>>> a = abs(10) + abs(-10)
>>> print(a)
>>> b = abs(-10) + -10
>>> print(b)
```

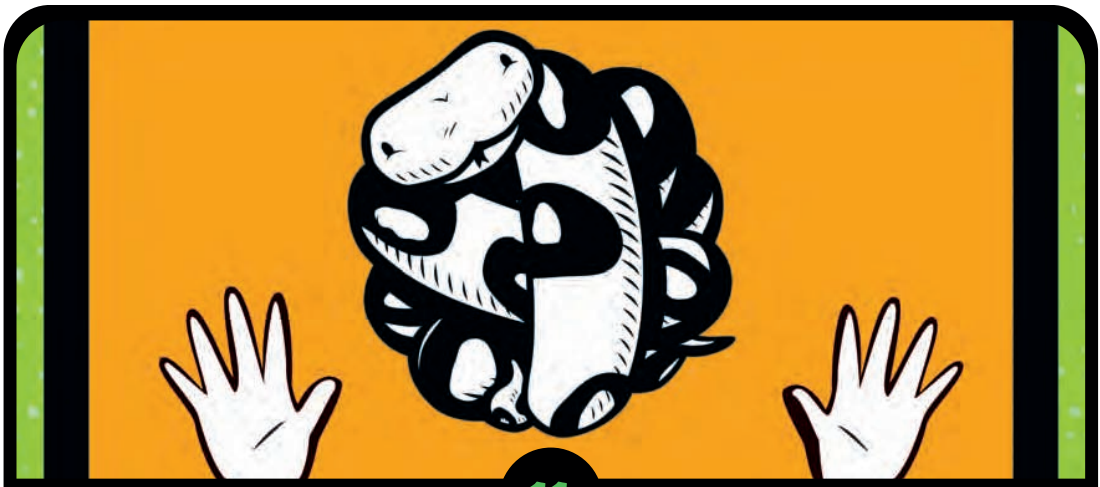
#2: Eine versteckte Botschaft

Versuche mit den Funktionen `dir` und `help` herauszufinden, wie man einen String in einzelne Wörter aufteilt, und schreibe dann ein kleines Programm, um jedes zweite Wort in folgendem String anzeigen zu lassen. Beginne dabei mit dem ersten Wort (Dies):

```
"Dies falls ist Du kein bist guter lesen Weg dann um hat eine es
Nachricht falsch zu Inhalt verstecken"
```

#3: Eine Datei kopieren

Schreibe ein Python-Programm, um eine Datei zu kopieren. (Tipp: Die Datei, die Du kopieren möchtest, musst Du erst öffnen. Dann musst Du sie einlesen und anschließend eine neue Datei erzeugen – die Kopie.) Prüfe, ob Dein Programm funktioniert, indem Du den Inhalt der neuen Datei auf dem Monitor ausgibst.



11
Nützliche Python-Module

Wie Du in Kapitel 8 gelernt hast, besteht ein Python-Modul aus jeder erdenklichen Kombination von Funktionen, Klassen und Variablen. Python setzt Module ein, um Funktionen und Klassen zu gruppieren, damit sie leichter zu benutzen sind. Das Modul `turtle` zum Beispiel, das wir in den vorigen Kapiteln benutzt haben, gruppiert Funktionen und Klassen, mit denen man eine Leinwand für eine Schildkröte erzeugen kann, um auf dem Monitor zu zeichnen.

Wenn Du ein Modul in ein Programm importierst, kannst Du dessen gesamten Inhalt nutzen. Als wir zum Beispiel in Kapitel 5 das Modul `turtle` importiert haben, hatten wir Zugriff auf die Klasse `Pen`, mit der wir ein Objekt auf der Leinwand gezeichnet haben, das für die Schildkröte stand:

```
>>> import turtle
>>> t = turtle.Pen()
```

Python enthält jede Menge Module, um die verschiedensten Aufgaben zu erledigen. In diesem Kapitel schauen wir uns einige der nützlichsten Module an und probieren einige ihrer Funktionen aus.

11.1 Mit dem Modul copy Kopien erstellen

Das Modul `copy` enthält Funktionen, mit denen man Kopien von Objekten erzeugt. Normalerweise erzeugt man beim Schreiben eines Programms neue Objekte. Manchmal kann es aber nützlich sein, eine Kopie eines Objekts zu erzeugen und diese dann zu benutzen, um ein neues Objekt zu erzeugen. Das macht man vor allen Dingen immer dann, wenn der Prozess der Erzeugung eines Objekts mehrere Schritte erfordert.

Stell Dir zum Beispiel einmal vor, wir hätten eine Klasse `Tier` mit einer Funktion `__init__`, die die Parameter `Art`, `Anzahl_der_Beine` und `Farbe` aufnimmt.

```
>>> class Tier:
    def __init__(self, Art, Anzahl_der_Beine, Farbe):
        self.Art = Art
        self.Anzahl_der_Beine = Anzahl_der_Beine
        self.Farbe = Farbe
```

Wir könnten in der Klasse `Tier` ein neues Objekt erzeugen, indem wir folgenden Code einsetzen. Lass uns ein pinkfarbenes Hippogreif mit sechs Beinen erzeugen, das `Harry` heißt.

```
>>> Harry = Tier('Hippogreif', 6, 'pink')
```

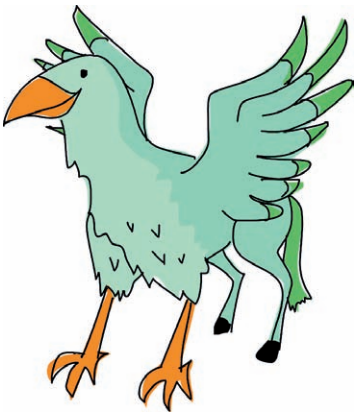
Nehmen wir an, wir wollten eine ganze Herde von pinkfarbenen Hippogreifs mit sechs Beinen haben. Wir könnten den Code von oben ständig wiederholen oder die Funktion `copy` aus dem Modul `copy` dazu benutzen:

```
>>> import copy
>>> Harry = Tier('Hippogreif', 6, 'pink')
>>> Harriet = copy.copy(Harry)
>>> print(Harry.Art)
Hippogreif
>>> print(Harriet.Art)
Hippogreif
```

In diesem Beispiel erzeugen wir ein Objekt und bezeichnen es mit der Variable `Harry`. Dann erstellen wir eine Kopie des Objekts, die wir mit `Harriet` bezeichnen. Es handelt sich dabei um komplett unterschiedliche Objekte, obwohl sie zur gleichen Art gehören. In diesem Fall spart man sich nur etwas Tippen, aber sobald die Objekte viel komplizierter werden, ist die Möglichkeit, kopieren zu können, äußerst nützlich.

Wir können auch eine Liste der Objekte `Tier` erzeugen und sie mit der Funktion `copy` kopieren.

```
>>> Harry = Tier('Hippogreif', 6, 'pink')
>>> Carrie = Tier('Chimäre', 4, 'grüne Punkte')
>>> Billy = Tier('Bogill', 0, 'gescheckt')
>>> meine_Tiere = [Harry, Carrie, Billy]
>>> mehr_Tiere = copy.copy(meine_Tiere)
>>> print(mehr_Tiere [0].Art)
Hippogreif
>>> print(mehr_Tiere [1].Art)
Chimäre
```



In den ersten drei Zeilen erzeugen wir drei Tier-Objekte und speichern sie in `Harry`, `Carrie` und `Billy`. In der vierten Zeile fügen wir diese Objekte der Liste `meine_Tiere` hinzu. Und danach benutzen wir `copy`, um eine neue Liste namens `mehr_Tiere` zu erzeugen. Zum Schluss zeigen wir die ersten beiden Objekte (`[0]` und `[1]`) in der Liste `mehr_Tiere` an und sehen nach, ob sie die gleichen wie in der Original-Liste sind: `Hippogreif` und `Chimäre`. Wir haben dadurch eine Kopie der Liste erstellt, ohne die ganzen Objekte erneut erzeugen zu müssen.

Schau Dir aber an, was passiert, sobald wir die Art eines unserer Objekte in `Tiere` in der Original-Liste `meine_Tiere` ändern (`Hippogreif` zu `Ghul`). Python ändert dann auch die Art in `mehr_Tiere`.

```
>>> meine_Tiere[0].Art = 'Ghul'
>>> print(meine_Tiere[0].Art)
Ghul
>>> print(mehr_Tiere[0].Art)
Ghul
```

Das ist ja merkwürdig. Haben wir die Art nicht gerade nur in `meine_Tiere` geändert? Warum wurde die Art in beiden Listen geändert?

Die Art wurde geändert, weil `copy` in Wirklichkeit eine *flache Kopie* angefertigt. Dies bedeutet, dass keine Objekte innerhalb der Objekte, die wir kopiert haben, kopiert werden. In diesem Fall hat Python das Hauptobjekt `Liste` kopiert, nicht aber die einzelnen Objekte innerhalb der Liste. Somit haben wir am Ende zwar eine neue Liste, die aber keine neuen Objekte enthält – die Liste `mehr_Tiere` enthält die gleichen drei Objekte wie `meine_Tiere`.

Wenn wir der ersten Liste (meine_Tiere) ein neues Tier hinzufügen, erscheint es aus dem gleichen Grund nicht in der Kopie (mehr_Tiere). Zum Beweis lässt Du Dir die Länge jeder Liste nach Hinzufügen jedes Tieres anzeigen:

```
>>> Sally = Tier('Sphinx', 4, 'Sand')
>>> meine_Tiere.append(Sally)
>>> print(len(meine_Tiere))
4
>>> print(len(mehr_Tiere))
3
```

Wie Du siehst, haben wir der ersten Liste, meine_Tiere, zwar ein neues Tier hinzugefügt, es wird aber nicht zu der Kopie dieser Liste (mehr_Tiere) hinzugefügt. Wenn wir die Funktion len benutzen und das Ergebnis anzeigen lassen, enthält die erste Liste vier Elemente und die zweite nur drei.

Eine weitere Funktion des Moduls copy, deepcopy (engl. für »tief kopieren«), erzeugt tatsächlich Kopien sämtlicher Objekte innerhalb des Objekts, das kopiert wird. Wenn wir zum Kopieren von meine_Tiere die Funktion deepcopy verwenden, bekommen wir eine neue vollständige Liste mit Kopien aller ihrer Objekte. Die Folge davon ist: Änderungen an unseren Tier-Originalobjekten haben keinen Einfluss auf die Objekte in der neuen Liste. Hier siehst Du ein Beispiel:

```
>>> mehr_Tiere = copy.deepcopy(meine_Tiere)
>>> meine_Tiere[0].Art = 'Wyrm'
>>> print(meine_Tiere[0].Art)
Wyrm
>>> print(mehr_Tiere[0].Art)
Ghul
```

Wenn wir die Art des ersten Objekts in der Originalliste von Ghul in Wyrm ändern, ändert sich die kopierte Liste nicht, wie wir sehen, wenn wir uns die Arten des ersten Objekts der Liste anzeigen lassen.

11.2 Mit dem Modul keyword einen Überblick über die Schlüsselwörter erhalten

In Python ist jedes Wort, das Bestandteil der Sprache selbst ist, wie etwa if, else und for, ein Schlüsselwort. Das Modul keyword enthält eine Funktion namens iskeyword und eine Variable, die kwlist heißt. Die Funktion iskeyword gibt wahr (true) zurück, sobald ein String ein Python-Schlüsselwort ist. Die Variable kwlist gibt eine Liste sämtlicher Python-Schlüsselwörter zurück.

Im folgenden Code kannst Du sehen, dass die Funktion `iskeyword` `true` für den String `if` und `false` (falsch) für den String `Oswald` zurückgibt. Wenn wir die Inhalte der Variable anzeigen lassen, kannst du die komplette Liste aller Schlüsselwörter sehen. Dies ist sehr nützlich, da Schlüsselwörter nicht immer gleich bleiben. Zukünftige Versionen (oder ältere Versionen) von Python können abweichende Schlüsselwörter enthalten.

```
>>> import keyword
>>> print(keyword.iskeyword('if'))
True
>>> print(keyword.iskeyword('Oswald'))
False
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal',
'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with',
'yield']
```

Für jedes der Schlüsselwörter findest Du eine Beschreibung im Anhang.

11.3 Wie man mit dem Modul random Zufallszahlen bekommt

Das Modul `random` enthält eine Reihe von Funktionen, mit denen man Zufallszahlen erzeugen kann. Das ist so, als würde man dem Computer sagen: »Ziehe eine Zahl.« Die nützlichsten Funktionen im Modul `random` sind `randint`, `choice` und `shuffle`.

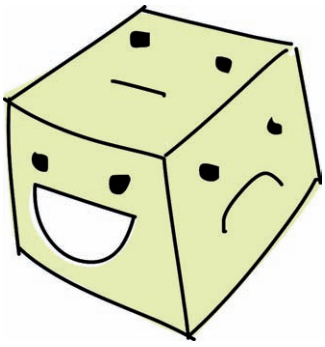
Mit randint eine Zufallszahl bestimmen lassen

Die Funktion `randint` zieht eine Zufallszahl innerhalb eines Zahlenbereichs – sagen wir, zwischen 1 und 100, zwischen 100 und 1000 oder zwischen 1000 und 5000. Hier siehst Du ein Beispiel:

```
>>> import random
>>> print(random.randint(1, 100))
58
>>> print(random.randint(100, 1000))
861
>>> print(random.randint(1000, 5000))
3795
```

Du kannst `randit` auch zu so etwas wie einem kleinen (und nervigen) Ratespiel nutzen, indem Du eine `while-Schleife` einsetzt:

```
>>> import random
>>> num = random.randint(1, 100)
❶ >>> while True:
❷         print('Rate eine Zahl zwischen 1 und 100')
❸         raten = input()
❹         i = int(raten)
❺         if i == num:
                print('Du hast richtig geraten')
❻               break
❼         elif i < num:
                print('Rate eine höhere Zahl')
❽         elif i > num:
                print('Rate eine niedrigere Zahl')
```



Als Erstes importieren wir das Modul `random` und weisen der Variablen `num` mit `randint` eine Zufallszahl zwischen 1 und 100 zu. Wir erzeugen dann eine `while`-Schleife in ❶, die unendlich läuft (oder zumindest so lange, bis der Spieler die Zahl errät).

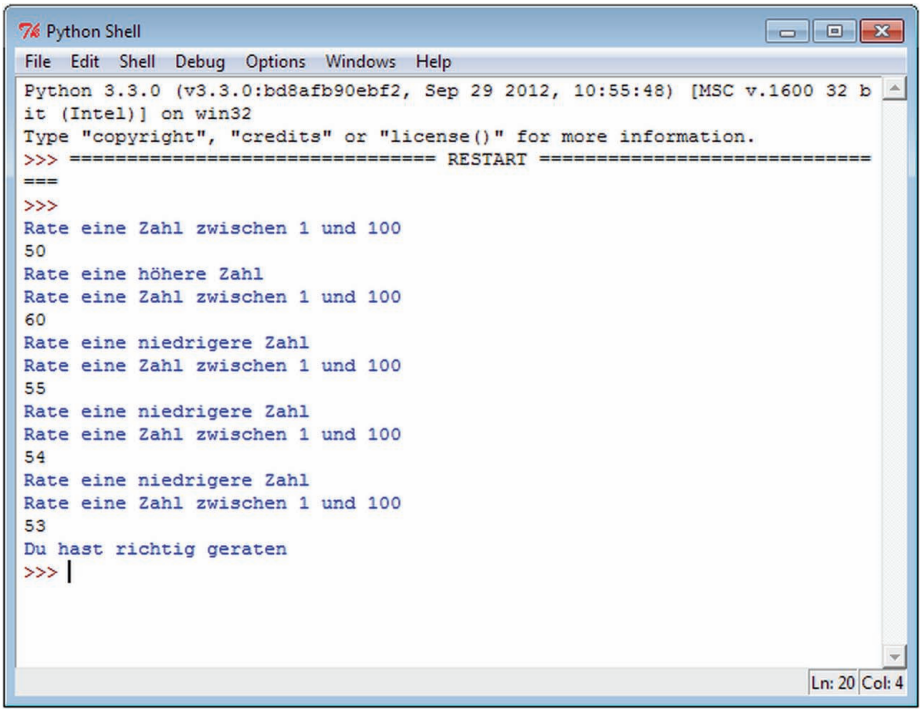
Als Nächstes zeigen wir eine Nachricht in ❷ an und benutzen `input`, um eine Benutzereingabe zu bekommen, die wir in der Variable `raten` speichern ❸. Wir wandeln die Eingabe mit `int` in eine Zahl um und speichern sie in ❹ in der Variablen `i`. Anschließend vergleichen wir sie mit der in ❺ zufällig ausgewählten Zahl.

Falls die Eingabe und zufällig ausgewählte Zahl gleich sind, zeigen wir die Nachricht »Du hast richtig geraten« an und verlassen die Schleife in ❻. Falls die Zahlen nicht übereinstimmen, prüfen wir, ob die Zahl, die der Spieler geraten hat, höher ist als die Zufallszahl ❼ oder niedriger ❽, und zeigen ihm eine Nachricht mit einem entsprechenden Hinweis.

Dieser Code ist ein bisschen lang, sodass Du ihn vielleicht besser in ein neues Shell-Fenster schreibst oder ein Text-Dokument erzeugst, es speicherst und anschließend in IDLE laufen lässt. Hier siehst Du eine kurze Erinnerung daran, wie man ein abgespeichertes Programm öffnet und ausführt:

1. Starte IDLE, und gehe auf **File ▶ Open**.
2. Arbeite Dich bis zu dem Verzeichnis durch, in dem Du die Datei gespeichert hast, und klicke auf den Dateinamen, um sie auszuwählen.
3. Klicke auf **Open.**
4. Nachdem sich das neue Fenster geöffnet hat, gehe auf **Run ▶ Run Module**.

Hier sieht man, was passiert, wenn wir das Programm laufen lassen:



Mit choice ein zufälliges Element aus einer Liste auswählen

Wenn Du statt einer Zufallszahl aus einem bestimmten Bereich ein zufälliges Element einer Liste auswählen lassen möchtest, kannst Du `choice` benutzen. Du kannst Dir zum Beispiel von Python den Nachtisch auswählen lassen.

```
>>> import random
>>> Nachtische = ['Eis', 'Götterspeise', 'Pudding', 'Lebkuchen',
                  'Schokolade']
>>> print(random.choice(Nachtische))
Pudding
```

Es sieht so aus, als gäbe es heute Pudding – keine schlechte Wahl.

Mit shuffle eine Liste mischen

Die Funktion `shuffle` mischt die Elemente einer Liste. Wenn Du gerade mit IDLE arbeitest und das Modul `random` schon importiert und die Nachtisch-Liste aus dem vorigen Beispiel schon erstellt hast, kannst Du gleich zum Befehl `random.shuffle` vorgehen und folgenden Code eingeben:

```
>>> import random
>>> Nachtische = ['Eis', 'Götterspeise', 'Pudding', 'Lebkuchen',
                  'Schokolade']
>>> random.shuffle(Nachtische)
>>> print(Nachtische)
['Lebkuchen', 'Schokolade', 'Pudding', 'Götterspeise', 'Eis']
```

Den Effekt des Durchmischens kannst Du sehen, wenn wir die Liste anzeigen – die Reihenfolge ist jetzt komplett anders. Wenn wir ein Kartenspiel schreiben würden, könntest Du diese Funktion benutzen, um eine Liste, die die Spielkarten beinhaltet, zu mischen.

11.4 Die Shell mit dem Modul sys steuern

Das Modul sys enthält Systemfunktionen, mit denen man die Python-Shell selbst steuern kann. Hier schauen wir uns an, wie man die Funktion `exit` benutzt, was man mit den Objekten `stidin` und `stdout` macht, und wir sehen uns die Variable `version` an.

Die Shell mit der Funktion exit verlassen

Die Funktion `exit` ist eine der Möglichkeiten, um die Python-Shell oder -Konsole zu beenden. Gib den folgenden Code ein, und Du wirst in einem Dialogfenster gefragt, ob Du die Shell beenden möchtest. Klicke auf **Yes**, und die Shell schließt sich.

```
>>> import sys
>>> sys.exit()
```

Dies funktioniert jedoch nicht, falls Du eine modifizierte Version von IDLE benutzt, die wir in Kapitel 2 aufgesetzt haben. Stattdessen bekommst Du eine Fehlermeldung wie diese:

```
>>> import sys
>>> sys.exit()
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    sys.exit()
SystemExit
```

In dem Objekt stdin lesen

Das Objekt `stdin` (Abkürzung für *standard input*, engl. für »Standard-Eingabe«) im Modul sys fordert den Benutzer auf, eine Information einzugeben, die in die Shell eingelesen und vom Programm verwendet wird. Wie Du schon in Kapitel 8

gelernt hast, enthält dieses Objekt eine Funktion `readline`, die eine Zeile Text einliest, der auf der Tastatur eingegeben wurde, bis der Benutzer die Enter-Taste drückt. Sie funktioniert wie die Funktion `input`, die wir im Zufallszahlen-Ratespiel weiter oben in diesem Kapitel benutzt haben. Gib zum Beispiel das Folgende ein:

```
>>> import sys
>>> v = sys.stdin.readline()
Wer zuletzt lacht, denkt am langsamsten
```

Python speichert nun den String `Wer zuletzt lacht, denkt am langsamsten` in die Variable `v`. Um uns dies bestätigen zu lassen, geben wir die Inhalte von `v` aus:

```
>>> print(v)
Wer zuletzt lacht, denkt am langsamsten
```

Einer der Unterschiede zwischen den Funktionen `input` und `readline` besteht darin, dass man mit der Funktion `readline` die Anzahl der Zeichen festlegen kann, die als Parameter gelesen werden. Zum Beispiel:

```
>>> v = sys.stdin.readline(17)
Wer zuletzt lacht, denkt am langsamsten
>>> print(v)
Wer zuletzt lacht
```

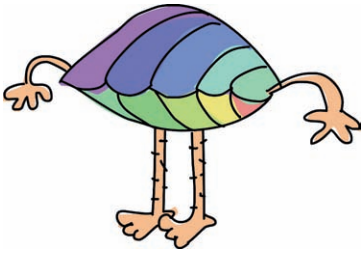
Mit dem Objekt stdout schreiben

Im Gegensatz zu `stdin` wird das Objekt `stdout` (Abkürzung für *standard output*, engl. für »Standard-Ausgabe«) verwendet, um Mitteilungen in die Shell (oder Konsole) zu schreiben, anstatt sie einzulesen. In gewisser Hinsicht ist es das Gleiche wie `print`; `stdout` ist aber ein Datei-Objekt. Es hat also die gleichen Funktionen, die wir in Kapitel 10 verwendet haben, wie etwa `write`. Hier siehst Du ein Beispiel:

```
>>> import sys
>>> sys.stdout.write("Welche drei Worte machen einen Hai glücklich?
                     Mann über Bord!")
Welche drei Worte machen einen Hai glücklich? Mann über Bord!61
```

Wie Du siehst, gibt `write` die Anzahl der Zeichen zurück, die es geschrieben hat: Am Ende der Mitteilung siehst Du die Zahl 61. Wir könnten diesen Wert in einer Variable speichern, um im Verlauf festzuhalten, wie viele Zeichen auf den Monitor geschrieben wurden.

Welche Python-Version benutze ich?



Die Variable `version` zeigt die Version von Python an. Dies kann ganz nützlich sein, um sicherzustellen, dass man auf dem aktuellen Stand ist. Manche Programmierer lassen gerne Informationen anzeigen, wenn ihre Programme gestartet werden. Du könntest zum Beispiel in einem »Über«-Fenster die Version von Python anzeigen lassen:

```
>>> import sys
>>> print(sys.version)
3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit
(Intel)]
```

11.5 Mit dem Modul time arbeiten

Das Modul `time` von Python enthält Funktionen zum Anzeigen der Zeit, allerdings nicht unbedingt so, wie Du es erwartest. Probiere einmal Folgendes aus:

```
>>> import time
>>> print(time.time())
1359450712.489086
```

Die Zahl, die nach dem Aufruf von `time()` zurückgegeben wird, ist die Anzahl von Sekunden, die seit dem 1. Januar 1970 um 0:00 Uhr vergangen sind. Dieser Referenzpunkt mag sehr ungewöhnlich erscheinen, hat aber seinen Sinn. Um zum Beispiel herauszufinden, wie lange Teile Deines Programms brauchen, um abzulaufen, kannst Du die Zeit am Anfang und am Ende aufzeichnen und diese Werte anschließend vergleichen. Probieren wir einmal aus, wie lange es dauert, alle Zahlen zwischen 0 und 999 anzuzeigen. Zunächst erzeugen wir eine Funktion wie diese:

```
>>> def ganz_viele_Zahlen(max):
        for x in range(0, max):
                print(x)
```

Als Nächstes rufen wir die Funktion auf, wobei `max` auf 1000 gesetzt ist:

```
>>> ganz_viele_Zahlen(1000)
```

Danach finden wir heraus, wie lange die Funktion braucht, indem wir unser Programm mit dem Modul `time` abändern:

```
>>> def ganz_viele_Zahlen(max):
❶       t1 = time.time()
❷       for x in range(0, max):
            print(x)
❸       t2 = time.time()
❹       print('Es hat %s Sekunden gebraucht' % (t2-t1)
```

Wenn wir das Programm wieder aufrufen, bekommen wir folgendes Ergebnis (das von der Geschwindigkeit Deines Systems abhängt):

```
>>> ganz_viele_Zahlen(1000)
0
1
2
3
.
.
.
997
998
999
Es hat 50.159196853637695 Sekunden gebraucht.
```

So funktioniert das Ganze: Beim ersten Aufrufen der Funktion `time()` weisen wir den zurückgegebenen Werten der Variablen `t1` eins zu ❶. Anschließend gehen wir in die Schleife und geben alle Zahlen in der dritten und vierten Zeile aus ❷. Nach der Schleife rufen wir wieder die Funktion `time()` auf und weisen den zurückgegebenen Wert der Variablen `t2` zu ❸. Da es mehrere Sekunden zum Durchlaufen der Schleife dauert, ist der Wert `t2` größer als `t1`, da bei ihm seit dem 1. Januar 1970 mehr Sekunden vergangen sind. Wenn man wie wir in ❹ `t1` von `t2` abzieht, bekommt man die Anzahl von Sekunden, die es gedauert hat, um alle Zeilen auszugeben.

Mit asctime ein Datum umwandeln

Die Funktion `asctime` nimmt ein Datum als Tupel auf und wandelt es in etwas um, das lesbarer ist. (Erinnere dich daran, dass ein Tupel eine Liste von Elementen ist, die man nicht verändern kann.) Wie Du in Kapitel 8 gesehen hast, zeigt `asctime` das aktuelle Datum und die Uhrzeit in einer lesbaren Form an, wenn man asctime ohne Parameter aufruft.

```
>>> import time
>>> print(time.asctime())
Tue Jan 29 10:53:18 2013
```

Um `asctime` mit einem Parameter aufzurufen, erzeugen wir zunächst ein Tupel mit Werten für das Datum und die Uhrzeit. Hier zum Beispiel weisen wir das Tupel der Variablen `t` zu:

```
>>> t = (2007, 5, 27, 10, 30, 48, 6, 0, 0)
```

Die Werte in dieser Sequenz sind Jahr, Monat, Tag, Stunden, Minuten, Sekunden, Wochentag (0 steht für Montag, 1 ist Dienstag usw.) und die Angabe, ob gerade Sommerzeit ist (0, wenn nicht; 1, wenn ja). Wenn wir `asctime` mit einem ähnlichen Tupel aufrufen, bekommen wir Folgendes:

```
>>> import time
>>> t = (2020, 2, 23, 10, 30, 48, 6, 0, 0)
>>> print(time.asctime(t))
Sun Feb 23 10:30:48 2020
```

Mit localtime Datum und Uhrzeit bekommen

Im Gegensatz zu `asctime` gibt die Funktion `localtime` das aktuelle Datum und die Uhrzeit als Objekt zurück. Die Werte haben dabei ungefähr die gleiche Reihenfolge wie bei `asctime`. Wenn Du das Objekt ausgibst, siehst Du den Namen der Klasse und jeden Wert als `tm_year`, `tm_mon` (für Monat), `tm_day` (für Tag des Monats), `tm_hour` und so weiter angezeigt.

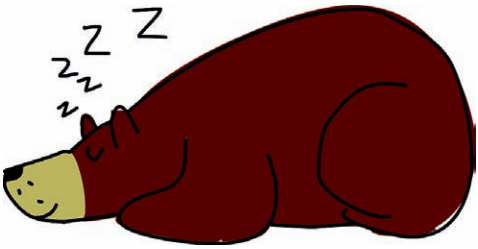
```
>>> import time
>>> print(time.localtime())
time.struct_time(tm_year=2013, tm_mon=1, tm_mday=29, tm_hour=11,
tm_min=24, tm_sec=47, tm_wday=1, tm_yday=29, tm_isdst=0)
```

Um das aktuelle Jahr und den Monat anzeigen zu lassen, kannst Du deren Index-Positionen verwenden (wie bei einem Tupel, das wir mit `asctime` benutzt haben). Anhand unseres Beispiels wissen wir, dass das Jahr an erster Stelle steht (Position 0) und der Monat (`month`) an der zweiten (1). Daher verwenden wir `Jahr = t[0]` und `Monat = t[1]` folgendermaßen:

```
>>> t = time.localtime()
>>> Jahr = t[0]
>>> Monat = t[1]
>>> print(Jahr)
2013
>>> print(Monat)
1
```

Wie Du siehst, haben wir den ersten Monat des Jahres 2013.

Mit sleep eine Pause machen



Die Funktion `sleep` ist ganz praktisch, wenn Du Dein Programm verzögern oder verlangsamen möchtest. Um zum Beispiel jede Sekunde von 1 bis 61 anzeigen zu lassen, können wir folgende Schleife verwenden:

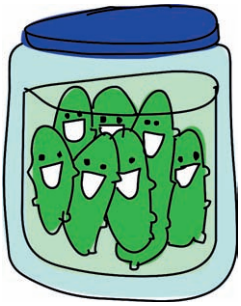
```
>>> for x in range(1, 61):
        print(x)
```

Dieser Code gibt ganz schnell alle Zahlen zwischen 1 und 60 aus. Wir können Python aber auch sagen, dass es zwischen jeder `print`-Anweisung eine Sekunde lang Pause machen soll:

```
>>> for x in range(1, 61):
        print(x)
        time.sleep(1)
```

Dadurch wird die Anzeige der nächsten Zahl verzögert. In Kapitel 13 werden wir die Funktion `sleep` dazu verwenden, eine Animation etwas realistischer erscheinen zu lassen.

11.6 Mit dem Modul pickle Informationen speichern



Mit dem Modul `pickle` wandelt man Python-Objekte in etwas um, das man in eine Datei schreiben und auch leicht wieder auslesen kann. Das Modul `pickle` ist dann ganz praktisch, wenn Du ein Spiel schreibst, bei dem Du Informationen über den Spielestand speichern möchtest. Hier siehst Du zum Beispiel, wie man Gegenstände bei einem Spiel hinzufügt und speichert:

```
>>> Spieldaten = {
    'Spielerposition' : 'N23 E45',
    'Hosentaschen': ['Schlüssel', 'Taschenmesser', 'polierter Stein'],
    'Rucksack' : ['Seil', 'Hammer', 'Apfel'],
    'Geld' : 158.50
}
```

Hier erzeugen wir eine Python-Map, die in unserem imaginären Spiel die Spielposition und eine Liste von Elementen in den Taschen und dem Rucksack des Spielers enthält sowie die Summe an Geld, die er bei sich trägt. Wir können diese Map

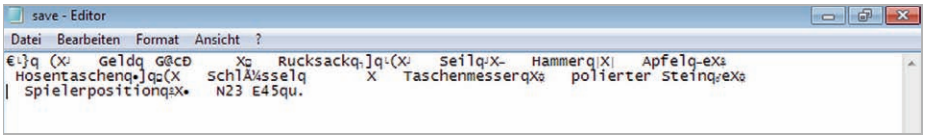
in eine Datei speichern, indem wir die Datei zum Schreiben öffnen und dann die Funktion `dump` von `pickle` aufrufen:

```
❶ >>> import pickle
❷ >>> Spieldaten = {
      'Spielerposition' : 'N23 E45',
      'Hosentaschen': ['Schlüssel', 'Taschenmesser', 'polierter Stein'],
      'Rucksack' : ['Seil', 'Hammer', 'Apfel'],
      'Geld' : 158.50
      }
❸ >>> speichere_Datei = open('save.dat', 'wb')
❹ >>> pickle.dump(Spieldaten, speichere_Datei)
❺ >>> speichere_Datei.close()
```

Als Erstes importieren wir in ❶ das Modul `pickle` und erzeugen in ❷ eine Map mit unseren Spielkarten. In ❸ öffnen wir die Datei *save.dat* mit dem Parameter `wb`, der Python sagt, dass es die Datei im Binärmodus schreiben soll (dazu musst Du vielleicht das richtige Verzeichnis wie */Users/martinoswald*, */home/susanneb/* oder *C:\\Users\JensIngrim* ausgeben, wie wir das in Kapitel 10 gemacht haben). In ❹ benutzen wir dann `dump`, um die Map und die Datei-Variable als zwei Parameter einzufügen. Zum Schluss schließen wir die Datei in ❺, da wir mit ihr fertig sind.

Achtung!

Einfache Textdateien enthalten nur Zeichen, die Menschen lesen können. Bilder, Musik-Dateien, Filme und `pickle`-Objekte in Python enthalten Informationen, die nicht immer für Menschen lesbar sind. Das sind die sogenannten Binärdateien. Wenn Du die Datei *save.dat* öffnen würdest, würdest Du sehen, dass sie nicht wie eine Textdatei aussieht, sondern wie eine wilde Mixtur aus normalem Text und Sonderzeichen.



Die mit `pickle` behandelten Objekte, die wir mit der Funktion `dump` von `pickle` in eine Datei geschrieben haben, können wir mit `load` rückgängig machen. Dabei kehren wir den Prozess von `pickle` um: Wir nehmen die Informationen, die in die Datei geschrieben wurden, auf und wandeln sie zurück in die Werte um, die unser Programm verwerten kann. Dieser Prozess funktioniert ähnlich wie bei der Funktion `dump`:

```
>>> lade_Datei = open('save.dat', 'rb')
>>> geladene_Spieldaten = pickle.load(lade_Datei)
>>> lade_Datei.close()
```

Als Erstes öffnen wir die Datei und benutzen `rb` als Parameter, was für »lies Binärdaten« (engl. *read binary*) steht. Wir reichen die Datei dann an `load` weiter und setzen den zurückgegebenen Wert in die Variable `geladene_Spieldaten`. Zum Schluss schließen wir die Datei wieder.

Um zu prüfen, dass die gespeicherten Daten korrekt geladen wurden, lassen wir die Variable ausgeben:

```
>>> print(geladene_Spieldaten)
{'Spielerposition': 'N23 E45', 'Hosentaschen': ['Schlüssel',
'Taschenmesser', 'polierter Stein'], 'Rucksack': ['Seil', 'Hammer',
'Apfel'], 'Geld': 158.5}
```

11.7 Was Du gelernt hast

In diesem Kapitel hast Du gelernt, wie Python-Module Funktionen, Klassen und Variablen zusammenfassen und wie man diese Funktionen durch Importieren der Module benutzt. Du hast gesehen, wie man Objekte kopiert, Zufallszahlen erzeugt und zufallsmäßig Listen von Objekten mischt. Außerdem hast Du gelernt, wie man in Python mit der Zeit arbeitet. Zu guter Letzt hast Du gelernt, wie man mit `pickle` Informationen in einer Datei speichert und wieder daraus lädt.

11.8 Programmier-Puzzles

Probier die folgenden Sachen aus, um den Umgang mit Pythons Modulen zu üben. Überprüfe Deine Antworten unter *www.dpunkt.de/python*.

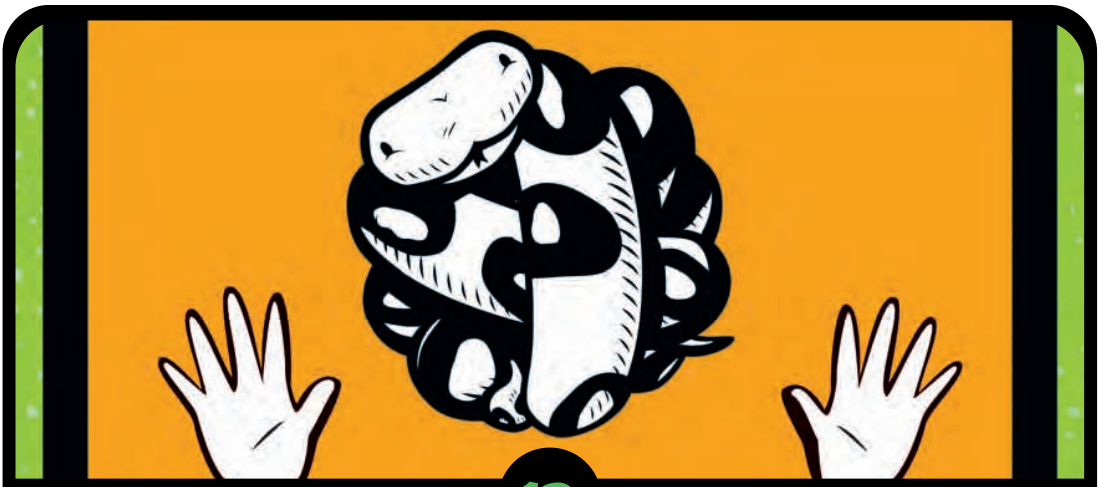
#1: Kopierte Autos

Was wird der folgende Code anzeigen?

```
>>> import copy
>>> class Auto:
        pass

>>> Auto1 = Auto()
>>> Auto1.Räder = 4
>>> Auto2 = Auto1
>>> Auto2.Räder = 3
>>> print(Auto1.Räder)

>>> Auto3 = copy.copy(Auto1)
>>> Auto3.Räder = 6
>>> print(Auto1.Räder)
```

#2: Favoriten in pickle

Erstelle eine Liste Deiner Lieblingsgegenstände, und benutze `pickle`, um sie in einer Datei namens *Favoriten.dat* zu speichern. Schließe dann die Python-Shell, öffne sie wieder, und lass die Liste Deiner Lieblingsgegenstände anzeigen, indem Du die Datei lädst.



12 Noch mehr Grafik mit turtle

Lass uns einen weiteren Blick auf das Modul `turtle` werfen, mit dem wir in Kapitel 5 begonnen haben. Wie Du in diesem Kapitel sehen wirst, kann Python mit seinen Schildkröten viel mehr anstellen, als nur einfache schwarze Linien zu zeichnen. Du kannst damit beispielsweise kompliziertere geometrische Formen zeichnen, unterschiedliche Farben erzeugen und Deine Formen sogar mit Farben füllen.

12.1 Fangen wir mit einem einfachen Quadrat an

Wie man mit der Schildkröte einfache Formen zeichnet, wissen wir schon. Bevor wir mit der Schildkröte zeichnen, müssen wir das Modul `turtle` importieren und das Objekt `Pen` erzeugen:

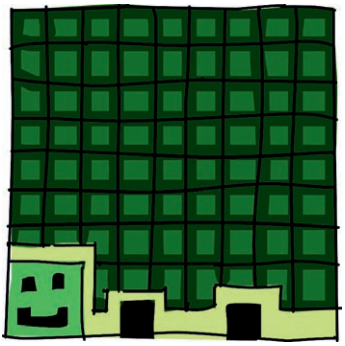
```
>>> import turtle
>>> t = turtle.Pen()
```

Hier siehst Du noch einmal der Code, mit dem wir in Kapitel 5 ein Quadrat erzeugt haben:

```
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
```

In Kapitel 7 hast Du etwas über `for`-Schleifen erfahren. Aufgrund unserer neuen Kenntnisse können wir diesen etwas umständlichen Code für ein Quadrat mit einer `for`-Schleife einfacher machen:

```
>>> t.reset()
>>> for x in range(1, 5):
        t.forward(50)
        t.left(90)
```



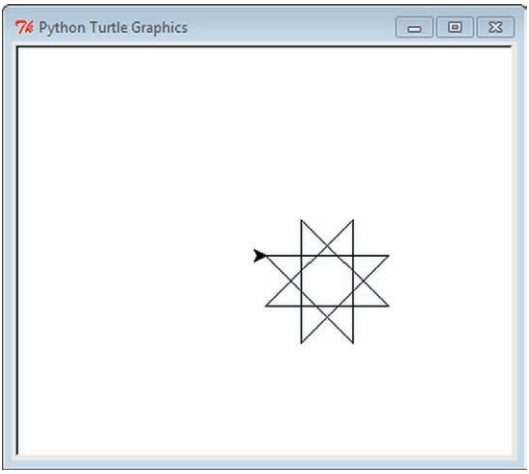
In der ersten Zeile sagen wir dem `Pen`-Objekt, dass es sich zurücksetzen soll. Als Nächstes starten wir eine `for`-Schleife, die mit dem Code `range(1, 5)` von 1 bis 4 zählt. Danach bewegen wir uns in den folgenden Zeilen bei jedem Durchlauf der Schleife 50 Pixel vorwärts und biegen 90° nach links ab. Weil wir die `for`-Schleife verwendet haben, ist dieser Code also ein bisschen kürzer als die vorherige Version – wenn man die `reset`-Zeile weglässt, haben wir statt sechs nur noch drei Zeilen.

12.2 Sterne zeichnen

Mit nur einigen kleinen Änderungen unserer `for`-Schleife können wir etwas Interessanteres zeichnen. Gibt Folgendes ein:

```
>>> t.reset()
>>> for x in range(1, 9):
        t.forward(100)
        t.left(225)
```

Dieser Code produziert einen achtzackigen Stern:



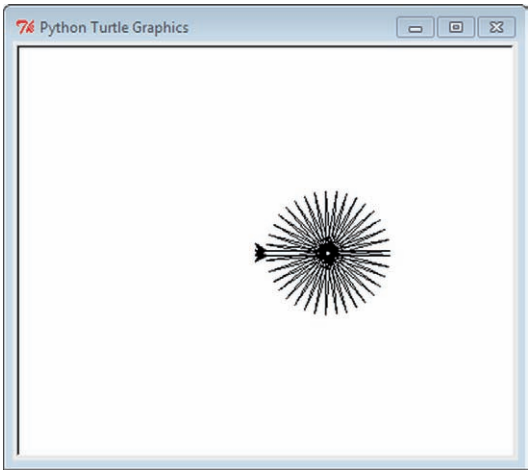
Der Code selbst ist dem, den wir beim Zeichnen des Quadrats verwendet haben, sehr ähnlich, bis auf ein paar Ausnahmen:

- Anstatt mit `range(1, 5)` die Schleife viermal zu durchlaufen, schleifen wir mit `range(1, 9)` achtmal hindurch.
- Anstatt uns um 50 Pixel vorwärts zu bewegen, nehmen wir 100 Pixel.
- Anstatt uns um 90° zu drehen, drehen wir uns um 225° nach links.

Jetzt entwickeln wir unseren Stern noch ein bisschen weiter. Indem wir einen 175°-Winkel verwenden und 37-mal durchschleifen, können wir einen Stern mit noch viel mehr Zacken malen:

```
>>> t.reset()
>>> for x in range(1, 38):
        t.forward(100)
        t.left(175)
```

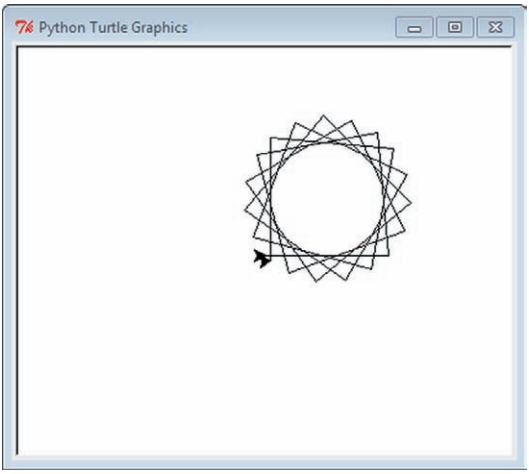
Hier ist das Ergebnis dieses Codes:



Wo wir gerade mit Sternen herumspielen: Hier ist der Code, um einen sich schraubenden Stern zu produzieren:

```
>>> t.reset()
>>> for x in range(1, 20):
        t.forward(100)
        t.left(95)
```

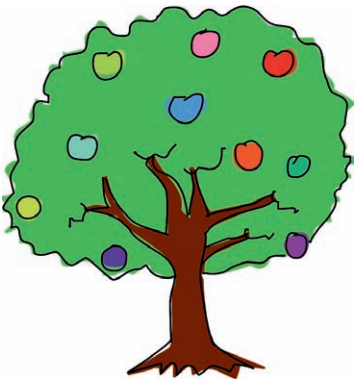
Indem wir den Drehwinkel ändern und die Anzahl der Schleifen reduzieren, zeichnet die Schildkröte einen ganz anderen Stern:



Mit sehr ähnlichem Code können wir eine ganze Reihe unterschiedlicher Formen erzeugen – von einem einfachen Quadrat bis zu einem sich schraubenden Stern. Wie Du siehst, haben wir es durch die Verwendung von `for`-Schleifen viel einfacher gemacht, diese Formen zu zeichnen. Ohne `for`-Schleifen hätte unser Code viel mehr mühsames Tippen erfordert.

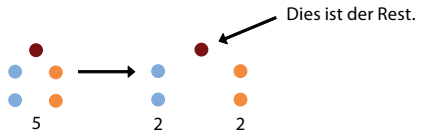
Lass uns jetzt mit einer `if`-Anweisung die Drehung der Schildkröte steuern und eine weitere Stern-Variante zeichnen. In diesem Beispiel möchten wir, dass die Schildkröte sich erst um einen Winkel und beim zweiten Mal um einen anderen Winkel dreht.

```
>>> t.reset()
>>> for x in range(1, 19):
        t.forward(100)
        if x % 2== 0:
                t.left(175)
        else:
                t.left(225)
```

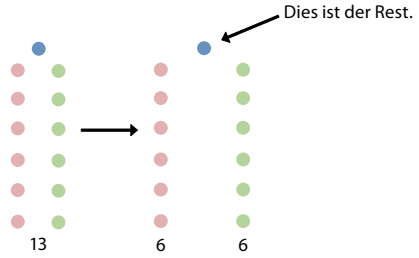


Hier erzeugen wir eine Schleife, die 18-mal durchlaufen wird (`range(1,19)`) und der Schildkröte sagt, dass sie sich 100 Pixel vorwärts bewegen soll (`t.forward(100)`). Das Neue hierbei ist die `if`-Anweisung (`if x % 2 ==:`). Diese Anweisung prüft, ob die Variable `x` eine gerade Zahl enthält, indem sie den sogenannten *Modulo*-Operator (Rest-Operator), also das `%` in dem Ausdruck `x % 2 == 0`, verwendet. Das ist, als ob man sagen würde, »x mod 2« ist gleich 0.

Der Ausdruck `x % 2` sagt im Grunde: »Wie groß ist der Rest, wenn man die Zahl in der Variable `x` in zwei gleiche Hälften teilt?« Wenn wir beispielsweise die Menge von fünf Bällen in zwei gleiche Teile teilen wollten, bekämen wir zwei Gruppen mit je zwei Bällen (einer Gesamtmenge von vier Bällen) und einen Rest von einem Ball, wie hier zu sehen ist:



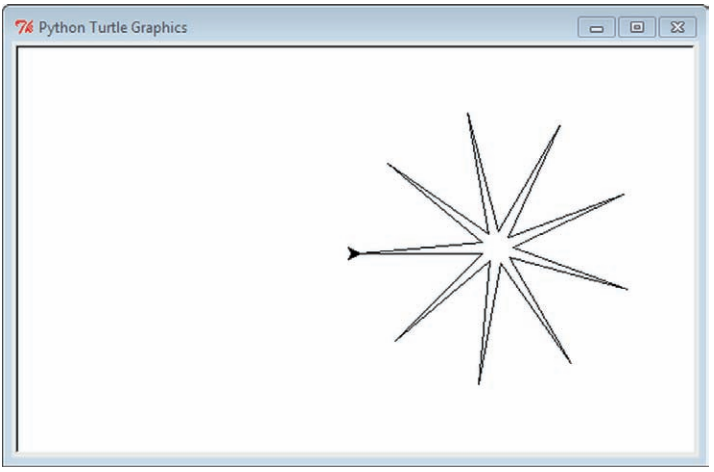
Wenn wir 13 Bälle in zwei gleich große Gruppen teilen wollten, bekämen wir zwei Gruppen mit je sechs Bällen und einem Ball als Rest:



Wenn wir prüfen, ob der Rest nach dem Teilen von `x` durch 2 gleich null ist, fragen wir in Wirklichkeit, ob ohne Rest in zwei Teile geteilt werden kann. Auf diese Weise kann man ganz elegant prüfen, ob eine Variable eine gerade Zahl enthält, da gerade Zahlen immer restlos durch zwei geteilt werden können.

In der fünften Zeile unseres Codes sagen wir der Schildkröte, dass sie um 175° nach links abbiegen soll (`t.left (175)`), falls `x` eine gerade Zahl ist (`if X % 2 == 0:`); anderenfalls (`else`) sagen wir ihr in der letzten Zeile, dass sie sich um 225° drehen soll (`t.left (225)`).

Hier siehst Du das Ergebnis dieses Codes:



12.3 Ein Auto zeichnen

Die Schildkröte kann mehr, als nur Sterne und einfache geometrische Formen zu zeichnen. In unserem nächsten Beispiel malen wir ein ziemlich primitiv aussehendes Auto. Als Erstes zeichnen wir die Karosserie des Autos. Gehe also auf **File ▶ New Window**, und gib in dem sich öffnenden Fenster den folgenden Code ein:

```
import turtle
t = turtle.Pen()
t.color(1,0,0)
t.begin_fill()
t.forward(100)
t.left(90)
t.forward(20)
t.left(90)
t.forward(20)
t.right(90)
t.forward(20)
t.left(90)
t.forward(60)
t.left(90)
t.forward(20)
t.right(90)
t.forward(20)
t.left(90)
t.forward(20)
t.end_fill()
```

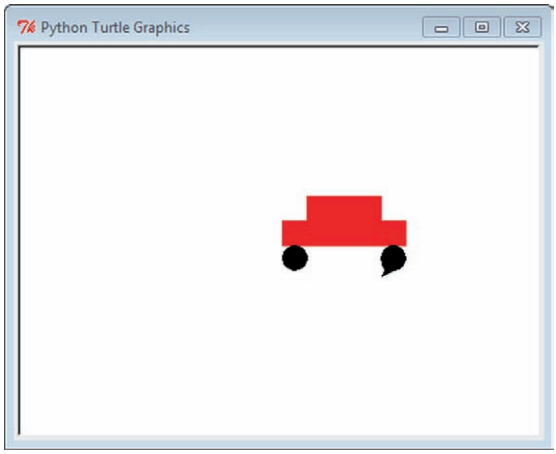
Als Nächstes kommt das erste Rad dran:

```
t.color(0,0,0)
t.up()
t.forward(10)
t.down()
t.begin_fill()
t.circle(10)
t.end_fill()
```

Zum Schluss zeichnen wir das zweite Rad:

```
t.setheading(0)
t.up()
t.forward(90)
t.right(90)
t.forward(10)
t.setheading(0)
t.begin_fill()
t.down()
t.circle(10)
t.end_fill()
```

Gehe auf **File ▶ Save As**. Gib einen Dateinamen wie zum Beispiel *Auto.py* ein. Gehe auf **Run ▶ Run Module**, um den Code auszuprobieren. Und hier ist unser Auto:



Vielleicht ist Dir aufgefallen, dass sich ein paar neue `turtle`-Funktionen in diesen Code eingeschlichen haben:

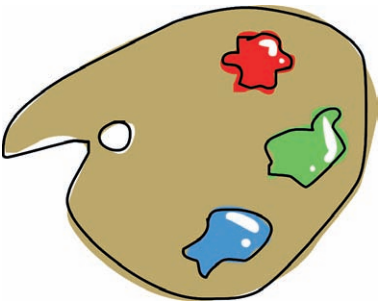
- Mit `color` ändert man die Farbe des Stifts.
- Mit `begin_ fill` und `end_fill` füllt man Flächen mit Farbe aus.
- Mit `circle` kann man einen Kreis in einer bestimmten Größe zeichnen.

- Mit `setheading` kann man die Schildkröte in eine bestimmte Richtung schauen lassen.

Schauen wir uns an, wie wir mit diesen Funktionen Farbe in unsere Zeichnungen bringen können.

12.4 Dinge einfärben

Die Funktion `color` nimmt drei Parameter auf. Der erste bestimmt den Rotanteil, der zweite den Grünanteil und der dritte den Blauanteil. Um zum Beispiel das helle Rot des Autos zu bekommen, haben wir `color(1,0,0)` benutzt und die Schildkröte dadurch angewiesen, mit einem zu 100 % roten Stift zu zeichnen.



Dieses Farb-Rezept aus Rot, Grün und Blau nennt man RGB.Auf Deinem Computer-Bildschirm werden durch die Mischung dieser *Primärfarben* auch alle anderen Farben dargestellt. Das ist ein bisschen so, als würdest Du mit Deinem Tuschkasten aus blauer und roter Farbe Violett oder aus Gelb und Rot Orange mischen.

Obwohl wir beim Mischen der Farben auf dem Computer-Monitor keine Farben aus dem Tuschkasten (sondern Licht) verwenden, hilft vielleicht die Vorstellung, dass dieses RGB-Rezept wie aus drei Farbeimern zusammengemischt wird: einem roten, einem grünen und einem blauen. Jeder dieser Eimer ist voll, und dem vollen Eimer weisen wir den Wert von 1 (oder 100 %) zu. Wir mischen dann die gesamte rote und grüne Farbe in einem Bottich zusammen, um Gelb zu erhalten (das wären 1 und 1 von jedem oder 100 % von jeder Farbe.

Kehren wir nun in die Welt des Codes zurück. Um mit der Schildkröte einen gelben Kreis zu zeichnen, würden wir je 100 % von der roten und der grünen Farbe verwenden, aber kein Blau:

```
>>> t.color(1,1,0)
>>> t.begin_fill()
>>> t.circle(50)
>>> t.end_fill()
```

Die `1,1,0` in der ersten Zeile stehen für 100 % Rot, 100 % Grün und 0 % Blau. In der nächsten Zeile sagen wir der Schildkröte, dass sie die von ihr gemalten Formen mit dieser RGB-Farbe (`t.begin_fill`) ausmalen und dann damit einen Kreis zeichnen soll (`t.circle`). In der letzten Zeile sagt `end_fill` der Schildkröte, dass sie den Kreis mit der RGB-Farbe ausfüllen soll.

Eine Funktion zum Zeichnen eines ausgefüllten Kreises

Um das Experimentieren mit verschiedenen Farben leichter zu machen, erzeugen wir eine Funktion aus dem Code, mit dem wir einen ausgefüllten Kreis gezeichnet haben.

```
>>> def meinKreis(Rot, Grün, Blau):
        t.color(Rot, Grün, Blau)
        t.begin_fill()
        t.circle(50)
        t.end_fill()
```

Indem wir nur die grüne Farbe verwenden, können wir uns einen hellen grünen Kreis zeichnen:

```
>>> meinKreis(0, 1, 0)
```

Oder wir können einen dunkleren grünen Kreis mit nur der Hälfte der grünen Farbe (0.5) zeichnen:

```
>>> meinKreis(0, 0.5, 0)
```

Um ein wenig mit den RGB-Farben auf Deinem Monitor zu spielen, versuche zunächst einen Kreis mit vollem Rot und dann mit halber Intensität (1 und 0.5) zu zeichnen, danach einen voll blauen Kreis und zum Schluss einen mit 50 % Blau:

```
>>> meinKreis(1, 0, 0)
>>> meinKreis(0.5, 0, 0)
>>> meinKreis(0, 0, 1)
>>> meinKreis(0, 0, 0.5)
```

Achtung!

Wenn Du Deine Leinwand aufräumen möchtest, kannst Du mit `t.reset()` Deine alten Zeichnungen entfernen. Denk daran, dass Du die Schildkröte auch ohne Linien zu zeichnen bewegen kannst, indem Du mit `t.up()` den Stift abhebst und ihn mit `t.down()` wieder absetzt.

Durch unterschiedliche Kombinationen von Rot, Grün und Blau kannst Du sehr viele verschiedene Farben erzeugen, wie etwa Gold:

```
>>> meinKreis(0.9, 0.75, 0)
```

Oder ein helles Rosa:

```
>>> meinKreis(1, 0.7, 0.75)
```

Und hier sind noch zwei verschiedene Orangetöne:

```
>>> meinKreis(1, 0.5, 0)
>>> meinKreis(0.9, 0.5, 0.15)
```

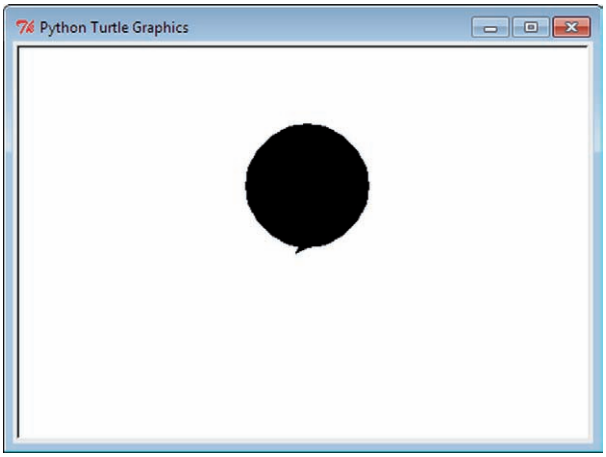
Mische Dir jetzt eigene Farben zusammen!

Reines Schwarz und Weiß erzeugen

Was geschieht, wenn man nachts alle Lampen abschaltet? Alles wird schwarz. Genau das Gleiche passiert mit den Farben auf dem Computer. Keine Farben ohne Licht, also enthält ein Kreis mit 0 für alle Primärfarben reines Schwarz:

```
>>> meinKreis(0, 0, 0)
```

Hier das Ergebnis:



Das Gegenteil trifft zu, wenn Du alle drei Farben auf 100 % setzt. In diesem Fall bekommst Du Weiß. Gib den folgenden Code ein, um den schwarzen Kreis wegzuwischen:

```
>>> meinKreis(1, 1, 1)
```

Eine Funktion zum Quadratezeichnen

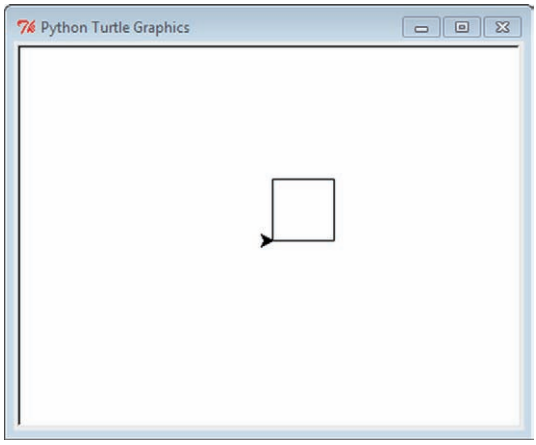
Du hast gesehen, dass wir Formen mit Farbe ausfüllen, indem wir der Schildkröte mit `begin_fill` sagen, dass sie damit anfangen soll, und mit der Funktion `end_fill`, dass jetzt Schluss ist. Jetzt machen wir noch ein paar weitere Experimente mit Formen und Färben. Wir nehmen dazu die Funktion zum Quadratezeichnen vom Anfang dieses Kapitels und bestimmen die Größe des Quadrats durch ihre Parameter.

```
>>> def meinQuadrat(Größe):
        for x in range(1, 5):
                t.forward(Größe)
                t.left(90)
```

Teste Deine Funktion, indem Du sie mit einer Größe von 50 aufrufst:

```
>>> meinQuadrat(50)
```

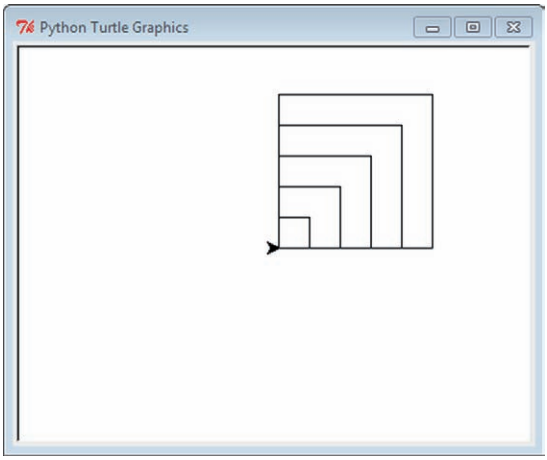
Das ergibt ein kleines Quadrat:



Jetzt probieren wir unsere Funktion mit unterschiedlichen Größen aus. Der folgende Code erzeugt fünf aufeinanderfolgende Quadrate mit den Seitenlängen 25, 50, 75, 100 und 125 Pixel:

```
>>> t.reset()
>>> meinQuadrat(25)
>>> meinQuadrat(50)
>>> meinQuadrat(75)
>>> meinQuadrat(100)
>>> meinQuadrat(125)
```

So sollten die Quadrate dann aussehen:



12.5 Ausgefüllte Quadrate zeichnen

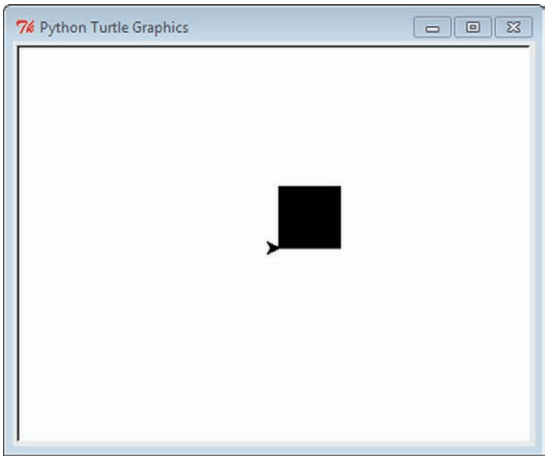
Um ausgefüllte Quadrate zu zeichnen, müssen wir zunächst die Leinwand zurücksetzen, mit dem Auffüllen anfangen und dann wieder unsere Quadrat-Funktion aufrufen:

```
>>> t.reset()
>>> t.begin_fill()
>>> meinQuadrat(50)
```

Bis Du das Füllen beendet hast, solltest Du ein leeres Quadrat sehen:

```
>>> t.end_fill()
```

Danach sollte das Quadrat so aussehen:



Lass uns jetzt diese Funktion so ändern, dass wir entweder ein ausgefülltes oder ein leeres Quadrat zeichnen können. Dafür brauchen wir einen weiteren Parameter sowie etwas komplizierteren Code:

```
>>> def meinQuadrat(Größe, ausgefüllt):
        if ausgefüllt == True:
                t.begin_fill()
        for x in range(1, 5):
                t.forward(Größe)
                t.left(90)
        if ausgefüllt == True:
                t.end_fill()
```

In der ersten Zeile ändern wir unsere Funktion, damit sie zwei Parameter aufnimmt: `Größe` und `ausgefüllt`. Als Nächstes prüfen wir, ob der Wert von `ausgefüllt` mit `if ausgefüllt == True` auf wahr gesetzt ist. Falls er es ist, rufen wir `begin_fill` auf, um der Schildkröte zu sagen, dass sie die gezeichnete Form ausfüllen soll. Danach durchlaufen wir die Schleife viermal (`for x in range(0, 4)`), um die vier Seiten des Rechtecks (durch Bewegung vorwärts und nach links) zu zeichnen. Danach prüfen wir mit `if ausgefüllt == True`, ob `ausgefüllt` wahr ist. Falls ja, stellen wir das Ausfüllen mit `t.end_fill` ab, und die Schildkröte füllt das Quadrat mit Farbe aus.

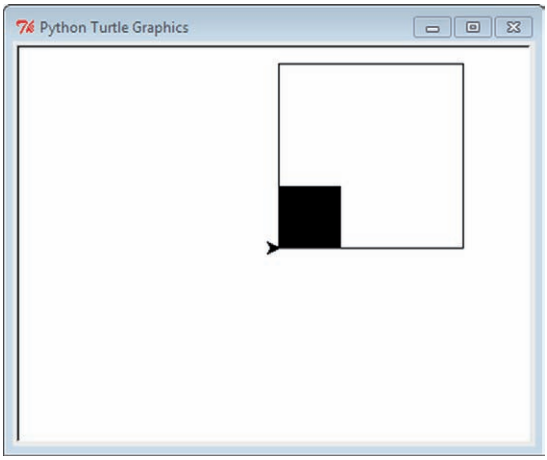
Jetzt können wir mit dieser Zeile ein ausgefülltes Quadrat zeichnen:

```
>>> meinQuadrat(50, True)
```

Oder wir zeichnen mit dieser Zeile ein unausgefülltes Quadrat:

```
>>> meinQuadrat(150, False)
```

Nach diesen beiden Aufrufen der Funktion `meinQuadrat` bekommen wir folgendes Bild, das ein bisschen wie ein quadratisches Auge aussieht:



Aber hier hört es noch lange nicht auf. Du kannst alle erdenklichen Formen zeichnen und sie mit Farbe füllen.

12.6 Ausgefüllte Sterne zeichnen

In unserem letzten Beispiel fügen wir unserem Stern, den wir zuvor gezeichnet haben, etwas Farbe hinzu. Der ursprüngliche Code sah so aus:

```
>>> t.reset()
>>> for x in range(1, 19):
        t.forward(100)
        if x % 2== 0:
            t.left(175)
        else:
            t.left(225)
```

Nun schreiben wir uns eine `meinStern`-Funktion. Wir werden die `if`-Anweisung aus der `meinStern`-Funktion verwenden und den Parameter `Größe` hinzufügen.

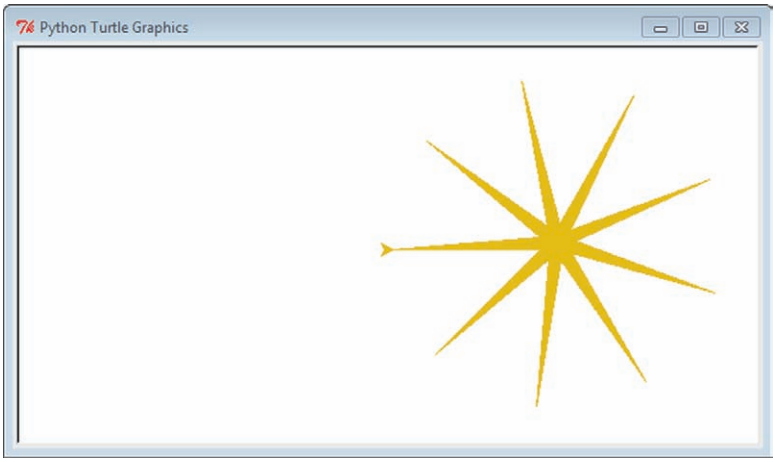
```
>>> def meinStern(Größe, ausgefüllt):
        if ausgefüllt == True:
            t.begin_fill()
        for x in range(1, 19):
            t.forward(Größe)
            if x % 2 == 0:
                t.left(175)
            else:
                t.left(225)
        if ausgefüllt == True:
            t.end_fill()
```

In den ersten beiden Zeilen dieser Funktion prüfen wir, ob `ausgefüllt` wahr ist, und falls dem so ist, beginnen wir mit dem Ausfüllen. In den letzten beiden Zeilen prüfen wir wieder, ob `ausgefüllt` wahr ist, und hören gegebenenfalls mit dem Ausfüllen auf. Wie schon bei der Funktion `meinQuadrat` setzen wir wieder die Größe des Sterns mit dem Parameter `Größe` fest und greifen auf diesen Wert zu, wenn wir `t.forward` aufrufen.

Wir legen jetzt die Farbe auf Gold fest (90 % Rot, 75 % Grün und 0 % Blau) und rufen die Funktion wieder auf.

```
>>> t.color(0.9, 0.75, 0)
>>> meinStern(120, True)
```

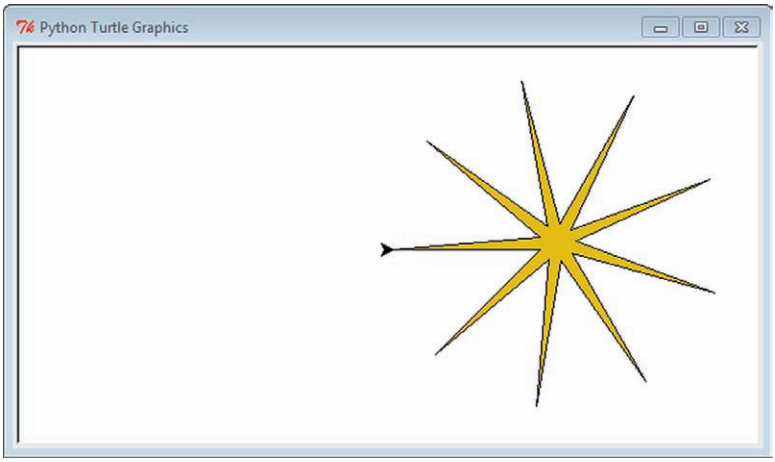
Damit zeichnet die Schildkröte diesen ausgefüllten Stern:



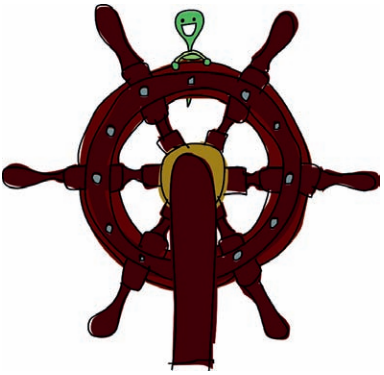
Damit der Stern eine Umrandung bekommt, änderst Du die Farbe in Schwarz und zeichnest den Stern, ohne ihn auszufüllen, noch einmal:

```
>>> t.color(0, 0, 0)
>>> meinStern(120, False)
```

Der goldene Stern hat jetzt eine schwarze Umrandung:



12.7 Was Du gelernt hast



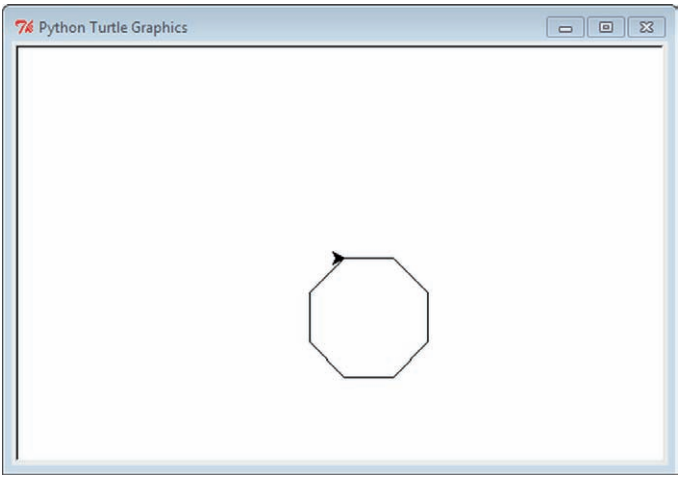
In diesem Kapitel hast Du gelernt, wie man mit dem Modul `turtle` ein paar grundlegende geometrische Formen zeichnet und wie man die Schildkröte mit `for`-Schleifen und der `if`-Anweisung auf dem Monitor steuert. Wir haben die Farbe des Stifts der Schildkröte verändert und die von ihr gezeichneten Formen ausgefüllt. Wir haben auch mit einigen Funktionen den Code anderer Zeichnungen wiederverwertet, um mit nur einem Aufruf einer Funktion ganz einfach Formen mit unterschiedlichen Farben zu zeichnen.

12.8 Programmier-Puzzles

Bei den folgenden Experimenten zeichnest Du Deine eigenen Formen mit der Schildkröte. Die Lösungen findest Du wie immer unter *www.dpunkt.de/python*.

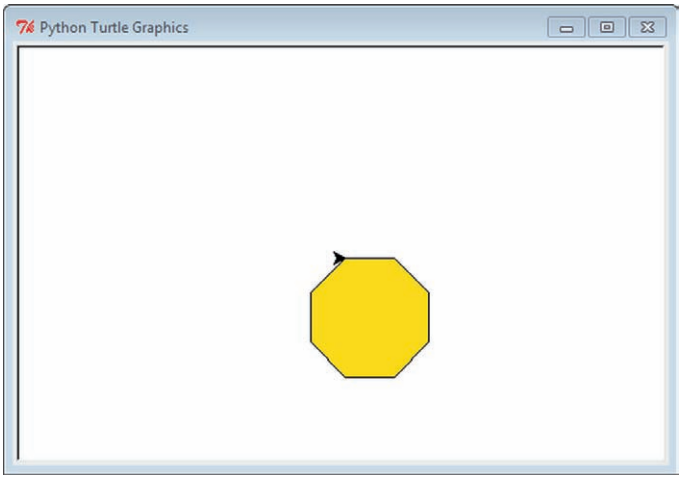
#1: Ein Oktagon zeichnen

Wir haben in diesem Kapitel Sterne, Quadrate und Rechtecke gezeichnet. Wie wäre es, wenn wir jetzt eine Funktion erstellen, mit der wir eine achtseitige Form, wie etwa ein Oktagon, zeichnen? (Hinweis: Versuche, die Schildkröte um 45° zu drehen.)



#2: Ein ausgefülltes Oktagon zeichnen

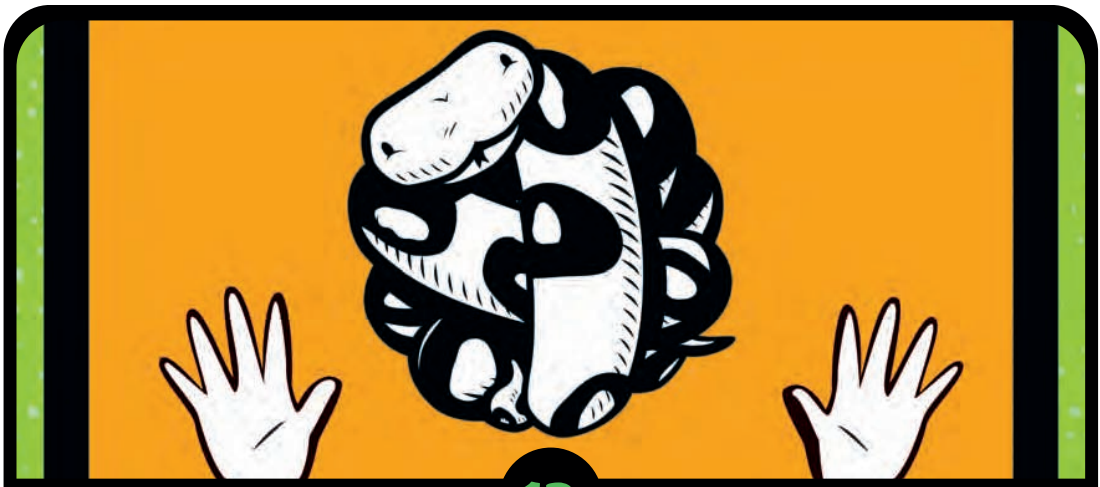
Jetzt, wo Du eine Funktion zum Zeichnen eines Oktagons hast, ändere sie so ab, dass sie ein ausgefülltes Oktagon zeichnet. Versuche, ein Oktagon mit einer Umrandung zu zeichnen, wie wir es bei dem Stern getan haben.



#3: Noch eine Funktion zum Sterne Zeichnen

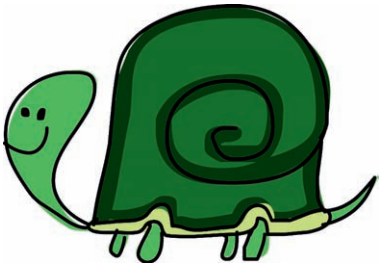
Erzeuge eine Funktion zum Zeichnen eines Sterns, die zwei Parameter aufnimmt: die Größe und die Anzahl der Zacken. Der Anfang der Funktion wird in etwa so aussehen:

```
>>> def zeichne_Stern(Größe, Zacken):
```



13
Bessere Grafiken mit tkinter

Das Problem beim Zeichnen mit einer Schildkröte besteht darin, dass … Schildkröten … wirklich … langsam … sind. Selbst wenn eine Schildkröte mit ihrer Höchstgeschwindigkeit läuft, ist sie immer noch nicht sehr schnell. Was bei Schildkröten kein Problem ist, ist bei Computergrafiken sehr wohl eins.



Computergrafiken, vor allem in Spielen, müssen sehr schnell ablaufen. Wenn Du eine Spielkonsole hast oder auf dem Computer spielst, denk mal einen Moment lang über die Grafiken nach, die Du auf dem Monitor siehst. Zweidimensionale (2D) Grafiken sind flach – die Figuren bewegen sich im Allgemeinen nur nach oben, unten, links oder rechts – wie in vielen Nintendo DS-, PlayStation Portable- (PSP) oder Handyspielen. Bei pseudo-dreidimensionalen (3D) Spielen – die fast 3D sind – sind die Bilder etwas realistischer, aber die Figuren bewegen sich meist nur in Relation zu einer Ebene (dies nennt man auch *isometrische Grafiken*). Und schließlich haben wir noch die 3D-Spiele, bei denen die Bilder auf dem Monitor gezeichnet werden, um die Realität nachzuahmen. Egal ob unsere Spiele nun 2D-, Pseudo-3D oder echte 3D-Grafiken darstellen, sie haben doch eines gemeinsam: Sie müssen sich alle sehr schnell auf den Computermonitor aufbauen.

Falls Du noch nie versucht hast, eine eigene Animation zu erstellen, probiere einmal Folgendes aus:

1. Nimm einen Block Papier, und zeichne etwas in die untere Ecke (zum Beispiel ein Strichmännchen).
2. In die Ecke der nächsten Seite malst Du das gleiche Strichmännchen, bewegst aber sein Bein ein wenig.
3. Auf die nächste Seite zeichnest Du das gleiche Strichmännchen, bewegst sein Bein aber noch ein bisschen mehr.
4. Füge immer mehr Seiten hinzu, auf die Du jeweils ein verändertes Strichmännchen in die Ecke zeichnest.

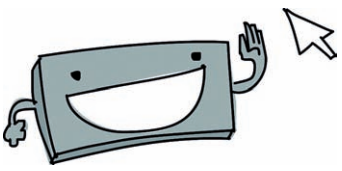
Wenn Du damit fertig bist, blätterst Du schnell durch die Seiten. Du siehst jetzt, wie sich das Strichmännchen bewegt. Dies ist das Grundprinzip aller Animationen, seien sie nun Zeichentrickfilme im Fernsehen oder Spiele auf Deiner Konsole oder auf Deinem Computer. Ein Bild wird dargestellt und nach einer kleinen Änderung noch einmal dargestellt, damit die Illusion einer Bewegung entsteht. Damit es so aussieht, als bewege sich das Bild, musst Du jedes Bild dieser Animation sehr schnell darstellen.

Python bietet verschiedene Möglichkeiten, um Grafiken zu erzeugen. Zusätzlich zum Modul `turtle` kannst Du externe Module (die separat installiert werden müssen) ebenso verwenden wie das Modul `tkinter`, das schon zu Deiner Standard-Python-Installation gehören sollte. Mit `tkinter` kann man vollständige Anwendungen, wie etwa einfache Textverarbeitungen, aber auch einfache Zeichnungen erstellen. In diesem Kapitel werden wir herausfinden, wie man mit `tkinter` Grafiken erzeugt.

13.1 Einen klickbaren Button erzeugen

In unserem ersten Beispiel benutzen wir `tkinter`, um eine einfache Anwendung mit einem Button zu erzeugen. Gib dazu diesen Code ein:

```
>>> from tkinter import *
>>> tk = Tk()
>>> btn = Button(tk, text = "Klick mich")
>>> btn.pack()
```



In der ersten Zeile importieren wir die Inhalte des Moduls `tkinter`. Indem wir `from module-name import *` schreiben, können wir die Inhalte eines Moduls verwenden, ohne dessen Namen zu benutzen. Wenn wir dagegen (wie in den vorherigen Beispielen) `import turtle` schreiben, müssen wir den Modulnamen mit einschließen, um an seine Inhalte zu kommen:

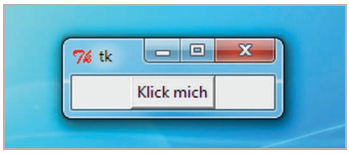
```
import turtle
t = turtle.Pen()
```

Mit `import *` müssen wir nicht `turtle.Pen` aufrufen, wie wir es in den Kapiteln 5 und 12 getan haben. Beim Modul `turtle` ist das nicht so wichtig, aber wenn man Module mit vielen Klassen und Funktionen verwendet, muss man sehr viel weniger tippen.

```
from turtle import *
t = Pen()
```

In der nächsten Zeile unseres Button-Beispiels erzeugen wir eine Variable, die ein Objekt der Klasse `tk` mit `tk = tk()` enthält, genau wie wir ein `Pen`-Objekt für die Schildkröte erzeugt haben. Das `tk`-Objekt erzeugt ein einfaches Fenster, in das wir andere Dinge – wie etwa Buttons, Eingabezeilen oder eine Leinwand zum Bemalen – einfügen können. Es ist auch die Hauptklasse des Moduls `tkinter`: Ohne ein Objekt der `tk`-Klasse wirst Du keine Grafik oder Animation erstellen können.

In der dritten Zeile erzeugen wir mit `btn = Button` einen Button und führen die Variable `tk` als ersten Parameter ein. `"Klick mich"` wird der Text, der durch `tk, text = "Klick mich"` auf dem Button erscheint. Obwohl wir den Button dem Fenster hinzugefügt haben, wird er so lange nicht sichtbar sein, bis Du die Zeile `btn.pack()` eingegeben hast, die dem Button sagt, dass er erscheinen soll. Dadurch wird auch alles auf dem Monitor richtig ausgerichtet, falls sich noch andere Buttons oder Objekte darauf befinden. Das Ergebnis sollte in etwa so aussehen:



Mit dem *Klick mich*-Button kann man noch nicht viel anfangen. Solange wir den Code nicht ein wenig ändern, kannst Du den ganzen Tag darauf herumklicken, ohne dass etwas passiert. (Bitte schließe dazu zunächst das Fenster, das wir vorher erzeugt haben!)

Als Erstes erzeugen wir eine Funktion, die ein wenig Text anzeigt:

```
>>> def Hallo():
        print('Hallo')
```

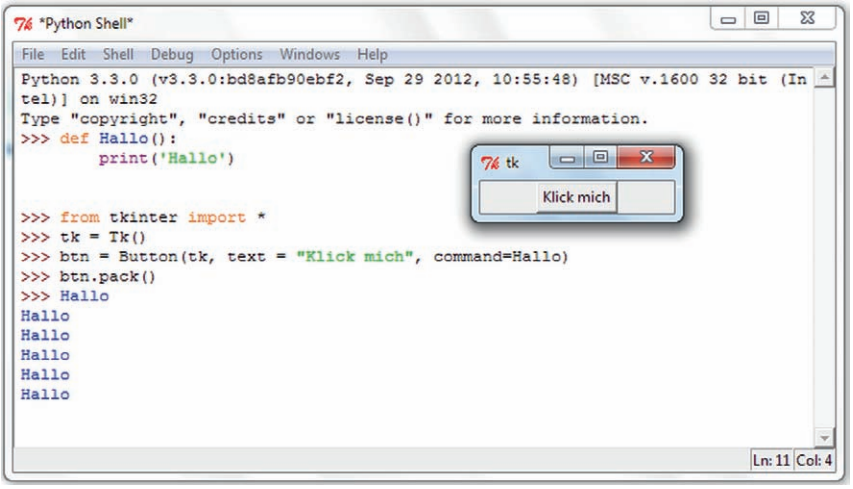
Anschließend ändern wir unser Beispiel oben, damit wir diese neue Funktion einsetzen können:

```
>>> from tkinter import *
>>> tk = Tk()
>>> btn = Button(tk, text = "Klick mich", command=Hallo)
>>> btn.pack()
```

Achte einmal darauf, dass wir nur eine ganz kleine Änderung im Code vorgenommen haben: Wir haben den Parameter `command` hinzugefügt, der Python sagt, dass es die Funktion `Hallo` aufrufen soll, sobald auf den Button geklickt wird.

Wenn Du jetzt auf den Button klickst, siehst Du, wie »Hallo« in die Shell geschrieben wird. Das passiert jedes Mal, wenn Du auf den Button klickst.

Im folgenden Beispiel habe ich den Button sechsmal geklickt.



Dies ist das erste Mal, dass wir in unseren Code-Beispielen unseren Parametern einen Namen gegeben haben. Sehen wir uns das einmal genauer an, bevor wir mit unseren Zeichnungen weitermachen.

13.2 Einsatz von benannten Parametern

Benannte Parameter sind wie normale Parameter. Allerdings müssen die Werte für eine Funktion keine bestimmte Reihenfolge haben (der erste Wert für den ersten Parameter, der zweite Wert für den zweiten Parameter usw.), sondern die Werte werden benannt, sodass sie in beliebiger Reihenfolge eingesetzt werden können.

Manchmal haben Funktionen eine ganze Menge Parameter, und wir müssen nicht immer für jeden Parameter einen Wert angeben. Mit benannten Parametern können wir genau denjenigen Parametern Werte liefern, die sie auch benötigen.

Nehmen wir beispielsweise an, wir hätten eine Funktion namens `Person`, die zwei Parameter aufnimmt: `Breite` und `Höhe`.

```
>>> def Person(Breite, Höhe):
    print('Ich bin %s Meter breit und %s Meter groß' % (Breite, Höhe))
```

Üblicherweise rufen wir diese Funktion folgendermaßen auf:

```
>>> Person(2, 1)
Ich bin 2 Meter breit und 1 Meter groß
```

Mit benannten Parametern konnten wir diese Funktion aufrufen und den Parameternamen mit jedem Wert bestimmen:

```
>>> Person(Breite=2, Höhe=1)
Ich bin 2 Meter breit und 1 Meter groß
```

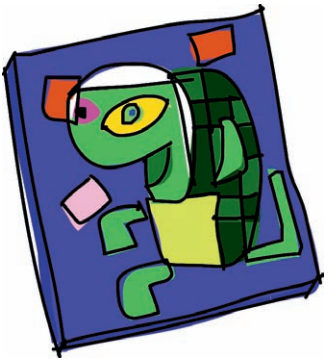
Benannte Parameter werden besonders nützlich, wenn wir mehr mit dem Modul `tkinter` arbeiten.

13.3 Eine Leinwand zum Zeichnen erzeugen

Buttons sind zwar ganz nett, helfen uns aber nicht so richtig weiter, wenn wir auf den Monitor Dinge zeichnen wollen. Wenn wir etwas Richtiges malen wollen, brauchen wir eine andere Komponente: ein Leinwand-Objekt (`canvas`), das zur Klasse `Canvas` gehört (und aus dem Modul `tkinter` stammt).

Um eine Leinwand zu erzeugen, müssen wir deren Breite (`width`) und Höhe (`height`) Python in Pixeln mitteilen. Ansonsten ähnelt der Code dem für Buttons. Hier siehst Du ein Beispiel:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=500, height=500)
>>> canvas.pack()
```



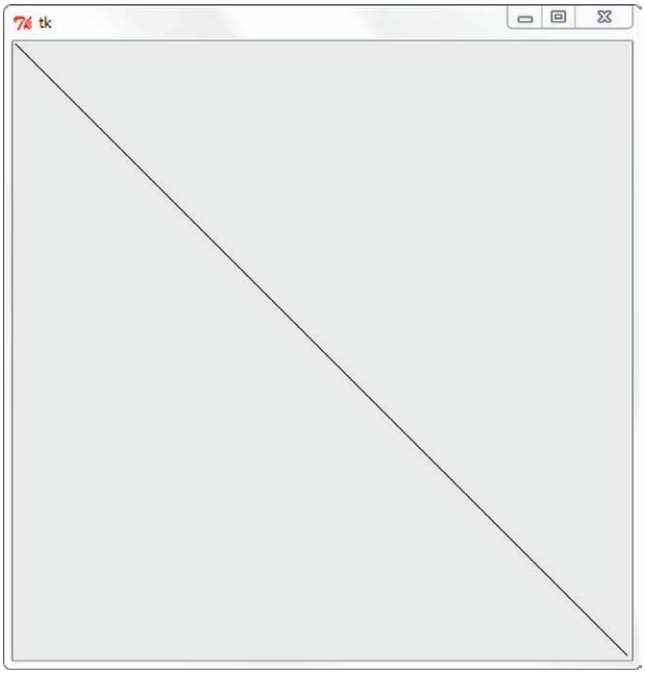
Wie schon beim Button-Beispiel erscheint, nachdem Du `tk = Tk()` eingegeben hast, ein Fenster. In der letzten Zeile wird durch canvas.pack() die Breite und Höhe der Leinwand auf jeweils 500 Pixel vergrößert, wie wir in der dritten Zeile festgelegt haben. Wie schon bei dem Beispiel mit dem Button sagt die Funktion `pack` der Leinwand, dass sie sich an der richtigen Position innerhalb des Fensters aufbauen soll. Solange diese Funktion nicht aufgerufen wird, wird nichts vernünftig dargestellt.

13.4 Linien zeichnen

Um eine Linie auf die Leinwand zu zeichnen, benutzen wir Pixel-Koordinaten. Mit *Koordinaten* wird die Position eines Pixels auf einer Fläche bestimmt. Auf einer Leinwand von `tkinter` beschreiben die Koordinaten, an welcher Stelle von

links nach rechts und an welcher Stelle von oben nach unten das Pixel platziert wird.

Da unsere Leinwand in unserem Beispiel 500 Pixel breit und 500 Pixel hoch ist, sind die Koordinaten der unteren rechten Ecke `(500,500)`. Um eine Linie wie in dem folgenden Bild zu zeichnen, verwenden wir die Startkoordinate `(0,0)` und die Zielkoordinate `(500,500)`.



Die Koordinaten legen wir mit der Funktion `create_line` fest:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=500, height=500)
>>> canvas.pack()
>>> canvas.create_line(0, 0, 500, 500)
1
```

Die Funktion `create_line` gibt die Zahl 1 zurück. Dabei handelt es sich um eine sogenannte *identifizierende Nummer*, über die Du später noch mehr erfahren wirst. Um das Gleiche im Modul `turtle` zu erzielen, hätten wir folgenden Code gebraucht:

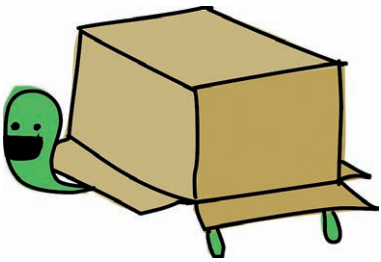
```
>>> import turtle
>>> turtle.setup(width=500, height=500)
>>> t = turtle.Pen()
>>> t.up()
```

```
>>> t.goto(-250, 250)
>>> t.down()
>>> t.goto(500, -500)
```

Der Code von `tkinter` stellt also schon eine Verbesserung dar: Er ist etwas kürzer und auch ein bisschen einfacher.

Jetzt schauen wir uns einige der verfügbaren Funktionen im Objekt `canvas` an, mit denen wir einige interessantere Zeichnungen machen können.

13.5 Kästchen zeichnen



Mit dem Modul `turtle` haben wir Kästchen gemalt, indem wir uns vorwärts bewegt haben, abgebogen sind, uns vorwärts bewegt haben, abgebogen sind und so weiter. Wir waren auch in der Lage, ein rechteckiges oder quadratisches Kästchen zu zeichnen, indem wir uns unterschiedlich weit vorwärts bewegt haben.

Das Modul `tkinter` erleichtert es sehr, ein Quadrat oder Rechteck zu zeichnen. Alles, was Du dazu brauchst, sind die Koordinaten der Ecken. Hier ist ein Beispiel (Du kannst die anderen Fenster jetzt schließen):

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_rectangle(10, 10, 50, 50)
```

Bei diesem Code verwenden wir `tkinter`, um eine Leinwand von 400 Pixeln Breite und 400 Pixeln Höhe zu erzeugen und anschließend ein Quadrat in die obere linke Ecke zu zeichnen:

Bei den Parametern, die wir in der letzten Zeile des Codes an `canvas.create_rectangle` übergeben, handelt es sich um die Koordinaten der oberen linken und unteren rechten Ecke des Quadrats. Wir geben diese Koordinaten als Abstand von der linken und der Oberseite der Leinwand an. In diesem Fall sind die ersten beiden Koordinaten (die obere linke Ecke) 10 Pixel von der linken und 10 Pixel von der Oberseite entfernt (die ersten beiden Zahlen: `10, 10`). Die Ecke unten rechts im Quadrat befindet sich 50 Pixel von der linken Seite der Leinwand und 50 von der Oberseite entfernt (das nächste Zahlenpaar: `50, 50`).

Wir bezeichnen diese beiden Koordinatensätze mit `x1`, `y1` und `x2`, `y2`. Um nun ein Rechteck zu zeichnen, können wir einfach den Abstand der zweiten Ecke von der linken Seite der Leinwand vergrößern (indem wir den Wert des Parameters `x2` erhöhen):

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_rectangle(10, 10, 300, 50)
```

In diesem Beispiel sind die Koordinaten des Rechtecks, also dessen Position im Fenster, oben links (10, 10) und unten rechts (300, 50). Als Ergebnis bekommen wir ein Reckteck, das die gleiche Höhe wie unser Quadrat (50 Pixel) hat, aber sehr viel breiter ist.

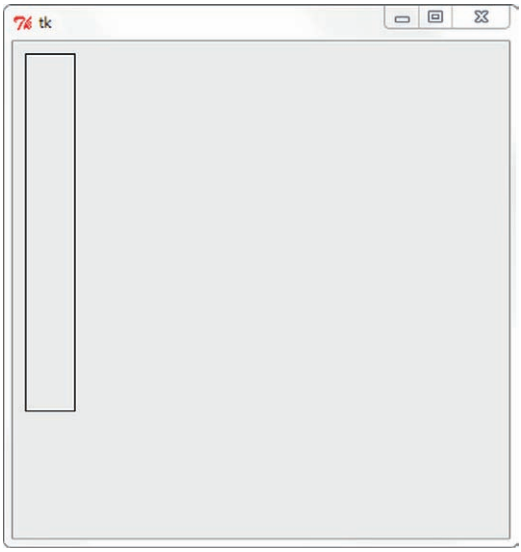
Indem wir den Abstand der zweiten Ecke von der oberen Kante der Leinwand vergrößern (also den Wert des Parameters `y2` erhöhen), können wir ein hochkantiges Rechteck zeichnen:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_rectangle(10, 10, 50, 300)
```

Bei diesem Aufruf der Funktion `create_rectangle` sagen wir im Prinzip Folgendes:

- Gehe 10 Pixel seitlich über die Leinwand (von oben links).
- Gehe 10 Pixel die Leinwand hinunter. Dies ist der Startpunkt des Rechtecks.
- Zeichne das Rechteck 50 Pixel nach rechts.
- Zeichne das Rechteck 300 Pixel nach unten.

Das Endergebnis sollte folgendermaßen aussehen:



Ganz viele Rechtecke zeichnen

Wie wäre es, wenn wir Rechtecke unterschiedlicher Größe auf die Leinwand brächten? Das ginge, indem wir das Modul `random` importieren und dann eine Funktion erzeugen, die eine Zufallszahl für die Koordinaten der oberen linken und unteren rechten Ecken verwendet.

Wir werden dabei eine Funktion benutzen, die aus dem Modul `random` stammt und sich `randrange` nennt. Wenn wir dieser Funktion eine Zahl geben, gibt sie uns eine zufällige Ganzzahl zwischen null und der Zahl zurück, die wir ihr mitgegeben haben. Wenn wir zum Beispiel `randrange(10)` aufrufen, würde sie eine Zahl zwischen 0 und 9 zurückgeben, und bei `randrange(100)` käme eine Zahl zwischen 0 und 99 zurück.

Hier siehst Du, wie man `randrange` in einer Funktion benutzt. Erzeuge ein neues Fenster, indem Du auf **File ▶ New Window** gehst und folgenden Code eingibst:

```
from tkinter import *
import random
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
def random_rectangle(width, height):
    x1 = random.randrange(width)
    y1 = random.randrange(height)
    x2 = x1 + random.randrange(width)
    y2 = y1 + random.randrange(height)
    canvas.create_rectangle(x1, y1, x2, y2)
```

Als Erstes definieren wir unsere Funktion (`def random_rectangle`), die zwei Parameter aufnimmt: Breite (`width`) und Höhe (`height`). Danach erzeugen wir mit der Funktion `randrange` die Variablen für die obere linke Ecke des Rechteckes und geben die Breite und die Höhe als Parameter mit `x1 = random.randrange(width)` beziehungsweise `y1 = random.randrange(height)` weiter. Mit der zweiten Zeile dieser Funktion sagen wir also: »Erzeuge eine Variable `x1`, und setze ihren Wert zufällig auf eine Zahl zwischen 0 und dem Wert des Parameters `width`.«

Die nächsten beiden Zeilen erzeugen Variablen für die Ecke unten rechts im Rechteck. Sie berücksichtigen dabei die Koordinaten der oberen linken Ecke (`x1` bzw. `y1`) und fügen diesen Werten eine Zufallszahl hinzu. Die dritte Zeile der Funktion sagt also: »Erzeuge die Variable `x2`, indem Du dem Wert, den wir schon für `x1` berechnet haben, eine Zufallszahl hinzufügst«.

Mit `canvas.create_rectangle` benutzen wir schlussendlich die Variablen `x1`, `y1`, `x2` und `y2`, um das Rechteck auf die Leinwand zu zeichnen.

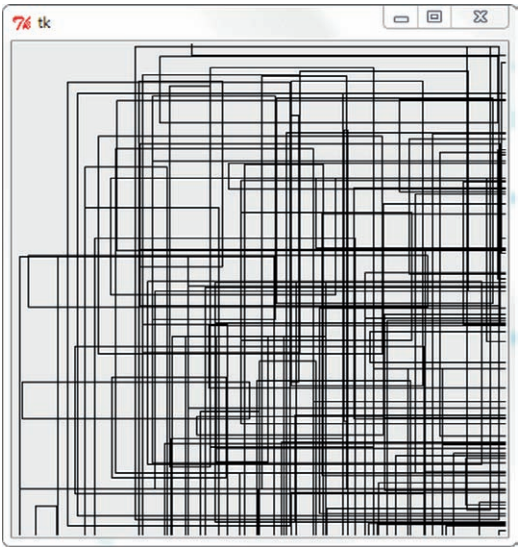
Um unsere Funktion `random_rectangle` auszuprobieren, geben wir die Breite und Höhe der Leinwand an. Füge dem bereits eingegebenen Code folgende Zeile hinzu:

```
random_rectangle(400, 400)
```

Speichere den gerade eingegebenen Code (gehe auf **File ▶ Save**, und gib ihm einen Dateinamen, wie zum Beispiel *Zufallsrechtecke.py*). Gehe dann auf **Run ▶ Run Module**. Sobald Du die Funktion bei der Arbeit gesehen hast, fülle das ganze Fenster mit Rechtecken, indem Du sie mittels einer Schleife `random_rectangle` mehrfach aufrufst. Probieren wir es einmal mit einer `for`-Schleife von 100 zufälligen Rechtecken. Füge den folgenden Code hinzu, speichere Deine Arbeit, und lass das Programm noch einmal ablaufen:

```
for x in range(0, 100):
    random_rectangle(400, 400)
```

Dieser Code produziert ein hübsches Durcheinander, aber es sieht auch ein bisschen wie moderne Kunst aus:

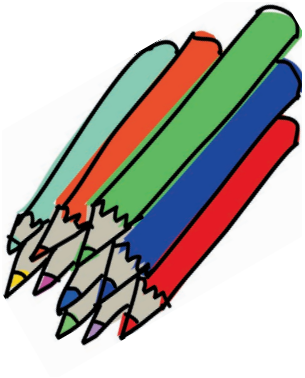


Die Farbe bestimmen

Selbstverständlich wollen wir unseren Grafiken auch Farben geben. Lass uns daher die Funktion `random_rectangle` dahingehend ändern, dass sie durch einen zusätzlichen Parameter (`fill_color`) den Rechtecken Farben gibt. Gib diesen Code in ein neues Fenster ein, und nenne die Datei beim Speichern *Farbrechtecke.py*:

```
from tkinter import *
import random
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()

def random_rectangle(width, height, fill_color):
    x1 = random.randrange(width)
    y1 = random.randrange(height)
    x2 = random.randrange(x1 + random.randrange(width))
    y2 = random.randrange(y1 + random.randrange(height))
    canvas.create_rectangle(x1, y1, x2, y2, fill=fill_color)
```

Die Funktion `create_rectangle` nimmt den Parameter `fill_color` auf, der die Farbe beim Zeichnen der Rechtecke bestimmt.



Wir können in eine solche Funktion auch benannte Farben einfügen (bei einer Leinwand von 400 Pixeln Breite und 400 Pixeln Höhe), um einen Haufen unterschiedlich gefärbter Rechtecke zu erzeugen. Beim Ausprobieren dieses Beispiels möchtest Du eventuell durch Kopieren und Einfügen etwas Tipparbeit sparen. Dazu wählst Du den zu kopierenden Text aus, drückst Ctrl-C, um ihn zu kopieren, klickst in eine leere Zeile und drückst Ctrl-V, um ihn einzufügen. Füge diesen Code gleich unter die Funktion in der Datei *Farbrechtecke.py* ein:

```
random_rectangle(400, 400, 'green')
random_rectangle(400, 400, 'red')
random_rectangle(400, 400, 'blue')
random_rectangle(400, 400, 'orange')
random_rectangle(400, 400, 'yellow')
random_rectangle(400, 400, 'pink')
random_rectangle(400, 400, 'purple')
random_rectangle(400, 400, 'violet')
random_rectangle(400, 400, 'magenta')
random_rectangle(400, 400, 'cyan')
```

Viele dieser benannten Farben stellen die Farben dar, wie Du sie erwartest, andere wiederum produzieren Fehlermeldungen (je nachdem, ob du Windows, MacOSX oder Linux benutzt).

Aber wie ist das mit einer selbst gewählten Farbe, die nicht genau einer benannten Farbe entspricht? Erinnere dich an Kapitel 12, wo wir die Farbe des Stifts der Schildkröte durch Prozentanteile der Farben Rot, Grün und Blau bestimmt haben. Den Anteil der Primärfarben (Rot, Grün und Blau) in einer Farbmischung bei `tkinter` zu bestimmen, ist etwas komplizierter, aber wir werden das gleich verstehen.

Als wir mit dem Modul `turtle` gearbeitet haben, haben wir mit 90 % Rot, 75 % Grün und ohne Blau die Goldfarbe erzeugt. In `tkinter` erzeugen wir die gleiche Goldfarbe mit dieser Zeile:

```
random_rectangle(400, 400, '#ffd800')
```

Das Doppelkreuz (`#`) vor dem Wert `ffd800` sagt Python, dass wir eine *Hexadezimalzahl* liefern. Beim Schreiben von Programmen werden Zahlen häufig hexadezimal dargestellt. Dabei wird statt wie bei Dezimalzahlen mit der Basis 10 (0–9) die Basis 16 (0–9 und danach A–F) verwendet. Falls Du im Mathematikunterricht noch nichts von der Basis gehört hast, merke Dir dazu einfach nur, dass Du eine normale Dezimalzahl durch einen Format-Platzhalter in einem String in eine

hexadezimale Zahl umwandeln kannst: `%x` (siehe Abschnitt »Werte in Strings einbetten« auf S. 31). Um zum Beispiel die Dezimalzahl 15 in eine Hexadezimalzahl umzuwandeln, kannst Du Folgendes schreiben:

```
>>> print('%x' % 15)
f
```

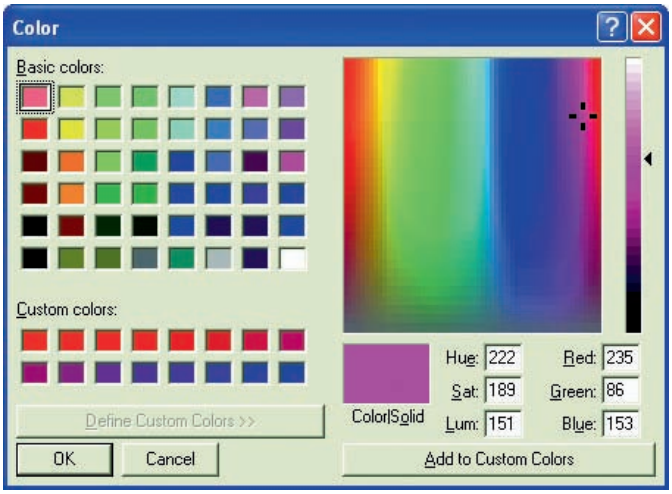
Um sicherzustellen, dass unsere Zahl mindestens zwei Stellen hat, können wir den Format-Platzhalter leicht abwandeln:

```
>>> print('%02x' % 15)
0f
```

Das Modul `tkinter` bietet eine ganz einfache Möglichkeit, einen hexadezimalen Farbwert zu bekommen. Versuche, folgenden Code zu *Farbrechtecke.py* hinzuzufügen (Du kannst die anderen Aufrufe der Funktion `random_rectangle` entfernen):

```
from tkinter import *
colorchooser.askcolor()
```

Daraufhin wird Dir ein Farbauswahlfenster gezeigt:



Sobald Du eine Farbe ausgewählt hast und auf **OK** klickst, wird ein Tupel angezeigt. Dieses Tupel enthält ein weiteres Tupel, das drei Zahlen und einen String enthält:

```
>>> colorchooser.askcolor()
((235.91796875, 86.3359375, 153.59765625), '#eb5699')
```

Die drei Zahlen stehen für den Rot-, Grün- und Blau-Anteil. In `tkinter` werden die Primärfarben einer Farbmischung durch eine Zahl zwischen 0 und 255 ange-

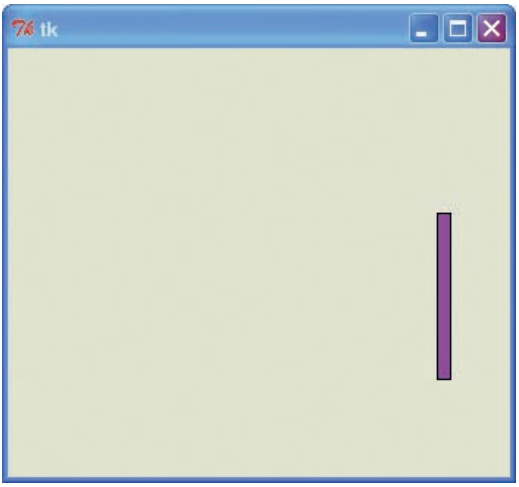
geben (im Unterschied zu dem prozentualen Anteil in jeder Primärfarbe im Modul turtle). Der String im Tupel enthält die hexadezimalen Versionen dieser drei Zahlen.

Du kannst den String-Wert entweder kopieren und einfügen oder aber den Tupel als Variable speichern und dann die Indexposition des Hexadezimalwertes verwenden.

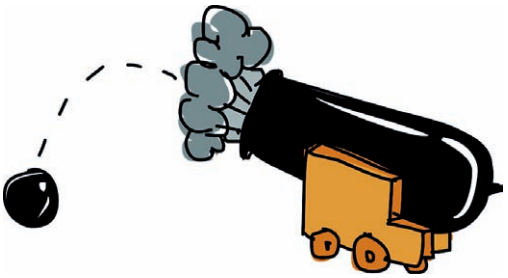
Lass uns jetzt die Funktion `random_rectangle` verwenden, um zu sehen, wie das funktioniert.

```
>>> c = colorchooser.askcolor()
>>> random_rectangle(400, 400, c[1])
```

Hier ist das Ergebnis:

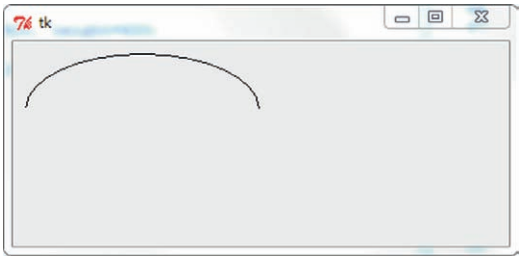


13.6 Bögen zeichnen



Ein Bogen ist ein Segment eines Kreisumfangs oder einer anderen Kurve. Um sie jedoch in `tkinter` zu zeichnen, musst Du sie innerhalb eines Rechtecks mit der Funktion `create_arc` zeichnen:

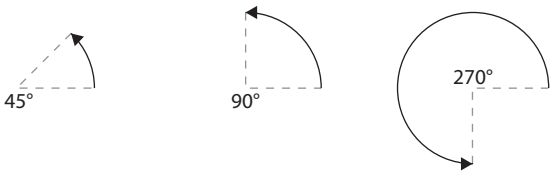
```
canvas.create_arc(10, 10, 200, 100, extent=180, style=ARC)
```



Falls Du in der Zwischenzeit alle `tkinter`-Fenster geschlossen oder IDLE neu gestartet hast, musst Du `tkinter` erneut importieren und die Leinwand mit diesem Code wieder erzeugen:

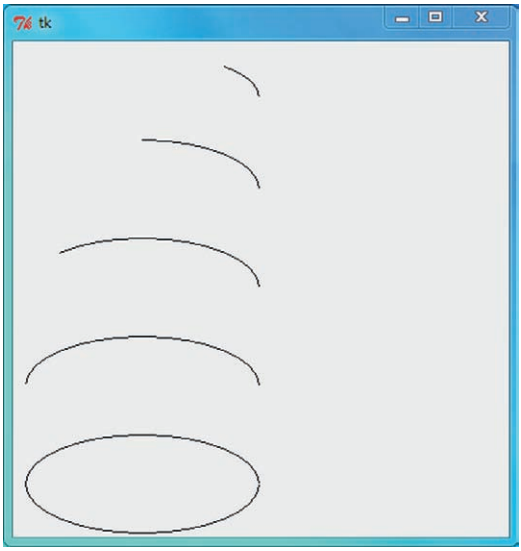
```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_arc(10, 10, 200, 100, extent=180, style=ARC)
```

Dieser Code platziert die obere linke Ecke des Rechtecks, in dem der Bogen enthalten ist, auf die Koordinaten (10, 10), also 10 Pixel zur Seite und 10 Pixel nach unten, und die untere rechte Ecke auf die Koordinaten (200,100), also 200 Pixel zur Seite und 100 Pixel nach unten. Mit dem nächsten Parameter, `extent`, wird der Winkelgrad des Bogens festgelegt. Aus Kapitel 5 weißt Du noch, dass man mit Graden die Strecke um einen Kreis misst. Hier siehst Du drei Beispiele für Bögen, in denen wir uns 45°, 90° beziehungsweise 270° im Kreis bewegen:



Der folgende Code zeichnet unterschiedliche Bögen auf die Seite, sodass Du beobachten kannst, was passiert, wenn wir unterschiedliche Gradzahlen mit der Funktion `create_arc` verwenden.

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_arc(10, 10, 200, 80, extent=45, style=ARC)
1
>>> canvas.create_arc(10, 80, 200, 160, extent=90, style=ARC)
2
>>> canvas.create_arc(10, 160, 200, 240, extent=135, style=ARC)
3
>>> canvas.create_arc(10, 240, 200, 320, extent=180, style=ARC)
4
>>> canvas.create_arc(10, 320, 200, 400, extent=359, style=ARC)
5
```



Achtung!

Im letzten Kreis verwenden wir 359° statt 360°, da `tkinter` 360° als das Gleiche wie 0° versteht und dann gar nichts zeichnen würde.

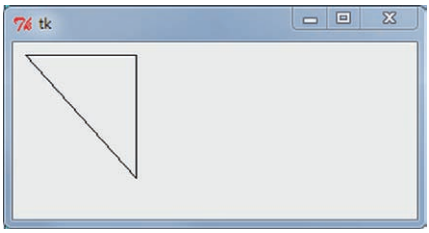
13.7 Polygone zeichnen

Als Polygone bezeichnet man Formen mit drei oder mehr Seiten. Es gibt gleichmäßig geformte Polygone wie Dreicke, Quadrate, Rechtecke, Fünfecke und Siebenecke und so weiter, aber auch unregelmäßig geformte mit ungleich langen Kanten, viel mehr Kanten sowie völlig ungewöhnliche Formen.

Wenn Du mit `tkinter` Polygone zeichnest, musst Du für jeden Punkt die Koordinaten liefern. Hier steht, wie man ein Dreieck zeichnet:

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 100, 10, 100, 110, fill="",
outline="black")
```

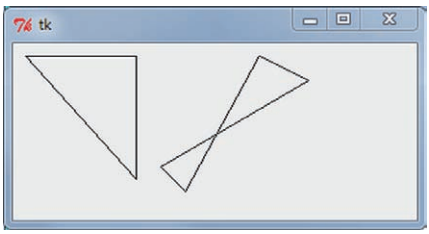
In diesem Beispiel wird ein Dreieck gezeichnet, indem mit den x-und y-Koordinaten (10, 10) begonnen wird. Dann geht es seitlich auf (100, 10) und dann bis (100,110). Hier ist das Ergebnis:



Mit folgendem Code können wir ein unregelmäßiges Polygon (eine Form mit ungleichmäßigen Winkeln oder Seiten) erzeugen:

```
canvas.create_polygon(200, 10, 240, 30, 120, 100, 140, 120, fill="",
outline="black")
```

Dieses Polygon startet bei den Koordinaten (200, 10), bewegt sich dann zu (240, 30), dann zu (120,100) und schließlich zu (100,140). Die schwarze Linie zurück zur ersten Koordinate fügt `tkinter` automatisch ein. Und hier ist das Ergebnis dieses Codes:



13.8 Darstellung von Text

Neben dem Zeichnen von Formen kannst Du mit `create_text` auch auf der Leinwand schreiben. Diese Funktion benötigt nur die zwei Koordinaten (x- und y-Positionen des Texts) und den Text, der angezeigt werden soll, als benannten Parameter. Im folgenden Code erzeugen wir wie immer unsere Leinwand und malen dann einen Satz, der sich auf den Koordinaten (150, 100) befindet. Speichere diesen Code als *Text.py*.

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_text(150, 100, text='Der Waidmann Hubertus vom Rhein')
```

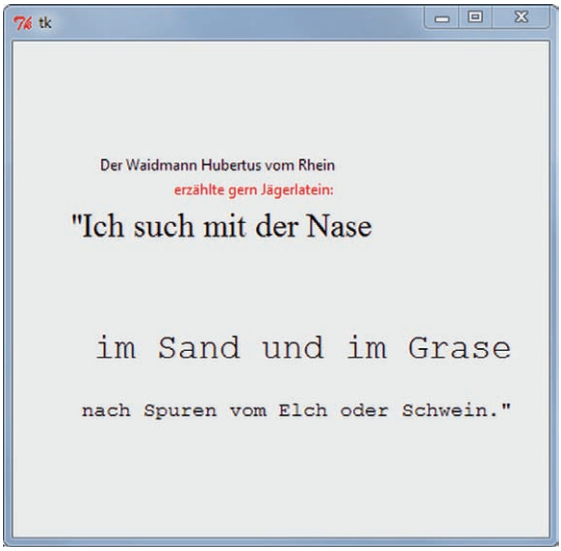
Die Funktion `create_text` nimmt aber noch weitere nützliche Parameter auf, wie etwa die Textfarbe. Im folgenden Code rufen wir die Funktion `create_text` mit den Koordinaten (180,120) auf, den Text, den wir anzeigen wollen, und die Farbe Rot.

```
canvas.create_text(180, 120, text='erzählte gern Jägerlatein:',
fill='red')
```

Du kannst auch den Font (die Schriftart des angezeigten Textes) als »Tupel« angeben, das den Font-Namen und die Größe des Textes enthält. Das Tupel für den Zeichensatz Times in der Größe 20 ist `('times', 20)`. Im folgenden Code zeigen wir den Text im Zeichensatz Times in der Größe 20 und im Zeichensatz Courier in den Größen 22 und 12 an.

```
canvas.create_text(150, 150, text= ' "Ich such mit der Nase',
font=('Times', 20))
canvas.create_text(220, 250, text= 'im Sand und im Grase',
font=('Courier', 22))
canvas.create_text(220, 300, text= 'nach Spuren vom Elch oder
Schwein." ', font=('Courier', 12))
```

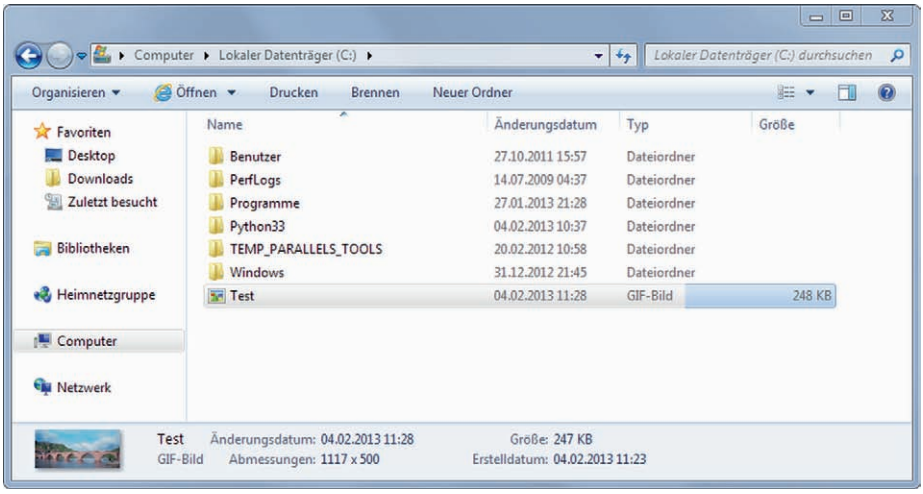
Und hier ist nun das Ergebnis der Funktionen, die verschiedene Schriften und unterschiedliche Größen verwendet haben:



13.9 Bilder anzeigen

Um ein Bild auf der Leinwand darzustellen, lädst Du erst das Bild und benutzt dann die Funktion `create_image` auf dem `canvas`-Objekt.

Alle Bilder, die Du lädst, müssen sich in einem Verzeichnis befinden, auf das Python Zugriff hat. In diesem Beispiel legen wir unser Bild *Test.gif* in das Verzeichnis C:\, also das Hauptverzeichnis unseres Laufwerks C:, aber Du kannst es auch woanders ablegen.



Wenn Du ein Mac- oder Linux-System benutzt, kannst Du das Bild in Dein Home-Verzeichnis legen. Falls Du keine Dateien auf deinem Laufwerk *C:* speichern kannst, kannst Du das Bild stattdessen auf Deinem Desktop ablegen.

Achtung!

Mit `tkinter` kannst Du nur GIF-Bilder laden, also Bild-Dateien, die mit dem Suffix *.gif* enden. Du kannst zwar auch andere Bildformate, wie etwa PNG (. png) und JPG (.jpg) verwenden, benötigst dafür allerdings ein anderes Modul, wie etwa die Python Imaging Library (*http://www.pythonware.com/products/pil/*).

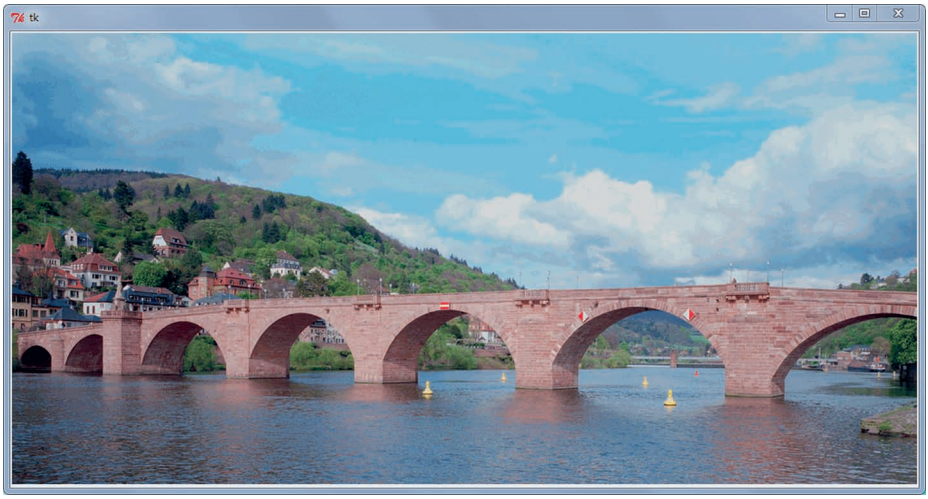
Das Bild *Test.gif* können wir folgendermaßen anzeigen:

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=1000, height=600)
canvas.pack()
mein_Bild = PhotoImage(file='c:\\Test.gif')
canvas.create_image(0, 0, anchor=NW, image=mein_Bild)
```

In den ersten Zeilen erzeugen wir wie immer die Leinwand. In der fünften Zeile wird das Bild in die Variable `mein_Bild` geladen. Im Verzeichnis `'c:\\Test. gif'` erzeugen wir `PhotoImage`. Falls Du Dein Bild auf dem Desktop abgelegt hast, solltest Du `PhotoImage` in dem entsprechenden Verzeichnis erzeugen:

```
mein_Bild = PhotoImage(file='c:\\Users\\Susanne
Sorglos\\Desktop\\Test.gif')
```

Sobald das Bild in die Variable geladen ist, zeigt `canvas.create_image (0, 0, anchor=NW, image=mein_Bild)` es mit der Funktion `create_image` an. Die Koordinaten (0, 0) geben an, wo das Bild im Fenster dargestellt wird, und `anchor=NW` sagt der Funktion, dass sie die obere linke Ecke (`NW` steht für *nordwest*) des Bildes als Startpunkt beim Aufbau verwenden soll (ansonsten benutzt sie die Bildmitte als Startpunkt). Der letzte genannte Parameter, `images`, zeigt auf die Variable für das geladene Bild. Hier siehst Du das Ergebnis:



13.10 Eine einfache Animation erzeugen

Wir wissen jetzt, wie man statische Zeichnungen erzeugt – also Bilder, die sich nicht bewegen. Wie funktioniert das nun mit den Animationen?

Animationen sind nicht gerade die Spezialität des Moduls `tkinter`, aber einfache Sachen kannst Du damit machen. Mit diesem Code können wir zum Beispiel ein ausgefülltes Dreieck über den Monitor bewegen (vergiss nicht, mit **File ▶ New Window** Deine Arbeit zu speichern und dann den Code mit **Run ▶ Run Module** laufen zu lassen):

```
import time
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=200)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
for x in range(0, 60):
    canvas.move(1, 5, 0)
    tk.update()
    time.sleep(0.05)
```

Wenn Du diesen Code ausführst, bewegt sich ein Dreieck bis zum Ende seines Wegs durch das Fenster:

Wie funktioniert das? Wie schon zuvor haben wir mit den ersten drei Zeilen nach dem Import von `tkinter` den grundsätzlichen Aufbau zur Darstellung einer Leinwand vorgenommen. In der vierten Zeile erzeugen wir mit dieser Funktion ein Dreieck:

```
canvas.create_polygon(10, 10, 10, 60, 50, 35)
```

Achtung!

Wenn Du diese Zeile eingibst, wird auf dem Monitor eine Zahl ausgegeben. Dabei handelt es sich um die identifizierende Nummer (kurz ID-Nummer) des Polygons. Wie wir im folgenden Beispiel sehen werden, können wir uns später damit auf diese Form beziehen.

Als Nächstes erzeugen wir eine einfache `for`-Schleife, die von 0 bis 59 zählt und mit `for x in range(0,60):` anfängt. Der Code-Block innerhalb der Schleife bewegt das Dreieck über den Monitor. Die Funktion `canvas.move` bewegt jedes gezeichnete Objekt, indem sie Werte zu dessen x- und y-Koordinaten hinzufügt. Mit `canvas.move(1, 5, 0)` bewegen wir das Objekt mit der ID 1 (der identifizierenden Nummer des Dreiecks) 5 Pixel zur Seite und 0 Pixel nach unten. Um es wieder zurück zu bewegen, könnten wir die Funktion `canvas.move(1, -5, 0)` aufrufen.

Die Funktion `tk.update()` zwingt `tkinter` dazu, den Monitor zu aktualisieren (also ihn erneut zu zeichnen). Ohne die Funktion `update` würde `tkinter` warten, bis die Schleife beendet ist, bevor es das Dreieck bewegt. Das Dreieck würde dann, anstatt sich sanft über die Leinwand zu bewegen, an die letzte Position springen. Die letzte Zeile der Schleife, `time sleep(0.05)`, sagt Python, dass es 0,05 Sekunden warten soll, bevor es weitermacht.

Um das Dreieck schräg über den Monitor wandern zu lassen, können wir den Code durch `move(1, 5, 5)` abändern. Um das auszuprobieren, schließt Du die

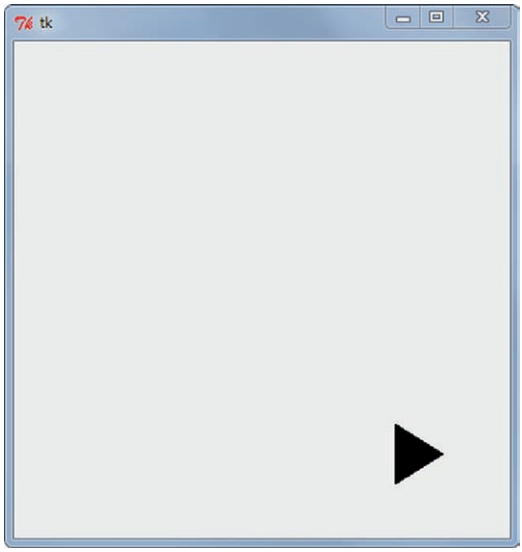
Leinwand und erzeugst eine neue Datei (**File ▶ New Window**) mit dem folgenden Code darin:

```
import time
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
for x in range(0, 60):
    canvas.move(1, 5, 5)
    tk.update()
    time.sleep(0.05)
```

Dieser Code unterscheidet sich von dem vorherigen in zwei Punkten:

- Wir ändern die Höhe der Leinwand auf 400 anstelle von 200 durch `canvas = Canvas(tk, width=400, height=400)`.
- Wir fügen den x-und y-Koordinaten durch `canvas.move(1, 5, 5)` die 5 hinzu.

Wenn der Code gespeichert und durchgelaufen ist, ist dies die Position des Dreiecks am Ende der Schleife:



Um das Dreieck auf dem gleichen Weg zurück zu seiner Startposition zu bewegen, setzt Du -5, -5 ein (füge diesen Code am Ende der Datei hinzu):

```
for x in range(0, 60):
    canvas.move(1, -5, -5)
    tk.update()
    time.sleep(0.05)
```

13.11 Ein Objekt auf etwas reagieren lassen

Durch *Ereignis-Bindungen* können wir das Dreieck reagieren lassen, wenn jemand eine Taste drückt. *Ereignisse* (engl. *events*) sind Aktionen, die geschehen, während ein Programm läuft: Jemand bewegt die Maus, drückt eine Taste oder schließt ein Fenster. Du kannst `tkinter` sagen, dass es nach solchen Ereignissen Ausschau halten und als Reaktion darauf etwas tun soll.

Um mit Ereignissen umzugehen (Python zu sagen, dass es etwas tun soll, sobald ein Ereignis eintritt), erzeugen wir als Erstes eine Funktion. Die Bindung kommt in dem Moment zustande, in dem wir `tkinter` sagen, dass es eine bestimmte Funktion mit einem bestimmten Ereignis verbinden soll. Diese Funktion wird also als Reaktion auf dieses Ereignis automatisch von `tkinter` aufgerufen.

Wenn wir zum Beispiel wollen, dass sich das Dreieck erst in Bewegung setzt, nachdem wir die Enter-Taste gedrückt haben, können wir das mit dieser Funktion definieren:

```
def movetriangle(event):
    canvas.move(1, 5, 0)
```

Die Funktion nimmt einen einzigen Parameter (`event`) auf, den `tkinter` benutzt, um der Funktion Informationen über das Ereignis zu schicken. Mit der Funktion `bind_all` sagen wir `tkinter` jetzt, dass diese Funktion für ein bestimmtes Ereignis auf der Leinwand benutzt werden soll. Der vollständige Code sieht dann so aus:

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
def movetriangle(event):
    canvas.move(1, 5, 0)
canvas.bind_all('<KeyPress-Return>', movetriangle)
```

Der erste Parameter in dieser Funktion beschreibt das Ereignis, nach dem `tkinter` Ausschau halten soll. In diesem Fall ist es `<KeyPress-Return>`, also ein Druck auf die Enter- oder die Return-Taste. Wir sagen `tkinter`, dass die Funktion `movetriangle` immer aufgerufen werden soll, sobald das `KeyPress`-Ereignis auftritt. Führe diesen Code aus, klicke einmal mit der Maus auf die Leinwand, und drücke dann die Enter-Taste auf Deiner Tastatur.

Wie wäre es, wenn wir die Richtung des Dreiecks durch unterschiedliche Tasten steuern würden, wie etwa mit den Pfeiltasten? Das geht ganz einfach. Wir müssen die Funktion `move.triangle` nur folgendermaßen ändern:

```
def movetriangle(event):
    if event.keysym == 'Up':
        canvas.move(1, 0, -3)
    elif event.keysym == 'Down':
        canvas.move(1, 0, 3)
    elif event.keysym == 'Left':
        canvas.move(1, -3, 0)
    else:
        canvas.move(1, 3, 0)
```

Das Ereignis-Objekt, das an `movetriangle` weitergereicht wird, enthält mehrere Variablen. Eine dieser Variablen heißt `keysym` (Abkürzung für engl. *key symbol*, also »Tasten-Symbol«) und ist ein String, der den Wert für die tatsächlich gedrückte Taste enthält. Die Zeile `if event.keysym == 'Up':` bedeutet: Wenn die Variable `keysym` den String `'Up'` (also »nach oben«) enthält, sollen wir `canvas.move` mit den Parametern (1, 0, -3) wie in der folgenden Zeile aufrufen. Falls `keysym` wie in `elif event.keysym == 'Down':` (also »nach unten«) enthält, rufen wir sie mit den Parametern (1, 0, 3) auf usw.

Beachte, dass der erste Parameter die ID-Nummer der Form ist, die auf die Leinwand gezeichnet wird. Der zweite Parameter ist der Wert, der zur x-Koordinate (der horizontalen Koordinate) addiert werden muss; und der dritte Parameter ist der Wert, der zur y-Koordinate (also zur vertikalen Koordinate) hinzugezählt werden muss.

Danach sagen wir `tkinter`, dass die Funktion `movetriangle` dazu benutzt werden sollte, die Ereignisse der vier verschiedenen Tasten (hoch, runter, links und rechts) zu steuern. Im Folgenden siehst Du, wie der Code bis jetzt aussieht. Beim Eingeben dieses Codes ist es viel praktischer, wenn Du Dir ein neues Shell-Fenster öffnest, indem Du auf **File ▶ New Window** gehst. Bevor Du den Code durchlaufen lässt, gibst Du ihm einen sinnvollen Dateinamen, wie zum Beispiel *animiertes_Dreieck.py*.

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
def movetriangle(event):
❶    if event.keysym == 'Up':
❷        canvas.move(1, 0, -3)
❸    elif event.keysym == 'Down':
❹        canvas.move(1, 0, 3)
```

```
❺      elif event.keysym == 'Left':
❻          canvas.move(1, -3, 0)
❼      else:
❽          canvas.move(1, 3, 0)
    canvas.bind_all('<KeyPress-Up>', movetriangle)
    canvas.bind_all('<KeyPress-Down>', movetriangle)
    canvas.bind_all('<KeyPress-Left>', movetriangle)
    canvas.bind_all('<KeyPress-Right>', movetriangle)
```

In der ersten Zeile der Funktion `movetriangle` überprüfen wir, ob die Variable `keysym` in ❶ `'Up'` enthält. Falls ja, bewegen wir das Dreieck mit der Funktion `move` und den Parametern 1, 0, -3 in ❷ nach oben. Der erste Parameter ist die ID-Nummer für das Dreieck; der zweite Parameter ist die Schrittweite nach rechts (wir möchten uns nicht nach oben bewegen, deshalb ist der Wert 0), und der dritte Parameter ist die Schrittweite nach unten (-3 Pixel).

Dann prüfen wir, ob `keysym` in ❸ `'Down'` enthält. Falls ja, bewegen wir das Dreieck in ❹ nach unten (um 3 Pixel). Zum Schluss prüfen wir, ob der Wert in ❺ `'Left'` ist, und falls dem so ist, bewegen wir das Dreieck nach links (-3 Pixel)❻. Falls keiner der Werte passt, bewegt das `else` in ❼ am Schluss das Dreieck in ❽ nach rechts.

Das Dreieck sollte sich jetzt in die Richtung der entsprechenden Pfeiltaste bewegen.

13.12 Weitere Anwendungen für die ID-Nummer

Immer wenn wir eine Funktion mit `create_` auf der Leinwand verwenden (wie etwa `create_polygon` oder `create_rectangle`), wird eine ID-Nummer zurückgegeben. Diese ID-Nummer kann mit anderen `canvas`-Funktionen verwendet werden, wie wir es zuvor mit der Funktion `move` getan haben:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_polygon(10, 10, 10, 60, 50, 35)
1
>>> canvas.move(1, 5, 0)
```

Das Problem an diesem Beispiel ist, dass `create_polygon` nicht in jedem Fall 1 zurückgibt. Falls Du etwa zuvor andere Formen erzeugt hast, kann sie auch 2, 3 oder sogar 100 ausgeben (je nachdem, wie viele Formen Du davor erzeugt hast). Wenn wir den Code so ändern, dass der Wert als Variable zurückkommt und danach die Variable verwenden (anstatt auf die Nummer 1 zu verweisen), dann funktioniert der Code immer, ganz egal welche Nummer zurückgegeben wird:

```
>>> meinDreieck = canvas.create_polygon(10, 10, 10, 60, 50, 35)
>>> canvas.move(meinDreieck, 5, 0)
```

Die Funktion `move` ermöglicht es uns, Objekte anhand ihrer ID-Nummer über den Monitor zu bewegen. Aber es gibt noch weitere Leinwand-Funktionen, die etwas ändern können, das wir gezeichnet haben. Mit der Funktion der Leinwand `itemconfig` kann man einige der Parameter einer Form, wie etwa die Füll- und die Umrissfarbe, ändern:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> meinDreieck = canvas.create_polygon(10, 10, 10, 60, 50, 35,
fill='red')
```

Wir können mit `itemconfig` dem Dreieck eine andere Farbe geben und die ID-Nummer als ersten Parameter benutzen. Der folgende Code bedeutet: »Ändere die Füllfarbe des Objekts in diejenige, die in der ID-Nummer der Variablen `meinDreieck` steht, also in Blau«.

```
>>> canvas.itemconfig(meinDreieck, fill= 'blue')
```

Wir können den Umriss des Dreiecks auch anders einfärben, indem wir die ID-Nummer als ersten Parameter wählen:

```
>>> canvas.itemconfig(meinDreieck, outline= 'red')
```

Später lernst Du, wie man weitere Änderungen an einer Zeichnung vornimmt – Du kannst sie zum Beispiel verschwinden lassen und wieder sichtbar machen. Wenn wir anfangen, im nächsten Kapitel Spiele zu schreiben, wirst Du merken, wie nützlich es ist, eine Zeichnung, die schon auf dem Monitor angezeigt wird, noch ändern zu können.

13.13 Was Du gelernt hast

In diesem Kapitel hast Du das Modul `tkinter` benutzt, um damit einfache geometrische Formen auf die Leinwand zu zeichnen, Bilder anzuzeigen und einfache Animationen auszuführen. Du hast gelernt, wie man mit Ereignis-Bindungen Zeichnungen auf so etwas wie einen Tastendruck reagieren lässt. Sobald wir an der Programmierung eines Spiels arbeiten werden, wird das sehr nützlich sein. Du hast gelernt, dass die `create`-Funktion ID-Nummern zurückgibt, womit man For-

men verändern kann, nachdem sie schon gezeichnet worden sind. So kannst Du die Formen beispielsweise auf dem Monitor verschieben oder ihre Farbe ändern.

13.14 Programmier-Puzzles

Probiere die folgenden Dinge aus, um mit dem Modul `tkinter` und einfachen Animationen zu spielen. Lösungen findest Du unter *www.dpunkt.de/python*.

#1: Fülle die Leinwand mit Dreiecken

Schreibe ein Programm mit `tkinter`, um die Leinwand mit Dreiecken zu füllen. Ändere dann den Code, um stattdessen die Leinwand mit unterschiedlich gefärbten (ausgefüllten) Dreiecken zu füllen.

#2: Das sich bewegende Dreieck

Ändere den Code des sich bewegenden Dreiecks (siehe den Abschnitt 13.10) so, dass es sich auf der Leinwand erst nach rechts, dann nach unten, dann nach links und zum Schluss wieder nach oben auf seine Startposition bewegt.

#3: Das sich bewegende Foto

Versuche selbst, mit `tkinter` ein Foto auf der Leinwand darzustellen. Achte darauf, dass es ein GIF-Bild ist! Kannst Du es dazu bringen, sich über die Leinwand zu bewegen?

Teil II

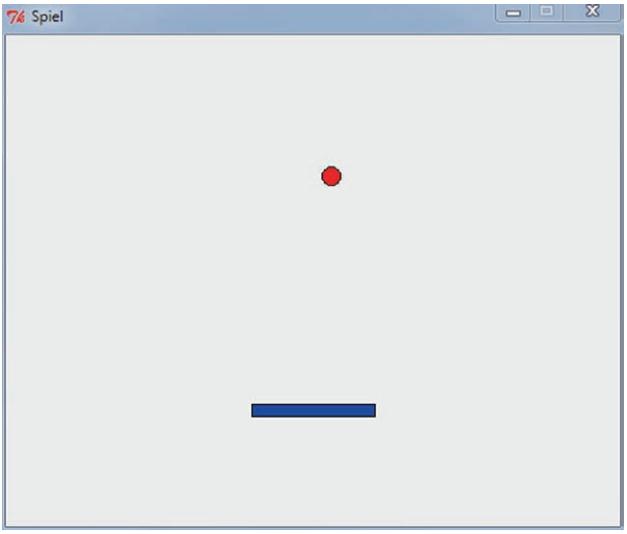
BOUNCE!

14
Der Anfang Deines ersten Spiels: BOUNCE!

Bis jetzt haben wir uns mit den Grundlagen des Computer-Programmierens befasst. Du hast gelernt, wie man mit Variablen Informationen speichert, mit `if`-Anweisungen bedingten Code einsetzt und wie man mit `for`-Schleifen Code wiederholt. Du weißt, wie man Funktionen erzeugt, um den Code wiederzuverwenden und wie man Klassen und Objekte einsetzt, um den Code in kleinere Einheiten zu unterteilen, damit er leichter zu verstehen ist. Du hast gelernt, wie man mit den Modulen `turtle` und `tkinter` Grafiken auf dem Monitor zeichnet. Jetzt bist Du so weit, dieses Wissen bei Deinem ersten selbst geschriebenen Spiel einzusetzen.

14.1 Schlag den hüpfenden Ball

Wir werden ein Spiel mit einem herumspringenden Ball und einem Schläger entwerfen. Der Ball wird über den Bildschirm fliegen, und der Spieler wird ihn vom Schläger abprallen lassen. Wenn der Ball die untere Kante des Bildschirms berührt, ist das Spiel zu Ende. Hier siehst Du eine Voransicht des fertigen Spiels:



Unser Spiel mag zwar sehr einfach erscheinen, aber der Code wird um einiges kniffeliger, als alles, was wir bis jetzt geschrieben haben, da er eine Menge von Dingen bewältigen muss. Er muss zum Beispiel den Schläger und den Ball animieren und erkennen, wann der Ball den Schläger oder die Wände berührt.

In diesem Kapitel beginnen wir mit der Entwicklung unseres Spiels, indem wir eine Spielfläche und einen hüpfenden Ball hinzufügen.

14.2 Erzeugen einer Spiele-Leinwand

Für Dein neues Spiel legst Du als Erstes eine neue Datei in der Python-Shell an (über **File ▶ New Window**). Dann importierst Du `tkinter` und erzeugst eine Leinwand, um darauf zu zeichnen:

```
from tkinter import *
import random
import time
tk = Tk()
tk.title("Spiel")
tk.resizable(0, 0)
tk.wm_attributes("-topmost", 1)
canvas = Canvas(tk, width=500, height=400, bd=0, highlightthickness=0)
canvas.pack()
tk.update()
```

Dies ist jetzt ein bisschen anders als bei den vorherigen Beispielen. Zu Anfang importieren wir die Module `time` und `random`, um sie weiter hinten im Code einzusetzen.

Mit `tk.title("Spiel")` verwenden wir die Funktion `title` des Objekts `tk`, das wir mit `tk = Tk()` benutzen, um dem Fenster einen Titel zu geben. Um dem Fenster eine feste Größe zu geben, verwenden wir `resizable`. Die Parameter `0, 0` bedeuten: »Die Größe dieses Fensters kann weder horizontal noch vertikal verändert werden.« Als Nächstes rufen wir `wm_attributes` auf, damit `tkinter` das Fenster mit unserer Leinwand vor alle anderen Fenster stellt (`"-topmost"`).

Du siehst, dass wir bei der Erzeugung des `canvas`-Objekts mit `canvas =` mehr benannte Parameter auflisten, als wir es bei den vorigen Beispielen getan haben. Sowohl `bd=0` als auch `highlightthickness=0` sorgen beispielsweise dafür, dass es keinen Rand außen um die Leinwand gibt, damit unser Spielfeld besser aussieht.

Die Zeile `canvas.pack()` sagt der Leinwand, dass sie ihre Größe anhand der Parameter `width` und `height` in der vorherigen Zeile anpassen soll. Zum Schluss wird `tkinter` durch `tk.update()` angewiesen, sich selbst für die Animation in unserem Spiel zu initialisieren. Ohne diese letzte Zeile würde nichts wie erwartet funktionieren.

Achte darauf, dass Du Deinen Code immer speicherst. Gib ihm beim ersten Speichern einen sinnvollen Dateinamen, wie zum Beispiel *Bounce.py*.

14.3 Erzeugen der Ball-Klasse

Jetzt werden wir die Klasse für den Ball erzeugen. Wir werden mit dem Code beginnen, den wir brauchen, um den Ball auf der Leinwand zu zeichnen. Folgende Schritte sind dazu nötig:

- Erzeuge eine Klasse namens `Ball`, die Parameter für die Leinwand sowie die Farbe des Balls aufnimmt, den wir zeichnen werden.
- Speichere die Leinwand als Objekt-Variable, da wir unseren Ball darauf zeichnen werden.
- Zeichne einen ausgefüllten Kreis auf die Leinwand, und benutze den Wert des Farbe-Parameters als Füllfarbe.
- Speichere die ID-Nummer, die `tkinter` zurückgibt, sobald es den Kreis (Oval) zeichnet. Wir brauchen sie, um den Ball auf dem Bildschirm zu bewegen.
- Bewege das Oval auf die Mitte der Leinwand.

Dieser Code sollte gleich nach den ersten beiden Zeilen in der Datei (nach `import time`) hinzugefügt werden:

```
from tkinter import *
import random
import time

❶ class Ball:
❷     def __init__(self, canvas, color):
❸         self.canvas = canvas
❹         self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
❺         self.canvas.move(self.id, 245, 100)

      def draw(self):
          pass
```

Als Erstes benennen wir unsere Klasse Ball in ❶. Danach erzeugen wir unsere Initialisierungsfunktion (wie in Kapitel 9 beschrieben), die die Parameter `canvas` und `color` in ❷ aufnimmt. In ❸ setzen wir die Objekt-Variable `canvas` auf den Wert des Parameters `canvas`.

In ❹ rufen wir die Funktion `create_oval` mit ihren fünf Parametern auf: Das sind die x- und y-Koordinaten für die Ecke oben links (10 und 10), die x- und y-Koordinaten für die Ecke unten rechts (25 und 25) und schließlich die Farbe des Ovals.

Die Funktion `create_oval` gibt eine ID-Nummer für die Figur zurück, die sie gezeichnet hat. Wir speichern diese Nummer in der Objekt-Variablen `id`. In ❺ bewegen wir das Oval in die Mitte der Leinwand (Position 245, 100). Die Leinwand weiß, was sie zu bewegen hat, da wir die ID-Nummer dieser Figur (die Objekt-Variable `id`) benutzt haben, um sie zu kennzeichnen.

In den letzten beiden Zeilen der Klasse `Ball` erzeugen wir mit `def draw(self)` die Funktion `draw`, deren Funktionskörper im Moment nur aus dem Schlüsselwort `pass` besteht. Wir fügen dieser Funktion bald mehr hinzu.

Nachdem wir jetzt unsere Ball-Klasse erzeugt haben, müssen wir aus dieser Klasse ein Objekt erzeugen. (Du erinnerst Dich sicher, dass eine Klasse zwar beschreibt, was sie tun kann, dass es aber das Objekt ist, das wirklich etwas tut). Um ein rotes Ball-Objekt zu erzeugen, fügst Du dem Ende des Programms folgenden Code hinzu:

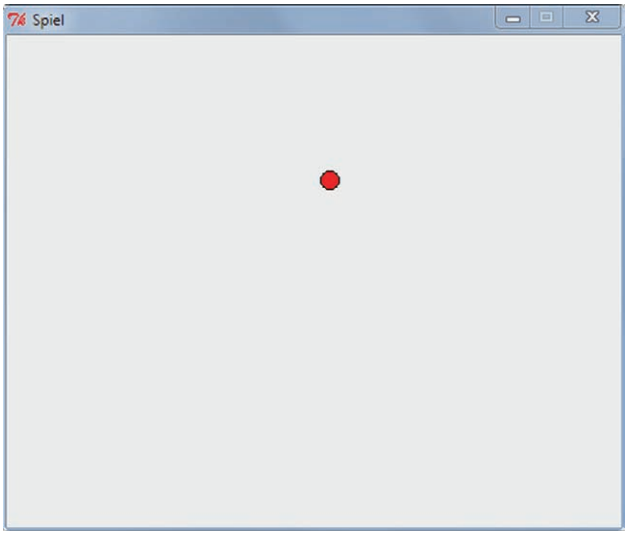
```
ball = Ball(canvas, 'red')
```

Wenn Du dieses Programm jetzt mit **Run ▶ Run Module** laufen lässt, erscheint die Leinwand für den Bruchteil einer Sekunde und verschwindet dann wieder. Damit das Fenster sich nicht sofort wieder schließt, müssen wir eine Animationsschleife hinzufügen, die sogenannte *Hauptschleife* unseres Spiels.

Die Hauptschleife ist der zentrale Bestandteil eines Programms, der in der Regel die meisten Dinge kontrolliert. Im Moment befiehlt unsere Hauptschleife `tkinter` lediglich, das Bild neu aufzubauen. Diese Schleife läuft ständig durch (zumindest so lange, bis wir das Fenster schließen). Sie weist `tkinter` fortlaufend an, das Bild neu aufzubauen und dann für eine Hundertstelsekunde zu pausieren. Diesen Code fügen wir am Ende unseres Programms an:

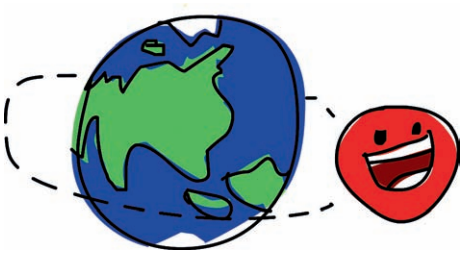
```
ball = Ball(canvas, 'red')

while 1:
  tk.update_idletasks()
  tk.update()
  time.sleep(0.01)
```

Wenn Du nun diesen Code durchlaufen lässt, sollte der Ball ziemlich in der Mitte der Leinwand erscheinen:



14.4 In Bewegung kommen

Eben haben wir die `Ball`-Klasse eingerichtet, und jetzt wird es Zeit, den Ball zu animieren. Wir werden ihn dazu bringen, sich zu bewegen, abzuprallen und seine Richtung zu ändern.



Den Ball in Bewegung setzen

Damit sich der Ball bewegt, ändern wir die Funktion `draw` wie folgt:

```
class Ball:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)

    def draw(self):
        self.canvas.move(self.id, 0, -1)
```

Da `__init__` den `canvas`-Parameter als Objekt-Variable `canvas` gespeichert hat, benutzen wir diese Variable mit `self.canvas` und rufen die Funktion `move` auf der Leinwand auf.

Wir reichen drei Parameter an `move` weiter: die `id` des Ovals und die Zahlen 0 und -1. Die 0 sagt, dass sich der Ball nicht horizontal bewegen soll, und die -1 bedeutet, dass er sich einen Pixel vertikal nach oben bewegen soll.

Wir nehmen diese kleine Änderung vor, weil es gut ist, Dinge zwischendurch erst auszuprobieren. Stell Dir vor, wir würden den gesamten Code unseres Spiels auf einmal schreiben und dann feststellen, dass er nicht funktioniert. Wie sollten wir dann jemals herausfinden, warum er streikt?

Die nächste Änderung betrifft unsere Hauptschleife ganz unten in unserem Programm. Im Block mit unserer `while`-Schleife (dies ist unsere Hauptschleife!) fügen wir den Aufruf der Funktion `draw` unseres Ball-Objekts hinzu:

```
while 1:
    ball.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Wenn Du diesen Code jetzt laufen lässt, sollte der Ball sich die Leinwand hinauf bewegen und dann aus ihr verschwinden, da der Code `tkinter` dazu zwingt, den Monitor schnell wieder aufzubauen – es sind die Befehle `update_idletasks` und `update`, die `tkinter` anweisen, sich zu beeilen und zu zeichnen, was auf der Leinwand ist.

Der Befehl `time.sleep` ruft die Funktion `sleep` aus dem Modul `time` auf, wodurch Python für eine Hundertstelsekunde (0.01) schläft. Dadurch wird dafür gesorgt, dass unser Programm nicht so schnell läuft, dass der Ball verschwindet, bevor Du ihn überhaupt sehen konntest.

Die Schleife sagt also im Grunde: »Bewege den Ball ein bisschen, baue den Bildschirm mit der neuen Position auf, warte ein bisschen, und fange dann von vorne an.«

Achtung!

Wenn Du das Spiele-Fenster schließt, kann es sein, dass Du in der Shell Fehlermeldungen siehst. Die tauchen auf, weil durch das Schließen des Fensters der Code aus der `while`-Schleife ausbricht und Python sich darüber beschwert.

Dein Spiel sollte bis jetzt folgendermaßen aussehen:

```
from tkinter import *
import random
import time

class Ball:
  def __init__(self, canvas, color):
     self.canvas = canvas
     self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
     self.canvas.move(self.id, 245, 100)
  def draw(self):
     self.canvas.move(self.id, 0, -1)

tk = Tk()
tk.title("Spiel")
tk.resizable(0, 0)
tk.wm_attributes("-topmost", 1)
canvas = Canvas(tk, width=500, height=400, bd=0, highlightthickness=0)
canvas.pack()
tk.update()

ball = Ball(canvas, 'red')

while 1:
  ball.draw()
  tk.update_idletasks()
  tk.update()
  time.sleep(0.01)
```

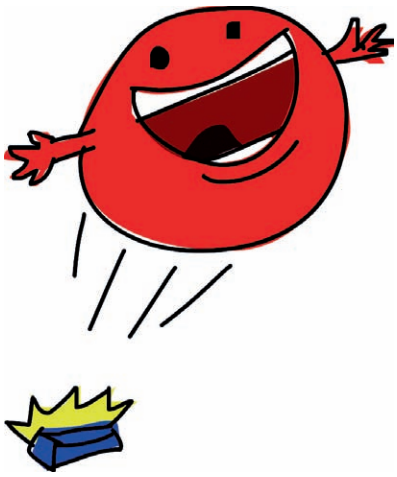
Den Ball springen lassen

Ein Ball, der einfach oben aus dem Bildschirm verschwindet, ist in einem Spiel nicht sonderlich hilfreich. Wir sollten ihn deswegen zurückspringen lassen. Als Erstes speichern wir ein paar zusätzliche Objekt-Variablen in der Initialisierungsfunktion der `Ball`-Klasse:

```
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)
    self.x = 0
    self.y = -1
    self.canvas_height = self.canvas.winfo_height()
```

Wir haben unserem Programm drei weitere Zeilen hinzugefügt. Wir haben mit `self.x = 0` die Objekt-Variable `x` auf 0 gesetzt und dann mit `self.y = -1` die Variable `y` auf -1 gesetzt. Zum Schluss haben wir die Objekt-Variable `canvas_height` durch den Aufruf der `canvas`-Funktion `winfo_height` gesetzt. Diese Funktion gibt die aktuelle Höhe der Leinwand zurück. Als Nächstes ändern wir wieder die Funktion `draw`:

```
    def draw(self):
❶       self.canvas.move(self.id, self.x, self.y)
❷       pos = self.canvas.coords(self.id)
❸       if pos[1] <= 0:
            self.y = 1
❹       if pos[3] >= self.canvas_height:
            self.y = -1
```



In ❶ ändern wir den Aufruf der `canvas`-Funktion `move`, indem wir ihr die Objekt-Variablen `x` und `y` übergeben. Als Nächstes erzeugen wir in ❷ die Variable `pos`, indem wir die `canvas`-Funktion `coords` aufrufen. Diese Funktion gibt die aktuellen `x`- und `y`-Koordinaten von allem zurück, was auf der Leinwand gezeichnet wird, solange Du die ID-Nummern kennst. In diesem Fall weisen wir `coords` die Objekt-Variable `id` zu, die die ID-Nummer des Ovals enthält.

Die Funktion `coords` gibt die Koordinaten in Form einer Liste von vier Zahlen zurück. Wenn wir uns die Ergebnisse des Aufrufs dieser Funktion anzeigen lassen wollen, sehen wir so etwas:

```
print(self.canvas.coords(self.id))
[255.0, 29.0, 270.0, 44.0]
```

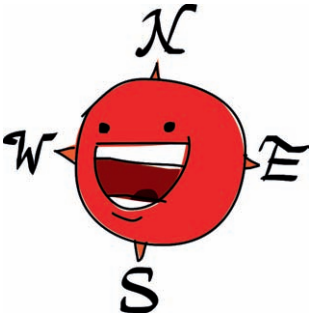
Die ersten beiden Zahlen dieser Liste (255.0 und 29.0) enthalten die oberen linken Koordinaten des Ovals (*x1* und *y1*); Die nächsten beiden (270.0 144.0) sind die Koordinaten *x2* und *y2* unten rechts. Wir werden diese Werte in den nächsten paar Zeilen des Codes verwenden.

In ❸ sehen wir, ob die *y1*-Koordinate (das ist die obere Kante des Balls) weniger oder gleich 0 ist. Falls ja, setzen wir die y-Objekt-Variable auf 1. Dadurch sagen wir: »Sobald Du oben anstößt, höre damit auf, 1 von der vertikalen Position abzuziehen, und stoppe dadurch die Bewegung nach oben.«

In ❹ sehen wir, ob die y2-Koordinate (das ist die untere Kante des Balls) größer oder gleich der Variable `canvas_height` ist. Falls sie es ist, setzen wir die y-Objekt-Variable auf -1 zurück.

Wenn Du diesen Code jetzt ausführst, sollte der Ball auf der Leinwand so lange auf und ab springen, bist Du das Fenster schließt.

Die Startposition des Balls ändern



Einen Ball langsam auf- und abspringen zu lassen, ist noch lange kein Spiel. Lass uns daher die Startposition des Balls ändern – den Winkel, in dem der Ball wegfliegt, wenn das Spiel startet. Ändere in der Funktion `__init__` die Zeilen

```
self.x = 0
self.y = -1
```

in Folgendes (achte darauf, dass Du die richtige Anzahl von Leerzeichen – hier sind es acht – vor jeder Zeile hast):

```
❶        starts = [-3, -2, -1, 1, 2, 3]
❷        random.shuffle(starts)
❸        self.x = starts[0]
❹        self.y = -3
```

In ❶ erzeugen wir die Variable `starts` mit einer Liste von sechs Zahlen. Diese Zahlen mischen wir in ❷, indem wir `random.shuffle` aufrufen. In ❸ setzen wir den Wert von x auf das erste Element dieser Liste, sodass x jede Zahl dieser Liste sein kann (zwischen -3 und 3).

Wenn wir y in ❹ auf -3 setzen (um den Ball schneller zu machen), müssen wir noch ein paar weitere Änderungen vornehmen, damit der Ball nicht einfach aus dem Fenster verschwindet. Füge folgende Zeile am Ende der Funktion `__init__` hinzu, damit die Breite der Leinwand in eine neue Objekt-Variable, `canvas_width`, gespeichert wird:

```
        self.canvas_width = self.canvas.winfo_width()
```

Wir verwenden diese neue Objekt-Variable in der Funktion `draw`, um zu sehen, ob der Ball oben oder unten gegen die Leinwand gestoßen ist:

```
        if pos[0] <= 0:
            self.x = 3
        if pos[2] >= self.canvas_width:
            self.x = -3
```

Da wir `x` auf 3 und -3 gesetzt haben, tun wir das Gleiche mit `Y`, damit sich der Ball in alle Richtungen mit der gleichen Geschwindigkeit bewegt. Deine `draw`-Funktion sollte jetzt folgendermaßen aussehen:

```
        def draw(self):
            self.canvas.move(self.id, self.x, self.y)
            pos = self.canvas.coords(self.id)
            if pos[1] <= 0:
                self.y = 3
            if pos[3] >= self.canvas_height:
                self.y = -3
            if pos[0] <= 0:
                self.x = 3
            if pos[2] >= self.canvas_width:
                self.x = -3
```

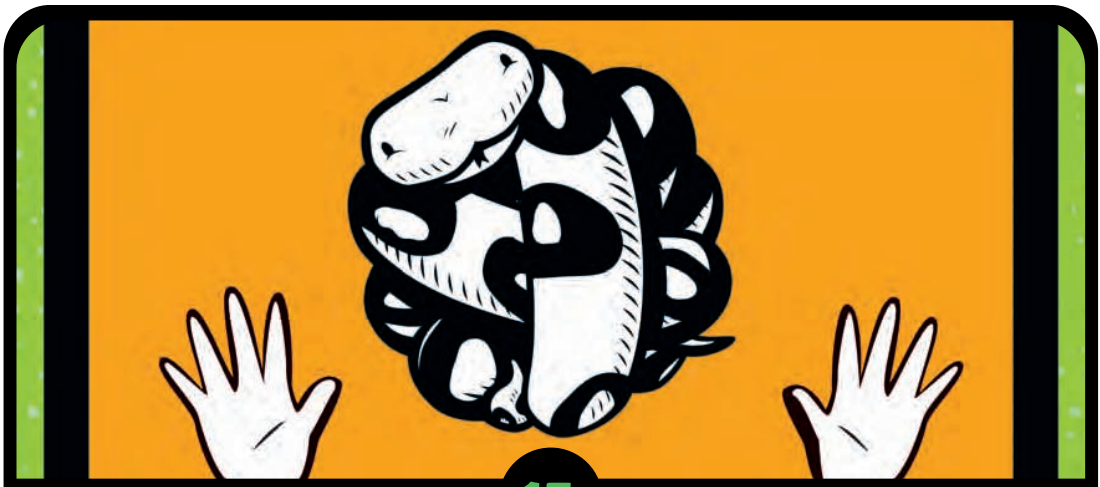
Speichere das Programm und starte es. Der Ball sollte jetzt auf dem Bildschirm herumspringen, ohne zu verschwinden. Das komplette Programm sieht nun so aus:

```
from tkinter import *
import random
import time

class Ball:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)
        starts = [-3, -2, -1, 1, 2, 3]
        random.shuffle(starts)
        self.x = starts[0]
        self.y = -3
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()
```

```
        def draw(self):
            self.canvas.move(self.id, self.x, self.y)
            pos = self.canvas.coords(self.id)
            if pos[1] <= 0:
                self.y = 1
            if pos[3] >= self.canvas_height:
                self.y = -1
            if pos[0] <= 0:
                self.x = 3
            if pos[2] >= self.canvas_width:
                self.x = -3

tk = Tk()
tk.title("Spiel")
tk.resizable(0, 0)
tk.wm_attributes("-topmost", 1)
canvas = Canvas(tk, width=500, height=400, bd=0, highlightthickness=0)
canvas.pack()
tk.update()

ball = Ball(canvas, 'red')

while 1:
    ball.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

14.5 Was Du gelernt hast

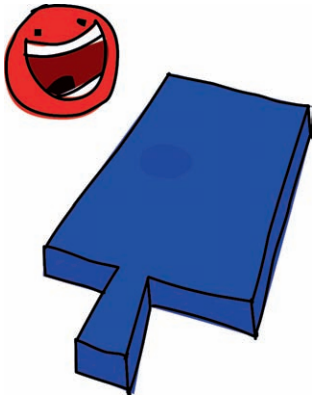
In diesem Kapitel haben wir mit dem Modul `tkinter` unser erstes Spiel geschrieben. Wir haben eine Klasse für einen Ball erzeugt, den wir so animiert haben, dass er sich über den Monitor bewegt. Wir haben durch Koordinaten geprüft, wann der Ball die Wände der Leinwand berührt, damit wir ihn abprallen lassen können. Im Modul `random` haben wir dazu noch die Funktion `shuffle` benutzt, damit unser Ball nicht immer von der gleichen Position aus startet. Im nächsten Kapitel werden wir das Spiel fertigstellen, indem wir einen Schläger hinzufügen.



15
Dein erstes Spiel vollenden: BOUNCE!

Im vorigen Kapitel haben wir angefangen, unser erstes Spiel zu programmieren: BOUNCE! Wir haben eine Leinwand erzeugt und unserem Spiel-Code einen hüpfenden Ball hinzugefügt. Unser Ball würde jedoch bis in alle Ewigkeiten auf dem Monitor herumhüpfen (oder zumindest so lange, bist Du den Computer ausschaltest), was bei einem Spiel nicht sehr sinnvoll ist. Jetzt werden wir dem Spieler einen Schläger geben, den er benutzen kann. Wir werden dem Spiel auch ein gewisses Zufallselement geben, damit es ein bisschen schwieriger und spannender zu spielen ist.

15.1 Einen Schläger hinzufügen



Ein hüpfender Ball macht nicht viel Spaß, wenn man ihn nicht mit irgendetwas schlagen kann. Zeit für einen richtigen Schläger!

Wir beginnen mit dem Hinzufügen des folgenden Codes gleich hinter der `Ball`-Klasse, um einen Schläger zu erzeugen. Dazu fügst Du den Code in eine neue Zeile unter der Funktion `draw` in der Klasse `Ball` ein:

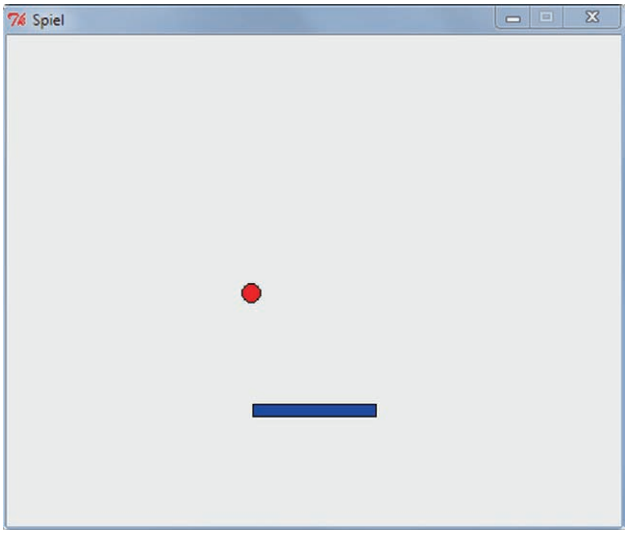
```
        def draw(self):
            self.canvas.move(self.id, self.x, self.y)
            pos = self.canvas.coords(self.id)
            if pos[1] <= 0:
                self.y = 1
            if pos[3] >= self.canvas_height:
                self.y = -1
            if pos[0] <= 0:
                self.x = 3
            if pos[2] >= self.canvas_width:
                self.x = -3

class Schläger:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_rectangle(0, 0, 100, 10, fill=color)
        self.canvas.move(self.id, 200, 300)
    def draw(self):
        pass
```

Dieser hinzugefügte Code ist fast exakt der gleiche wie bei der `Ball`-Klasse, nur dass wir `create_rectangle` (statt `create_oval`) aufrufen und dass wir das Rechteck (engl. *rectangle*) auf Position 200, 300 (200 Pixel zur Seite und 300 Pixel nach unten) bewegen.

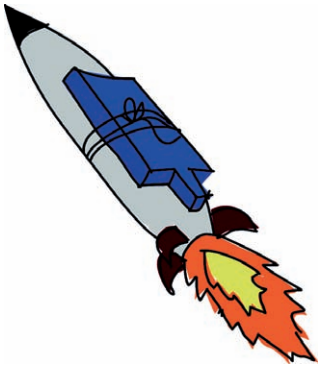
Als Nächstes erzeugst Du ganz unten in Deinem Listing in der Klasse `Schläger` ein Objekt und änderst dann die Hauptschleife, um die Funktion `draw` in `Schläger` aufzurufen:

```
schläger = Schläger(canvas, 'blue')
ball = Ball(canvas, 'red')

while 1:
    ball.draw()
    schläger.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Wenn Du das Spiel jetzt laufen lässt, solltest Du einen springenden Ball und einen rechteckigen Schläger sehen, der sich nicht bewegt:



Den Schläger in Bewegung setzen



Um den Schläger nach links und rechts zu bewegen, werden wir Ereignis-Bindungen einsetzen, damit die linke und rechte Pfeiltaste mit neuen Funktionen in der `Schläger`-Klasse verbunden wird. Wenn der Spieler die linke Pfeiltaste drückt, wird die `x`-Variable auf -2 gesetzt (um sich nach links zu bewegen). Das Drücken der rechten Pfeiltaste setzt die `x`-Variable auf 2 (für die Bewegung nach rechts).

Der erste Schritt besteht also darin, die `x`-Objekt-Variable der Funktion `__init__` zu unserer Schläger-Klasse hinzuzufügen. Dazu kommt noch eine Variable für die Breite der Leinwand (wie schon in der `Ball`-Klasse):

```
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_rectangle(0, 0, 100, 10, fill=color)
        self.canvas.move(self.id, 200, 300)
        self.x = 0
        self.canvas_width = self.canvas.winfo_width()
```

Nun benötigen wir die Funktionen zum Wechsel der Richtungen zwischen links (`nach_links`) und rechts (`nach_rechts`). Diese fügen wir gleich hinter der Funktion `draw` ein:

```
    def nach_links(self, evt):
        self.x = -2

    def nach_rechts(self, evt):
        self.x = 2
```

Mit diesen beiden Programmzeilen verbinden wir unsere Funktionen zum Richtungswechsel mit den passenden Tasten. Dazu nutzen wir die Funktion `__init__` in dieser Klasse. Auf Seite 176 haben wir im Abschnitt »Ein Objekt auf etwas reagieren lassen« Python eine Funktion aufrufen lassen, sobald eine Taste gedrückt wurde. Jetzt verbinden wir die Funktion `nach_links` in unserer `Schläger`-Klasse mit der linken Pfeiltaste mit dem Ereignisnamen `'<KeyPress-Left>'`. Anschließend verbinden wir die `nach_rechts`-Funktion mit der rechten Pfeiltaste mit dem Ereignisnamen `'<KeyPress-Right>'`. Unsere Funktion `__init__` sieht jetzt folgendermaßen aus:

```
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_rectangle(0, 0, 100, 10, fill=color)
        self.canvas.move(self.id, 200, 300)
        self.x = 0
        self.canvas_width = self.canvas.winfo_width()
        self.canvas.bind_all('<KeyPress-Left>', self.nach_links)
        self.canvas.bind_all('<KeyPress-Right>', self.nach_rechts)
```

Die Funktion `draw` ist für die Schläger-Klasse so ähnlich wie die in der `Ball`-Klasse:

```
    def draw(self):
        self.canvas.move(self.id, self.x, 0)
        pos = self.canvas.coords(self.id)
        if pos[0] <= 0:
            self.x = 0
        elif pos[2] >= self.canvas_width:
            self.x = 0
```

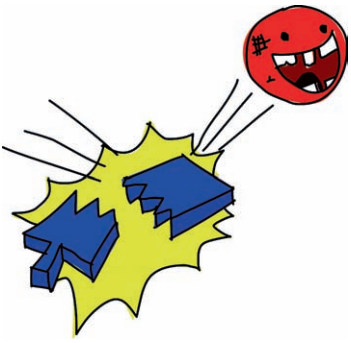
Wir benutzen die `move`-Funktion der Leinwand, um den Schläger in Richtung der `x`-Variablen mit `self.canvas.move(self.id, self.x, 0)` zu bewegen. Danach bekommen wir die Koordinaten des Schlägers, um mit dem Wert in `pos` zu sehen, ob er die linke oder rechte Seite des Fensters berührt hat.

Anstatt wie ein Ball dann einfach abzuprallen, sollte der Schläger bei Kontakt mit dem Fenster anhalten. Wenn die linke `x`-Koordinate (`pos[0]`) weniger oder gleich 0 ist (`<= 0`) setzen wir die `x`-Variable mit `self. x = 0` auf 0. Und wenn die rechte `x`-Koordinate (`pos[2]`) größer oder gleich der Breite der Leinwand ist (`>= self.canvas_width`), dann setzen wir auch die `x`-Variable mit `self.x = 0` auf 0.

Achtung!

Wenn Du das Programm jetzt laufen lässt, musst Du zuerst auf die Leinwand klicken, damit das Spiel die Aktionen der linken und rechten Pfeiltaste erkennt. Durch das Klicken auf die Leinwand weiß sie, dass sie reagieren muss, sobald jemand eine Taste auf der Tastatur drückt.

15.2 Merken, dass der Ball auf den Schläger trifft



Zum jetzigen Zeitpunkt trifft der Ball nicht auf den Schläger, sondern fliegt einfach durch ihn hindurch. Der Ball muss wissen, dass er auf den Schläger trifft, genau wie der Ball wissen muss, dass er an die Wand prallt.

Wir hätten dieses Problem dadurch lösen können, dass wir der Funktion `draw` Code hinzufügen (an der Stelle, wo der Code nach Wänden sucht). Aber es ist klüger, solchen Code in eine neue Funktion zu legen, damit die Dinge besser unterteilt sind. Wenn wir zu viel Code an einer Stelle ansammeln (beispielsweise in einer einzigen Funktion), wird der Code viel schwieriger zu verstehen. Lass uns die nötigen Änderungen vornehmen.

Als Erstes ändern wir die Funktion `__init__` des Balls, sodass wir das Schläger-Objekt als Parameter einführen können:

```
    class Ball:
❶       def __init__(self, canvas, color):
            self.canvas = canvas
❷           self.Schläger = Schläger
            self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
            self.canvas.move(self.id, 245, 100)
            starts = [-3, -2, -1, 1, 2, 3]
            random.shuffle(starts)
            self.x = starts[0]
            self.y = -3
            self.canvas_height = self.canvas.winfo_height()
            self.canvas_width = self.canvas.winfo_width()
```

In ❶ haben wir den Parameter von `__init__` so geändert, dass er den Schläger mit einschließt. In ❷ haben wir dann den `Schläger`-Parameter der Objekt-Variablen `Schläger` zugewiesen.

Nachdem wir das `Schläger`-Objekt gespeichert haben, müssen wir noch den Code an der Stelle ändern, an der wir das `Ball`-Objekt erzeugt haben. Diese Änderung wird ganz unten im Programm vorgenommen, kurz vor der Hauptschleife:

```
schläger = Schläger(canvas, 'blue')
ball = Ball(canvas, Schläger, 'red')

while 1:
    ball.draw()
    schläger.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Den Code, den wir brauchen, um zu sehen, ob der Ball den Schläger berührt hat, ist ein wenig komplizierter, als der Code, den man braucht, um zu prüfen, ob die Wände berührt wurden. Wir werden diese Funktion `triff_schläger` nennen und sie der `draw`-Funktion in der `Ball`-Klasse hinzufügen, wo wir sehen, ob der Ball den Boden des Fensters berührt hat:

```
    def draw(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[1] <= 0:
            self.y = 1
        if pos[3] >= self.canvas_height:
            self.y = -1
        if self.triff_schläger(pos) == True:
            self.y = -3
        if pos[0] <= 0:
            self.x = 3
        if pos[2] >= self.canvas_width:
            self.x = -3
```

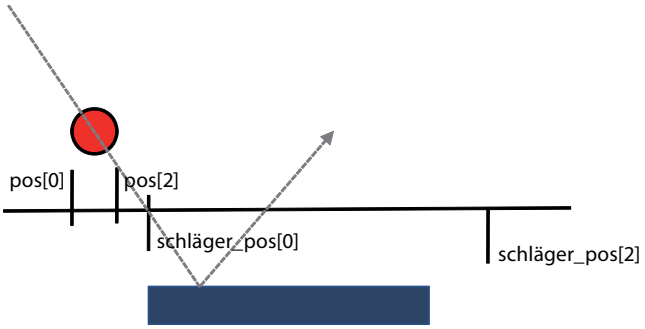
Wie Du im neu hinzugefügten Code erkennen kannst, ändern wir – falls `triff_schläger` wahr (`True`) zurückgibt – die Flugrichtung des Balls, indem wir die `y`-Objekt-Variable mit `self.y = -3` auf -3 setzen. Versuche jetzt aber noch nicht, das Spiel laufen zu lassen, denn wir haben die Funktion `triff_schläger` noch nicht definiert! Das machen wir erst jetzt.

Füge die Funktion `triff_schläger` genau vor die `draw`-Funktion ein:

```
    def triff_schläger(self, pos):
❶       schläger_pos = self.canvas.coords(self.schläger.id)
❷       if pos[2] >= schläger_pos[0] and pos[0] <= schläger_pos[2]:
❸           if pos[3] >= schläger_pos[1] and pos[3] <= schläger_pos[3]:
❹               return True
            return False
```

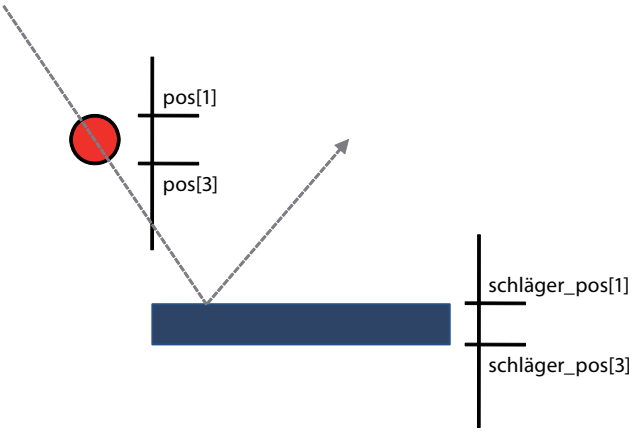
Als Erstes definieren wir in ❶ die Funktion mit dem Parameter `pos`. Diese Zeile enthält die aktuellen Koordinaten des Balls. In ❷ bekommen wir dann die Koordinaten des Schlägers und speichern sie in der Variable `pos`.

In ❸ haben wir den ersten Teil unserer ersten `if-then`-Anweisung. Wir sagen damit: »Wenn die rechte Seite neben dem Ball größer ist als die linke Seite neben dem Schläger und die linke Seite neben dem Balls kleiner ist als die rechte Seite neben dem Schläger…« Hier enthält `pos[2]` die x-Koordinate für die Seite rechts neben dem Ball. `schläger_pos[0]` enthält die x-Koordinate für die Seite links neben dem Schläger, und `schläger_pos[2]` enthält die x-Koordinate für die Seite rechts neben dem Schläger. Das folgende Diagramm zeigt, wie diese Koordinaten aussehen, kurz bevor der Ball auf dem Schläger auftrifft.



Der Ball fällt in Richtung des Schlägers, aber in diesem Fall siehst Du, dass die rechte Seite neben dem Ball (`pos[2]`) noch nicht die linke Seite des Schlägers (`schläger_[0]`) passiert hat.

In ❹ schauen wir nach, ob die Unterseite des Balls (`pos[3]`) sich zwischen der Oberseite des Schlägers (`schläger_post[1]`) und seiner Unterseite (`schläger_post[3]`) befindet. Im nächsten Diagramm siehst Du, dass die Unterseite des Balls (`pos[3]`) noch auf die Oberseite des Schlägers (`schläger_pos[1]`) auftreffen muss.

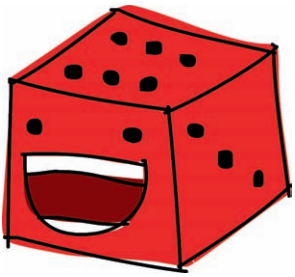


Aufgrund der aktuellen Position des Balls würde die Funktion `triff_schläger` falsch zurückgeben.

Achtung!

Warum müssen wir wissen, ob die Unterseite des Balls sich zwischen der oberen und der unteren Seite des Schlägers befindet? Warum schauen wir nicht einfach nach, ob die Unterseite des Balls die Oberseite des Schlägers berührt hat? Einfach, weil jedes Mal, wenn wir den Ball über die Leinwand bewegen, 3-Pixel-Sprünge gemacht werden. Wenn wir lediglich nachgucken würden, ob der Ball die Oberseite des Schlägers (`pos[1]`) erreicht hat, könnten wir schon über diese Position hinaus gesprungen sein. In diesem Fall würde der Ball weiterfliegen und ohne anzuhalten durch den Schläger gehen.

15.3 Dem Spiel etwas Zufälliges geben



Jetzt ist es an der Zeit, aus unserem Programm mit dem hüpfenden Ball und dem Schläger ein Spiel zu machen. Spiele brauchen ein Element des Zufalls – irgendetwas, damit der Spieler auch verlieren kann. So, wie unser Spiel jetzt ist, würde der Ball unendlich lange durch die Gegend hüpfen, also gibt es auch nichts zu verlieren.

Wir werden unser Spiel dadurch abschließen, dass wir Code hinzufügen, der das Spiel beendet, falls der Ball die Unterseite der Leinwand berührt (mit anderen Worten: falls er auf den Boden fällt).

Als Erstes fügen wir die Objekt-Variable `hit_bottom` (engl. für »triff Boden«) der Funktion `__init__` ganz unten in der Ball-Klasse hinzu:

```
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()
        self.hit_bottom = False
```

Danach ändern wir ganz unten im Programm die Hauptschleife:

```
while 1:
    if ball.hit_bottom == False:
        ball.draw()
        schläger.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

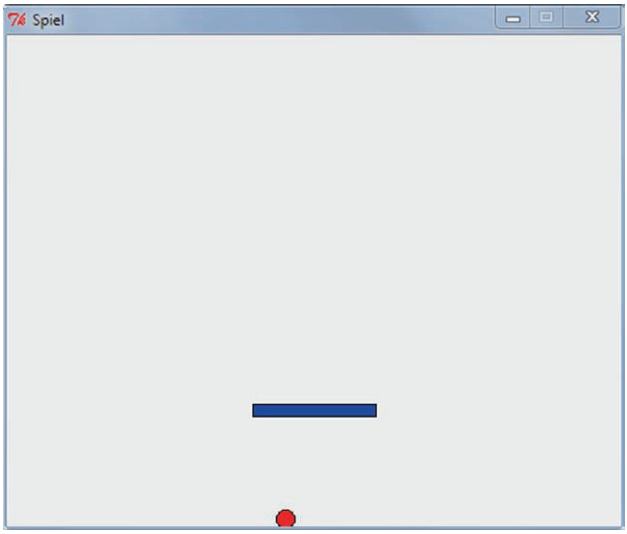
Jetzt überprüft die Schleife ständig `hit_bottom`, um nachzuschauen, ob der Ball den Boden des Fensters berührt hat. Wie Du in unserer `if`-Anweisung siehst, sollte der Code ständig Ball und Schläger in Bewegung halten, solange der Ball nicht den Boden berührt hat. Das Spiel ist beendet, sobald sich Ball und Schläger nicht mehr bewegen. (Wenn wir sie nicht mehr animieren.)

Die letzte Änderung betrifft die `draw`-Funktion der `Ball`-Klasse:

```
def draw(self):
    self.canvas.move(self.id, self.x, self.y)
    pos = self.canvas.coords(self.id)
    if pos[1] <= 0:
        self.y = 3
    if pos[3] >= self.canvas_height:
        self.hit_bottom = True
    if self.schläger(pos) == True:
        self.y = -3
    if pos[0] <= 0:
        self.x = 3
    if pos[2] >= self.canvas_width:
        self.x = -3
```

Wir haben die `if`-Anweisung geändert, um zu prüfen, ob der Ball den Boden berührt hat (ob er höher oder gleich `canvas_height` ist). Falls ja, setzen wir in der folgenden Zeile `hit_bottom` auf wahr, anstatt den Wert der `y`-Variable zu ändern, da es nicht mehr nötig ist, den Ball springen zu lassen, sobald er den Boden des Fensters berührt hat.

Wenn Du das Spiel jetzt laufen lässt und den Ball nicht mehr mit dem Schläger erwischst, kommt alle Bewegung auf Deinem Bildschirm zum Stillstand, und das Spiel ist beendet, sobald der Ball den Boden berührt hat:



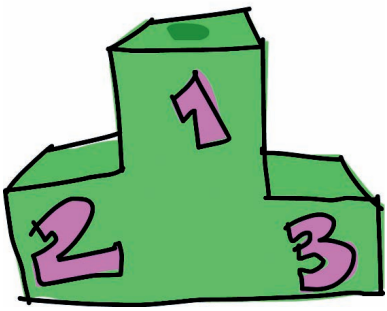
Dein Programm sollte wie das folgende Listing aussehen. Falls Du Probleme hast, Dein Spiel ans Laufen zu bekommen, vergleiche es mit Deinen Eingaben:

```
from tkinter import *
import random
import time

class Ball:
    def __init__(self, canvas, schläger, color):
        self.canvas = canvas
        self.schläger = schläger
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)
        starts = [-3, -2, -1, 1, 2, 3]
        random.shuffle(starts)
        self.x = starts[0]
        self.y = -3
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()
        self.hit_bottom = False

    def triff_schläger(self, pos):
        schläger_pos = self.canvas.coords(self.schläger.id)
        if pos[2] >= schläger_pos[0] and pos[0] <= schläger_pos[2]:
            if pos[3] >= schläger_pos[1] and pos[3] <= schläger_pos[3]:
                return True
        return False

    def draw(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[1] <= 0:
            self.y = 3
        if pos[3] >= self.canvas_height:
            self.hit_bottom = True
        if self.triff_schläger(pos) == True:
            self.y = -3
        if pos[0] <= 0:
            self.x = 3
        if pos[2] >= self.canvas_width:
            self.x = -3

class Schläger:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_rectangle(0, 0, 100, 10, fill=color)
        self.canvas.move(self.id, 200, 300)
        self.x = 0
```

```
        self.canvas_width = self.canvas.winfo_width()
        self.canvas.bind_all('<KeyPress-Left>', self.nach_links)
        self.canvas.bind_all('<KeyPress-Right>', self.nach_rechts)

    def draw(self):
        self.canvas.move(self.id, self.x, 0)
        pos = self.canvas.coords(self.id)
        if pos[0] <= 0:
            self.x = 0
        elif pos[2] >= self.canvas_width:
            self.x = 0

    def nach_links(self, evt):
        self.x = -2

    def nach_rechts(self, evt):
        self.x = 2

tk = Tk()
tk.title("Spiel")
tk.resizable(0, 0)
tk.wm_attributes("-topmost", 1)
canvas = Canvas(tk, width=500, height=400, bd=0, highlightthickness=0)
canvas.pack()
tk.update()

schläger = Schläger(canvas, 'blue')
ball = Ball(canvas, schläger, 'red')

while 1:
    if ball.hit_bottom == False:
        ball.draw()
        schläger.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

15.4 Was Du gelernt hast

In diesem Kapitel haben wir unser erstes Spiel mit dem Modul `tkinter` abgeschlossen. Wir haben für den Schläger, der in unserem Spiel benutzt wird, Klassen erzeugt, und wir haben mithilfe von Koordinaten geprüft, ob der Ball den Schläger oder die Wände der Leinwand berührt. Wir haben Ereignisbindungen benutzt, um die linke und rechte Pfeiltaste



an die Bewegung des Schlägers zu binden. Mit der Hauptschleife haben wir die Funktion `draw` aufgerufen, um den Schläger zu animieren. Zum Schluss haben wir unseren Code so geändert, dass er ein Zufallselement bekam: Wenn ein Spieler den Ball verpasst, ist das Spiel vorbei, sobald der Ball den Boden der Leinwand berührt.

15.5 Programmier-Puzzles

Im Moment ist unser Spiel noch etwas einfach. Man kann noch jede Menge ändern, um ein professionelleres Spiel zu programmieren. Versuche es auf die folgenden Weisen interessanter zu machen, und überprüfe Deine Antworten unter *www.dpunkt.de/python*.

#1: Verzögere den Spielstart

Unser Spiel geht ein bisschen schnell los, und man muss auch noch auf die Leinwand klicken, damit es reagiert, wenn der Spieler auf die linke und rechte Pfeiltaste der Tastatur drückt. Kannst Du beim Start des Spiels eine Verzögerung einbauen, damit der Spieler genug Zeit hat, auf die Leinwand zu klicken? Oder – noch besser – kannst Du eine Ereignisbindung an einen Mausklick hinzufügen, damit nur dadurch das Spiel gestartet wird?

- Hinweis 1:
  In der `Schläger`-Klasse hast Du bereits Ereignisbindungen verwendet. Dies könnte ein guter Ausgangspunkt sein.
- Hinweis 2:
  Die Ereignisbindung für die linke Maustaste ist der String `'<Button-1>'`.

#2: Ein richtiges »Game Over«

Wenn das Spiel zu Ende ist, friert alles einfach ein, und das ist nicht sehr spielerfreundlich. Versuche den Text »Game over« erscheinen zu lassen, wenn der Ball den Boden des Fensters berührt. Du könntest die Funktion `create_text` verwenden, aber vielleicht findest Du auch den benannten Parameter `state` nützlich (er nimmt Werte wie `normal` und `hidden` [engl. für »verborgen«] an). Sieh Dir im Abschnitt 13.12 `itemconfig` an. Als zusätzliche Herausforderung baust Du eine Verzögerung ein, damit der Text nicht sofort erscheint.

#3: Beschleunige den Ball

Falls Du Tennis spielst, weißt Du, dass ein Ball, der auf Deinen Schläger trifft, manchmal schneller wieder zurückfliegt, als er gekommen ist – je nachdem, wie hart Du schlägst. Der Ball in unserem Spiel bewegt sich immer in derselben Geschwindigkeit, egal ob sich der Schläger bewegt oder nicht. Versuche, das Programm so zu ändern, dass sich die Schlägergeschwindigkeit auf die Geschwindigkeit des Balls auswirkt.

#4: Zeichne den Punktestand auf

Wie wäre es mit einem Punktestand? Jedes Mal, wenn der Ball den Schläger berührt, sollte der Punktestand steigen. Versuche, den Punktestand in der oberen rechten Ecke der Leinwand anzuzeigen. Eventuell blätterst Du für einen Tipp noch einmal auf Seite 178 zum Abschnitt 13.12 zurück und siehst Dir `itemconfig` noch einmal an.

Teil III

Herr Strichmann rennt zum Ausgang

16 Wir erstellen Grafiken für das Strichmännchenspiel

Wenn man ein Spiel (oder jedes andere Programm) schreibt, ist es gut, sich einen Plan zu machen. Dein Plan sollte beschreiben, worum es in Deinem Spiel geht, und eine Beschreibung der Hauptelemente und -charaktere Deines Spiels enthalten. Wenn es mit dem Programmieren losgeht, hilft Dir Deine Beschreibung, Dich auf das zu konzentrieren, was Du entwickeln willst. Dein Spiel wird vielleicht nicht immer genau wie die Originalbeschreibung werden – aber das macht nichts.

In diesem Kapitel beginnen wir mit der Entwicklung des lustigen Spiels »Herr Strichmann rennt zum Ausgang«.

16.1 Der Strichmännchen-Spielplan

Hier ist die Beschreibung unseres neuen Spiels:

- Geheimagent Herr Strichmann ist in der Höhle des Dr. Harmlos gefangen, und Du möchtest ihm helfen, durch den Ausgang im obersten Stockwerk zu entkommen.
- Im Spiel gibt es ein Strichmännchen, das von links nach rechts laufen und hochspringen kann. Auf jedem Stockwerk gibt es Ebenen, auf die es springen muss.
- Das Ziel des Spiels ist es, die Ausgangstür zu erreichen, bevor es zu spät ist und das Spiel beendet wird.

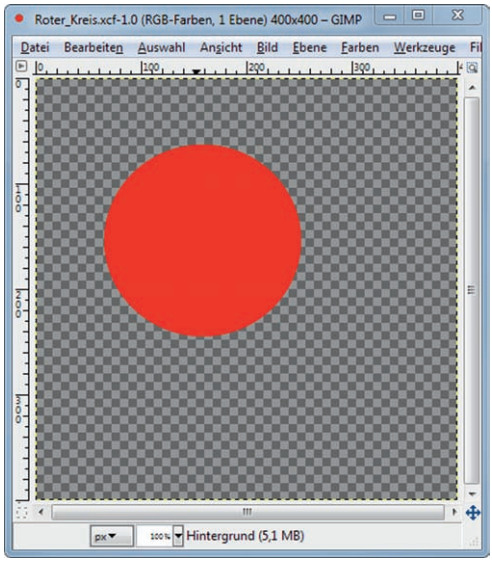
Anhand dieser Beschreibung wissen wir, dass wir mehrere Bilder brauchen – und zwar Bilder von Herrn Strichmann, von den Ebenen und von der Tür. Wir brauchen ganz offensichtlich Code, um all diese Dinge zusammenzubringen. Aber bevor wir so weit sind, erstellen wir zunächst die Grafiken für unser Spiel, das wir im nächsten Kapitel entwickeln werden.

Wie werden wir die Elemente in unserem Spiel zeichnen? Wir könnten Grafiken wie die des springenden Balls und des Schlägers aus dem vorigen Kapitel benutzen. Die sind allerdings viel zu einfach für dieses Spiel. Stattdessen werden wir uns Sprites erzeugen.

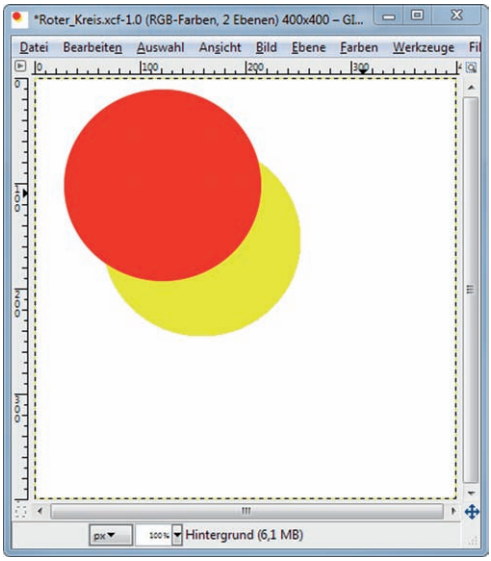
Als *Sprites* bezeichnet man die Dinge in einem Spiel – meistens Figuren. Sprites werden normalerweise vorher erstellt (bevor das Programm läuft) und nicht wie die Polygone im Programm selber, wie es bei unserem Bounce!-Spiel der Fall war. Sowohl Herr Strichmann als auch die Ebenen werden aus Sprites bestehen. Um diese Bilder zu erzeugen, musst Du ein Grafikprogramm installieren.

16.2 GIMP installieren

Es gibt zwar viele Grafikprogramme, für unser Spiel brauchen wir jedoch eines, das Transparenzen unterstützt. (Die Transparenz wird auch Alphakanal genannt.) Durch die Transparenz kann ein Bild Teile enthalten, bei denen keine Farben auf dem Bildschirm dargestellt werden. Wir benötigen transparente Anteile, denn wenn ein Bild bei der Bewegung über den Monitor über ein anderes bewegt wird, möchten wir nicht, dass der Hintergrund des einen Bildes das andere überdeckt. Bei diesem Bild zum Beispiel, steht das Schachbrettmuster im Hintergrund für die transparente Region:



Wenn wir jetzt also das gesamte Bild kopieren und es über einem anderen Bild einfügen, überdeckt der Hintergrund nichts:



GIMP (*http://www.gimp.org*), ist ein kostenloses Bildbearbeitungsprogramm für Linux, MacOSX und Windows, das transparente Bilder unterstützt. Die Website ist zwar leider auf Englisch, das Programm lässt sich jedoch auf Deutsch einstellen. Lade und installiere es wie folgt:

- Wenn Du Windows benutzt, erkennt die GIMP-Website Dein Betriebssystem und leitet Dich unter *Downloads* zur Installationsdatei.
- Wenn Du Ubuntu benutzt, installierst Du GIMP, indem Du das Ubuntu-Software Center öffnest und im Suchfenster *GIMP* eingibst. Sobald *GIMP-Bildverarbeitung* im Suchergebnis auftaucht, klickst Du einmal darauf und anschließend auf den Button *Install* der GIMP-Bildbearbeitung.
- Wenn Du MacOSX benutzt, erkennt die Website Dein Betriebssystem ebenso wie bei Windows und führt Dich nach dem Button *Download* zur Auswahl der richtigen Installationsdatei.

Als Nächstes solltest Du Dir für Dein Spiel einen Ordner anlegen. Dazu klickst Du auf Deinem Desktop irgendwo, wo Platz ist, mit der rechten Maustaste und gehst auf **Neu ▶ Ordner** (in Ubuntu heißt es **Neuen Ordner anlegen**; in MacOSX **Neuer Ordner**). Im darauf folgenden Dialogfenster gibst Du als Ordnernamen *Strichmann* ein.

16.3 Erzeugen der Spielelemente

Sobald Du Dein Grafikprogramm installiert hast, kannst Du mit dem Zeichnen anfangen. Wir erstellen folgende Bilder als unsere Spielelemente:

- Bilder eines Strichmännchens, das nach links und rechts laufen und springen kann
- Bilder einer Ebene in drei unterschiedlichen Größen
- Bilder einer Tür: eine offen und eine geschlossen
- Ein Bild für den Spielhintergrund (weil ein einfacher weißer oder grauer Hintergrund ein langweiliges Spiel ergibt)

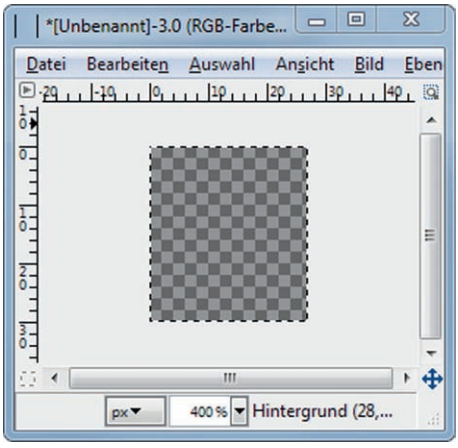
Bevor wir mit dem Zeichnen anfangen, müssen wir unsere Bilder mit transparenten Hintergründen vorbereiten.

Ein transparentes Bild erstellen

Um ein Bild mit Transparenz – einem Alphakanal – einzurichten, startest Du GIMP und führst folgende Schritte durch:

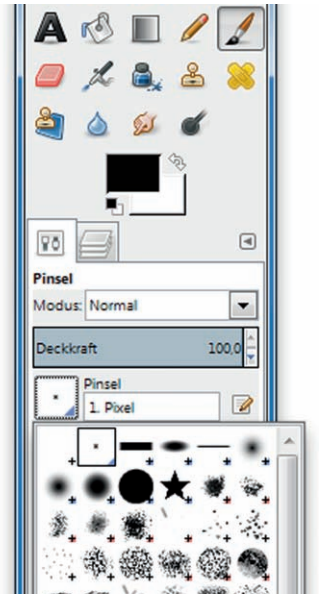
1. Gehe auf **Datei ▶ Neu…**
2. Im Dialogfenster gibst Du als Bildbreite 27 Pixel und als Bildhöhe 30 Pixel ein.
3. Gehe auf **Ebene ▶ Transparenz ▶ Alphakanal hinzufügen**.
4. Gehe auf **Auswahl ▶ Alles**.
5. Gehe auf **Bearbeiten ▶ Ausschneiden.**

Das Endergebnis sollte ein Bild sein, das aus einem Schachbrettmuster aus dunklem und hellem Grau besteht, wie es hier (vergrößert) zu sehen ist:



Jetzt können wir anfangen, unseren Geheimagenten, Herrn Strichmann, zu erzeugen.

Herrn Strichmann zeichnen



Um das erste Strichmännchen-Bild zu zeichnen, klickst Du auf das Pinsel-Werkzeug im GIMP-Werkzeugkasten. Anschließend wählst Du in der Palette darunter die Pinsel-Palette (sie klappt sich wie auf dem Bild rechts zur Seite auf). Aus der Pinsel-Palette wählst Du einen Pinsel aus, der wie ein kleiner Punkt aussieht.

Wir werden drei unterschiedliche Bilder (auch *Frames* genannt) von unserer Strichmännchenfigur malen, um sie beim Laufen und Springen nach rechts zu zeichnen. Diese Frames werden wir benutzen, um Herrn Strichmann zu animieren, wie wir es schon in Kapitel 13 getan haben.

Wenn Du diese Bilder stark vergrößerst, könnten sie in etwa so aussehen:

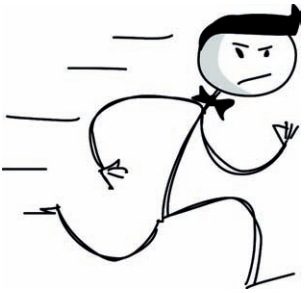
Deine Bilder müssen zwar nicht genauso aussehen, sie sollten aber ein Strichmännchen in drei unterschiedlichen Positionen der Bewegung zeigen. Denke daran, dass jedes dieser Bilder 27 Pixel hoch und 30 Pixel breit ist.

Herr Strichmann rennt nach rechts

Als Erstes zeichnen wir eine Sequenz von Frames von Herrn Strichmann, wie er nach rechts rennt. Das erste Bild erzeugst Du folgendermaßen:

1. Zeichne das erste Bild (das linke Bild in der vorigen Illustration).
2. Gehe auf **Datei ▶ Exportieren …**
3. Im Dialogfenster gibst Du als Dateinamen *Figur-R1.gif* ein. Klicke dann auf den kleinen Button mit dem +-Zeichen, neben dem **Dateityp: Nach Endung** steht.
4. In der Liste, die dann aufklappt, wählst Du **GIF-Bild** aus.
5. Speichere die Datei im *Strichmännchen*-Ordner, den Du vorher erzeugt hast.

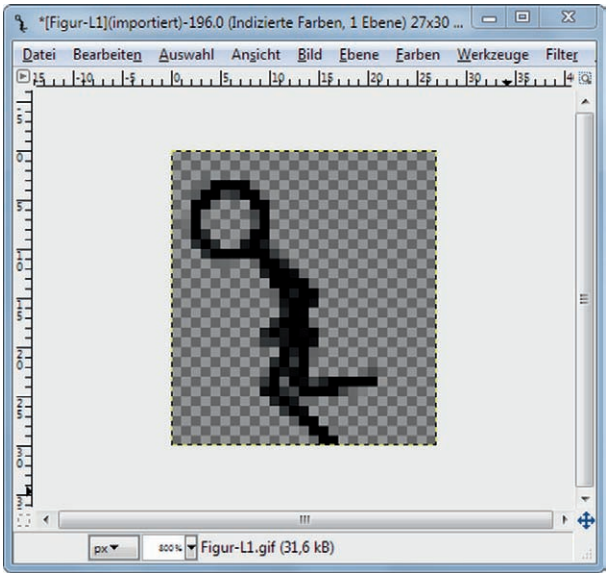
Führe die gleichen Schritte durch, um ein neues 27×30 Pixel großes Bild anzulegen, und zeichne den nächsten Herrn Strichmann. Speichere dieses Bild als *Figur-R2.gif*. Wiederhole diesen ganzen Ablauf bis zum letzten Bild, das Du als *Figur-R3.gif* speicherst.



Herr Strichmann rennt nach links

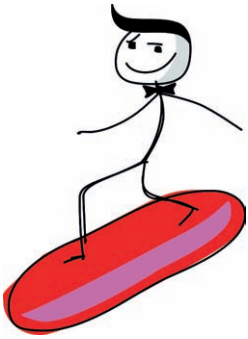
Um das Strichmännchen auch nach links laufen zu lassen, brauchen wir es nicht erneut zu zeichnen. Stattdessen spiegeln wir mit GIMP unsere Frames von Herrn Strichmann, der nach rechts läuft.

In GIMP öffnest Du die Bilder nacheinander und wählst unter **Werkzeuge ▶ Transformationen ▶ Spiegeln**. Wenn Du danach auf das Bild klickst, solltest Du sehen, dass es gespiegelt wird. Speichere diese Bilder als *Figur-L1.gif*, *Figur-L2.gif* und *Figur-L3.gif*.



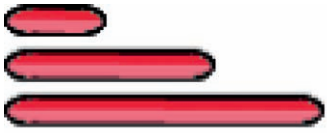
Jetzt haben wir sechs Bilder von Herrn Strichmann, brauchen aber noch Bilder für die Ebenen und die Ausgangstür.

Ebenen zeichnen



Wir erstellen drei Ebenen in unterschiedlichen Größen: 100 Pixel breit und 10 Pixel hoch, 60 Pixel breit und 10 Pixel hoch sowie 30 Pixel und 10 Pixel hoch. Du kannst diese Ebenen so zeichnen, wie Du möchtest, musst allerdings darauf achten, dass ihre Hintergründe wie bei den Strichmännchen transparent sind.

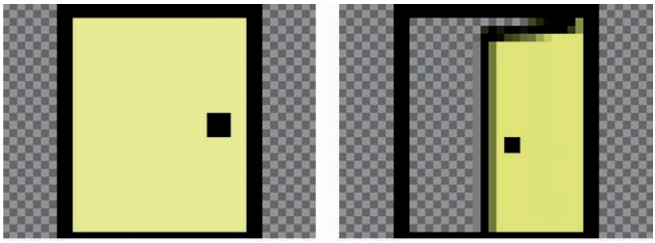
So könnten diese drei Bilder der Ebenen vergrößert aussehen:



Genau wie die Bilder der Strichmännchen speicherst Du die Ebenen im *Strichmännchen*-Ordner. Nenne die kleinste Ebene *Ebene1.gif*, die mittlere *Ebene2.gif* und die größte *Ebene3.gif*.

Die Tür zeichnen

Die Größe der Tür sollte der Größe von Herrn Strichmann entsprechen (27 Pixel breit und 30 Pixel hoch). Wir benötigen zwei Bilder: eines mit der geschlossenen Tür und das andere mit der offenen Tür. Die Türen (wieder vergrößert) könnten so aussehen:



Um diese Türen zu erstellen, machst Du Folgendes:

1. Klicke in die Box mit der Vordergrundfarbe (ganz unten im GIMP-Werkzeugkasten), um das Fenster *Vordergrundfarbe ändern* zu bekommen. Wähle darin die gewünschte Farbe für Deine Tür. Im Beispiel rechts wurde Gelb gewählt.
2. Wähle das Werkzeug *Füllen* (das ist der Farbeimer im Werkzeugkasten), und fülle das Fenster mit der von Dir gewählten Farbe.

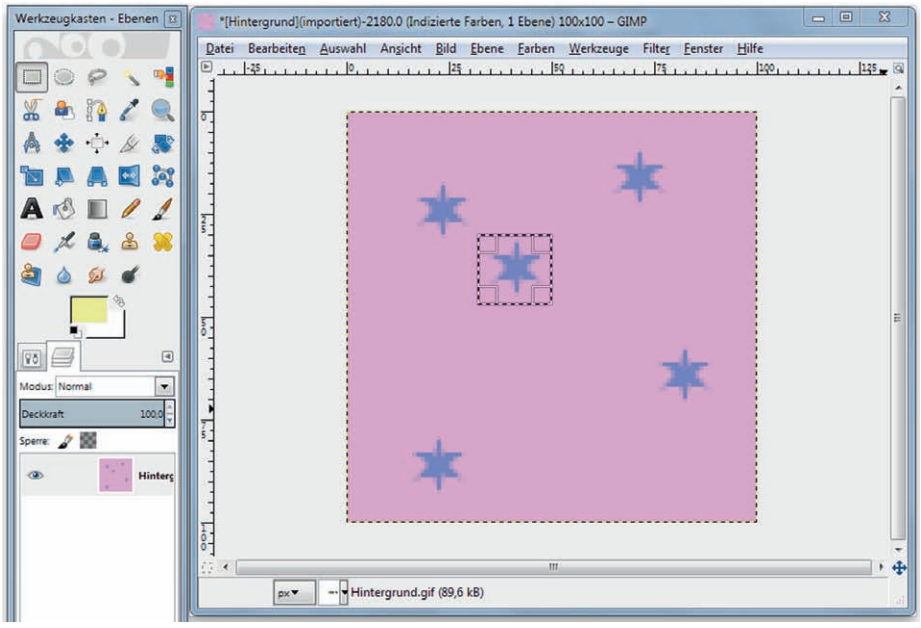
3. Ändere die Vordergrundfarbe in Schwarz.
4. Wähle dann entweder das Stift- oder das Pinsel-Werkzeug (rechts neben dem Füllen-Werkzeug), und male den schwarzen Umriss der Tür und den Türknauf.
5. Speichere die Bilder im *Strichmännchen*-Ordner, und nenne sie *Tür1.gif* und *Tür2.gif*.

Den Hintergrund zeichnen

Das letzte Bild, das wir noch brauchen, ist der Hintergrund. Dieses Bild machen wir 100 Pixel breit und 100 Pixel hoch. Es benötigt keinen transparenten Hintergrund, da wir es mit einer einzigen Farbe ausfüllen, die gewissermaßen die Hintergrund-Tapete für alle anderen Elemente unseres Spiels darstellt.

Um den Hintergrund zu erstellen, gehst Du auf **Datei ▶ Neu…** und gibst als Bildgröße 100 Pixel Breite und 100 Pixel Höhe ein. Suche Dir für die Tapete des Bösewichts eine ausreichend üble Farbe aus. Ich habe mich für ein dunkles Rosa entschieden.

Du kannst Deine Tapete noch mit Blumenstreifen oder Sternen ausschmücken – was immer Du für dieses Spiel für passend hältst. Wenn Du beispielsweise Sterne auf Deiner Tapete haben möchtest, wählst Du eine andere Farbe aus,

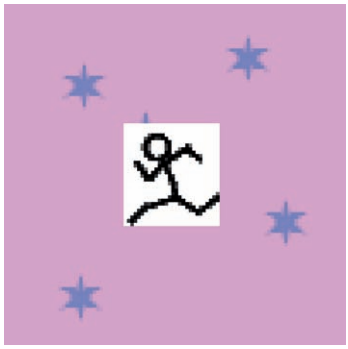


nimmst das Stift-Werkzeug und zeichnest damit Deinen ersten Stern. Mit einem der Auswahl-Werkzeuge (zum Beispiel der Rechteck-Auswahl) ziehst Du einen Kasten um den Stern auf, kopierst ihn und fügst ihn an einer beliebigen Stelle auf dem Bild ein (gehe dazu auf **Bearbeiten ▶ Kopieren** und dann auf **Bearbeiten ▶ Einfügen**). Es sollte Dir möglich sein, das eingefügte Bild auf dem Monitor zu bewegen, wenn Du auf es klickst. Unten auf Seite 218 siehst Du ein Beispiel mit einigen Sternen und der aktiven Rechteckauswahl im Werkzeugkasten.

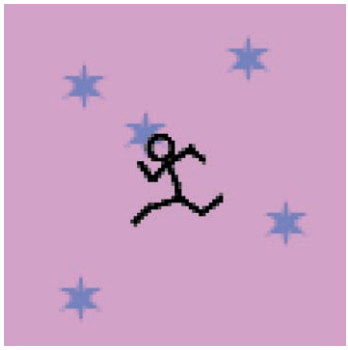
Sobald Du mit Deiner Zeichnung zufrieden bist, speicherst Du das Bild als *Hintergrund.gif* im *Strichmann*-Ordner.

Transparenz

Mit den nun erstellten Grafiken kannst Du selbst herausfinden, warum sie (mit Ausnahme des Hintergrunds) Transparenz benötigen. Was würde passieren, wenn wir Herrn Strichmann vor unserer Hintergrundtapete platzieren würden, ohne dass er einen transparenten Hintergrund hätte? Hier ist die Antwort:

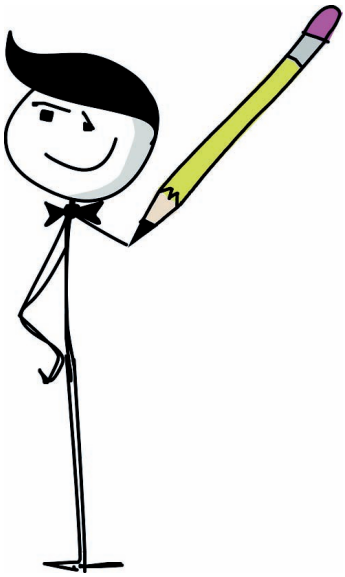


Der weiße Hintergrund von Herrn Strichmann überdeckt Teile der Hintergrundtapete. Wenn wir dagegen unser transparentes Bild verwenden, bekommen wir Folgendes:

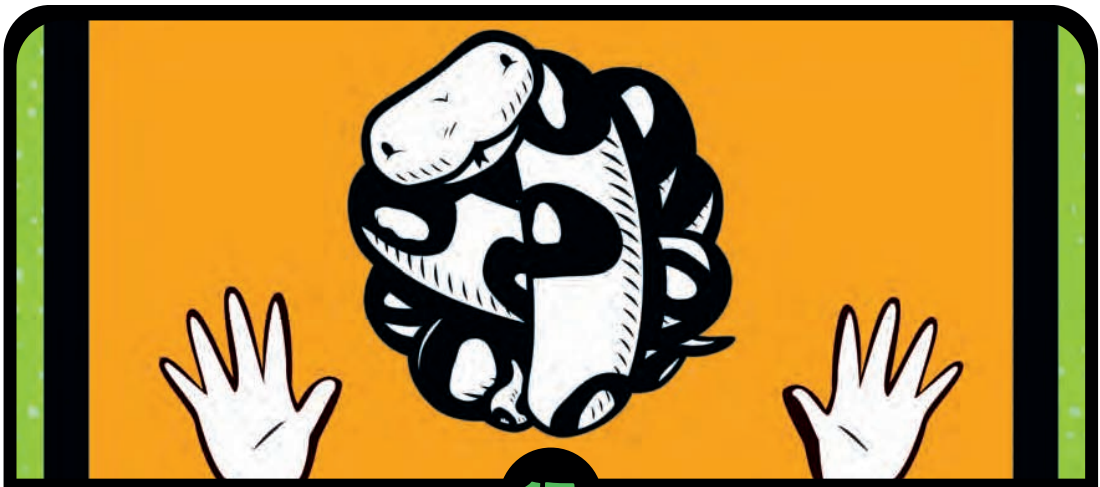


Das Strichmännchen verdeckt nur so viel Platz auf der Hintergrundtapete, wie es selbst einnimmt. Das sieht viel professioneller aus!

16.4 Was Du gelernt hast



In diesem Kapitel hast Du gelernt, wie man einen grundlegenden Plan für ein Spiel schreibt (in diesem Fall für *Herr Strichmann rennt zum Ausgang*), und herausgefunden, wo man anfängt. Da wir die grafischen Elemente benötigen, bevor wir das Spiel machen können, haben wir mit einem Grafikprogramm die einfachen Grafiken für unser Spiel erzeugt. Dabei hast Du im Verlauf gelernt, wie man die Hintergründe dieser Bilder transparent macht, damit sie nicht die anderen Bilder auf dem Monitor überdecken. Im nächsten Kapitel werden wir einige der Klassen für unser Spiel erstellen.



17 Entwicklung des Strichmännchenspiels

Im vorigen Kapitel haben wir die Bilder für unser Spiel *Herr Strichmann rennt zum Ausgang* erzeugt. Daher können wir jetzt mit der Entwicklung des Codes beginnen. Die Beschreibung des Spiels aus dem vorigen Kapitel gibt uns eine ungefähre Vorstellung davon, was wir benötigen: ein Strichmännchen, das rennen und springen kann, und Ebenen, auf die es springen kann.

Wir brauchen Code, um das Strichmännchen und seine Bewegung auf dem Monitor darzustellen und um die Plattformen anzuzeigen. Bevor wir jedoch anfangen, diesen Code zu schreiben, müssen wir eine Leinwand erzeugen, auf der unser Hintergrundbild angezeigt wird.

17.1 Erzeugen der Spiel-Klasse

Als Erstes erzeugen wir eine Klasse namens `Spiel`, die die Hauptsteuerung unseres Programms übernimmt. Die `Spiel`-Klasse wird eine `__init__`-Funktion zur Initialisierung des Spiels enthalten sowie eine Funktion `Hauptschleife`, die für die Animation sorgt.

17.2 Den Fenstertitel bestimmen und die Leinwand erzeugen

Im ersten Teil der `__init__`-Funktion legen wir den Titel des Fensters fest und erzeugen eine Leinwand. Wie Du siehst, ähnelt dieser Code dem, den wir für das Spiel Bounce! in Kapitel 14 geschrieben haben. Öffne Deinen Editor, gib den folgenden Code ein, und speichere die Datei als *Strichmännchenspiel.py*. Achte darauf, dass Du sie im gleichen Verzeichnis speicherst, das wir in Kapitel 16 angelegt haben (namens *Strichmännchen*).

```
from tkinter import *
import random
import time
class Spiel:
    def __init__(self):
        self.tk = Tk()
        self.tk.title("Herr Strichmann rennt zum Ausgang")
        self.tk.resizable(0, 0)
        self.tk.wm_attributes("-topmost", 1)
        self.canvas = Canvas(self.tk, width=500, height=500, \
                highlightthickness=0)
        self.canvas.pack()
        self.tk.update()
        self.canvas_height = 500
        self.canvas_width = 500
```

In der ersten Hälfte dieses Programms (sie reicht von `from tkinter import *` bis `self.tk.wm_attributes`) erzeugen wir das `tk`-Objekt und setzen den Titel des Fensters mit `self.tk.title` auf (`"Herr Strichmann rennt zum Ausgang"`). Wir fixieren die Fenstergröße (damit man sie nicht mehr verändern kann), indem wir die Funktion `resizable` aufrufen, und lassen mit der Funktion `wm_attributes` das Fenster vor allen anderen Fenstern erscheinen.

Als Nächstes erzeugen wir mit der Zeile `self.canvas = Canvas` die Leinwand und rufen die Funktionen `pack` und `update` des `tk`-Objekts auf. Zum Schluss erzeugen wir für unsere `Spiel`-Klasse die Variablen `height` (Höhe) und `width` (Breite), um Höhe und Breite der Leinwand zu speichern.

Achtung!

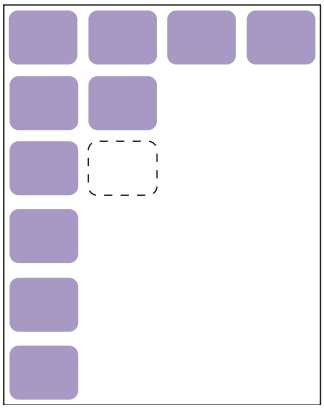
Der Rückwärtsschrägstrich(\) in der Zeile self.canvas = Canvas wird nur zur Trennung der langen Code-Zeile verwendet. Dies ist zwar nicht unbedingt erforderlich, aber ich habe ihn zwecks besserer Lesbarkeit eingefügt, da die vollständige Zeile nicht auf die Seite passt.

Abschluss der __init__-Funktion

Gib in der Datei *Strichmännchenspiel.py* jetzt den Rest der `__init__`-Funktion ein. Dieser Code wird das Hintergrundbild laden und es dann auf der Leinwand darstellen:

```
        self.tk.update()
        self.canvas_height = 500
        self.canvas_width = 500
❶       self.bg = PhotoImage(file="Hintergrund.gif")
❷       w = self.bg.width()
        h = self.bg.height()
❸       for x in range(0, 5):
❹           for y in range(0, 5):
❺               self.canvas.create_image(x * w, y * h, \
                        image=self.bg, anchor='nw')
❻       self.sprites = []
        self.rennen = True
```

In ❶ erzeugen wir die Variable `bg`, die ein `PhotoImage`-Objekt enthält – die Hintergrund-Bilddatei *Hintergrund.gif*, die wir in Kapitel 16 erzeugt haben. Als Nächstes beginnen wir in ❷ mit dem Speichern der Breite und Höhe des Bildes in den Variablen `w` und `h`. Die `PhotoImage`-Klassen-Funktionen `width` und `height` geben nach dem Laden die Größe des Bildes zurück.



Danach kommen zwei Schleifen innerhalb dieser Funktion. Was tun sie? Stell Dir vor, Du hättest einen kleinen rechteckigen Stempel, ein Stempelkissen und ein großes Stück Papier. Wie würdest Du damit das gesamte Blatt mit Farbe ausfüllen? Du könntest entweder einfach so lange wild (zufallsmäßig) herumstempeln, bis das ganze Blatt ausgefüllt ist. Das sähe dann ziemlich chaotisch aus und würde auch eine ganze Weile dauern, aber am Ende hättest Du das Blatt ausgefüllt. Oder Du bestempelst in einer Spalte das Blatt von oben nach unten, gehst wieder nach oben und stempelst die nächste Spalte nach unten, wie rechts zu sehen ist.

Unser Stempel wird das Hintergrundbild sein, das wir im vorigen Kapitel erstellt haben. Wir wissen, dass die Leinwand 500 Pixel breit und 500 Pixel hoch ist und dass wir unser Hintergrundbild als Quadrat mit 100 Pixeln Kantenlänge angelegt haben. Daraus folgt, dass wir fünf Spalten zur Seite und fünf Spalten nach unten brauchen, um das Fenster mit Bildern zu füllen. Mit der Schleife in ❸ berechnen wir die Anzahl der Spalten zur Seite und mit der Schleife in ❹ die Anzahl der Reihen nach unten.

In ❺ multiplizieren wir die erste Schleifen-Variable `x` mit der Breite des Bildes (`x * w`) und berechnen damit, wie weit wir seitlich zeichnen. Anschließend multiplizieren wir die zweite Schleifen-Variable `y` mit der Höhe des Bildes (`x * h`), um zu berechnen, wie weit wir nach unten zeichnen. Wir verwenden dann die Funktion `create_image` des Objekts aus canvas (`self.canvas.create_image`), um das Bild mit diesen Koordinaten auf dem Monitor zu zeichnen.

In ❻ und der folgenden Zeile erzeugen wir die beiden Variablen `sprites`, die eine bis jetzt leere Liste beinhaltet, und `rennen`, die den booleschen Wert `True` enthält. Diese Variablen brauchen wir später noch.

Erzeugen der Hauptschleifen-Funktion

Wir werden die Funktion `Hauptschleife` in der `Spiel`-Klasse zum Animieren unseres Spiels verwenden. Diese Funktion hat viel Ähnlichkeit mit der Hauptschleife (oder Animationsschleife), die wir für das Spiel Bounce! in Kapitel 14 erzeugt haben. Hier ist sie:

```
            for x in range(0, 5):
                for y in range(0, 5):
                    self.canvas.create_image(x * w, y * h, \
                        image=self.bg, anchor='nw')
            self.sprites = []
            self.rennen = True

        def Hauptschleife(self):
❶           while 1:
❷               if self.rennen == True:
❸                   for sprite in self.sprites:
❹               sprite.move()
❺               self.tk.update_idletasks()
                self.tk.update()
                time.sleep(0.01)
```

In ❶ erzeugen wir eine `while`-Schleife, die so lange läuft, bis das Spiel-Fenster geschlossen wird. Als Nächstes prüfen wir in ❷, ob die Variable wahr (`True`) ist. Falls ja, schleifen wir alle Sprites in der Liste der sprites (`self.sprites`) in ❸ durch, indem wir für jedes Sprite in ❹ die Funktion `move` aufrufen. (Natürlich müssen wir zunächst Sprites erstellen; daher macht dieser Code noch nichts, wenn Du das Programm jetzt laufen lässt.)

Die letzten drei Zeilen der Funktion, die in ❺ beginnen, zwingen das `tk`-Objekt dazu, das Monitorbild neu aufzubauen und einen Sekundenbruchteil lang mit `sleep` zu pausieren, wie wir das schon beim Spiel Bounce! in Kapitel 14 getan haben.

Damit Du den Code jetzt durchlaufen lassen kannst, fügst Du die folgenden beiden Zeilen hinzu (beachte, dass diese beiden Zeilen keine Einrükkung benötigen) und speicherst die Datei.

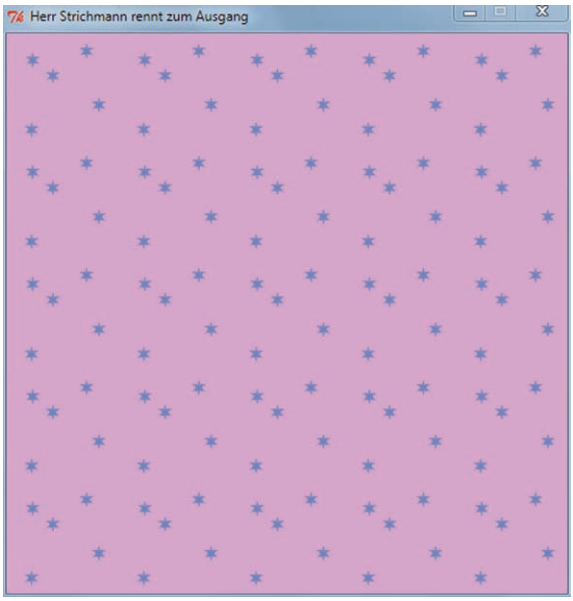
```
s = Spiel()
s.Hauptschleife()
```

Achtung!

Achte darauf, dass Du diesen Code ganz unten in Deine Spiel-Datei schreibst. Sorge außerdem dafür, dass Deine Bilder im selben Ordner wie die Python-Datei liegen. Falls Du den Ordner *Strichmännchen* in Kapitel 16 angelegt und dort alle Deine Bilder gespeichert hast, sollte die Python-Datei für dieses Spiel ebenfalls dort sein.

Dieser Code erzeugt ein Objekt der `Spiel`-Klasse und speichert sie als Variable `s`. Anschließend rufen wir die Funktion `Hauptschleife` für das neue Objekt auf, um das Spiel-Fenster zu zeichnen.

Sobald Du das Programm gespeichert hast, führst Du es in IDLE aus, indem Du auf **Run ▶ Run Module** gehst. Danach erscheint ein Fenster, das mit dem Hintergrundbild ausgefüllt ist.

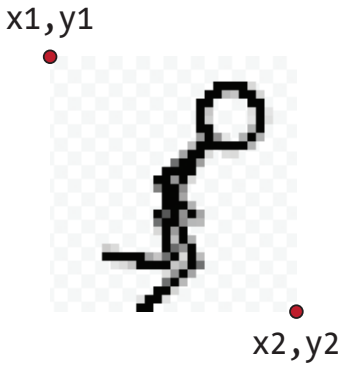


Jetzt haben wir einen hübschen Hintergrund für unser Spiel sowie eine Animation-Schleife erzeugt, die uns die Sprites zeichnen wird (sobald wir sie erstellt haben).

17.3 Erstellen der Klasse Koordinaten

Jetzt erstellen wir die Klasse, in der wir die Position von etwas im Spielfenster festlegen. Diese Klasse speichert die Koordinaten oben links (x1 und y1) und unten rechts (x2 und y2) jeder Komponente unseres Spiels.

So wird beispielsweise die Position des Strichmännchens mit Koordinaten bestimmt:



Unsere neue Klasse werden wir `Koordinaten` nennen, und sie wird nur eine `__init__`-Funktion enthalten, der wir die vier Parameter (`x1`, `y1`, `x2`, und `y2`) übergeben. Hier ist der Code, den Du dafür hinzufügen musst (lege ihn an den Anfang der Datei *Strichmännchenspiel.py*):

```
class Koordinaten:
    def __init__(self, x1=0, y1=0, x2=0, y2=0):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
```

Beachte, dass jeder Parameter als Objekt-Variable mit gleichem Namen (`x1`, `y1`, `x2`, und `y2`) gespeichert wird. Wir werden bald Objekte dieser Klasse einsetzen.

17.4 Zusammenstöße erkennen

Wir wissen jetzt, wie man die Position unserer Sprites speichert. Wir müssen aber noch herausfinden, wie man feststellt, ob ein Sprite den anderen berührt hat – wenn Herr Strichmann über den Bildschirm springt und dabei eine der Ebenen berührt. Damit dieses Problem leichter zu lösen ist, teilen wir es in zwei kleinere Probleme auf: Wir detektieren (erkennen) die Zusammenstöße in vertikaler Richtung, und wir detektieren die Zusammenstöße in horizontaler Richtung. Hinterher kombinieren wir diese kleineren Problemlösungen und können so leicht feststellen, ob sich die Sprites in irgendeiner Richtung berühren!

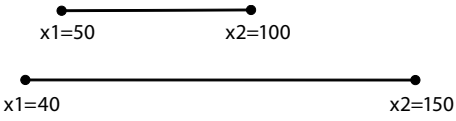
Sprites stoßen horizontal zusammen

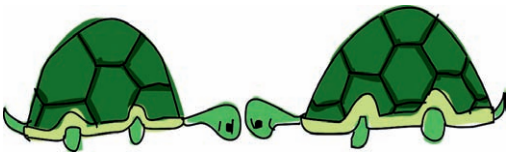
Als Erstes erzeugen wir die Funktion `innerhalb_x`, um festzustellen, ob sich ein Satz von x-Koordinaten (`x1` und `x2`) mit einem anderen Satz von x-Koordinaten (auch `x1` und `x2`) überlappt. Dafür gibt es mehrere Möglichkeiten. Hier ist ein einfaches Verfahren, das Du einfach unterhalb der `Koordinaten`-Klasse hinzufügen kannst:

```
    class Koordinaten:
        def __init__(self, x1=0, y1=0, x2=0, y2=0):
            self.x1 = x1
            self.y1 = y1
            self.x2 = x2
            self.y2 = y2

    def innerhalb_x(co1, co2):
❶       if co1.x1 > co2.x1 and co1.x1 < co2.x2:
❷           return True
❸       elif co1.x2 > co2.x1 and co1.x2 < co2.x2:
❹           return True
❺       elif co2.x1 > co1.x1 and co2.x1 < co1.x2:
            return True
❻       elif co2.x2 > co1.x1 and co2.x2 < co1.x1:
            return True
❼       else:
❽           return False
```

Die Funktion `innerhalb_x` nimmt die beiden Parameter `co1` und `co2` auf, die beide Objekte der Klasse `Koordinaten` sind. In ❶ prüfen wir, ob die äußerste linke Position des ersten Koordinaten-Objekts (`co1.x1`) zwischen der äußersten linken Position (`co2. x1`) und der äußersten rechten Position (`co2. x2`) des zweiten Koordinaten-Objekts liegt. Wenn dem so ist, geben wir in ❷ `True` zurück.

Betrachten wir jetzt zwei Linien mit überlappenden x-Koordinaten, um zu verstehen, wie das Ganze funktioniert. Beide Linien fangen bei `x1` an und enden bei `x2`.





Die erste Linie in diesem Diagramm (`co1`) beginnt an der Pixel-Position 50 (`x1`) und endet an Position 100 (`x2`). Die zweite Linie (`co2`) startet an Position 40 und endet bei 150. Weil in diesem Fall die `x1`-Position der ersten Linie zwischen der `x1`- und `x2`-Position der zweiten Linie liegt, wäre die `if`-Anweisung in dieser Funktion bei diesen beiden Koordinaten-Sätzen wahr (`True`).

Mit dem `elif` in ❸ prüfen wir, ob die äußerste rechte Position der ersten Linie (`co1.x2`) zwischen der äußersten linken Position (`co2.x1`) und der äußersten rechten Position (`co2.x2`) der zweiten Linie liegt. Falls sie das tut, geben wir in ❹ `True` (wahr) zurück. Die beiden `elif`-Anweisungen in ❺ und ❻ machen fast das Gleiche: Sie vergleichen die äußersten linken und rechten Positionen der zweiten Linie (`co2`) mit denen der ersten Linie (`co1`).

Falls keine der `if`-Anweisungen passt, kommen wir in ❼ zu `else` und geben in ❽ `False` (falsch) zurück. Dies bedeutet im Prinzip: »Nein, die beiden Koordinaten-Objekte überlappen sich nicht horizontal.«

Schau Dir noch einmal das Diagramm an, das die erste und zweite Linie zeigt, wenn Du ein Beispiel für die Arbeit dieser Funktion sehen willst: Die `x1`- und `x2`-Positionen des ersten Koordinaten-Objekts sind 40 und 100, die `x1`- und `x2`-Positionen des zweiten Koordinaten-Objekts sind 50 und 150. Hier siehst Du, was passiert, wenn wir die Funktion `innerhalb_x` aufrufen, die wir geschrieben haben:

```
>>> c1 = Koordinaten(40, 40, 100, 100)
>>> c2 = Koordinaten(50, 50, 150, 150)
>>> print(innerhalb_x(c1, c2))
True
```

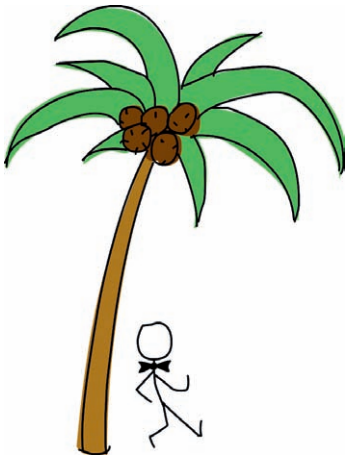
Die Funktion gibt `True` zurück. Was die Fähigkeit betrifft zu erkennen, ob ein Sprite gegen einen anderen gestoßen ist, ist dies der erste Schritt. Sobald wir eine Klasse für Herrn Strichmann und die Ebenen erzeugt haben, können wir sagen, ob sich deren x-Koordinaten überlappt haben.

Es ist kein besonders guter Programmierstil, jede Menge `if`-oder `elif`-Anweisungen zu haben, die den gleichen Wert zurückgeben. Dieses Problem können wir lösen, indem wir die `innerhalb_x`-Funktion kürzen. Dazu setzen wir jede ihrer Bedingungen in Klammern und trennen sie durch das Schlüsselwort `or`. Falls Du also eine etwas schickere Funktion mit ein paar weniger Code-Zeilen haben möchtest, kannst Du die Funktion folgendermaßen ändern:

```
def innerhalb_x(co1, co2):
    if (co1.x1 > co2.x1 and co1.x1 < co2.x2) \
            or (co1.x2 > co2.x1 and co1.x2 < co2.x2) \
            or (co2.x1 > co1.x1 and co2.x1 < co1.x2) \
            or (co2.x2 > co1.x1 and co2.x2 < co1.x1):
        return True
    else:
        return False
```

Wie weiter oben schon erklärt wurde, benutzen wir einen Rückwärtsschrägstrich (\), damit sich unsere `if`-Anweisung über mehrere Zeilen erstrecken darf. Ansonsten müsste sie in einer sehr, sehr langen Zeile stehen.

Sprites stoßen vertikal zusammen



Wir müssen auch noch wissen, ob die Sprites in vertikaler Richtung zusammenstoßen. Die Funktion `innerhalb_y` ist der Funktion `innerhalb_x` sehr ähnlich. Um sie zu erzeugen, prüfen wir, ob sich die `y1`-Position der ersten Koordinate mit den `y1`- und `y2`-Positionen der zweiten überlappt und umgekehrt. Dazu musst Du noch folgende Funktion hinzufügen (schreibe sie unter die `innerhalb_x`-Funktion). Und dieses Mal schreiben wir gleich die kürzere Version des Codes (anstelle der vielen `if`-Anweisungen):

```
def innerhalb_y(co1, co2):
    if (co1.y1 > co2.y1 and co1.y1 < co2.y2) \
            or (co1.y2 > co2.y1 and co1.y2 < co2.y2) \
            or (co2.y1 > co1.y1 and co2.y1 < co1.y2) \
            or (co2.y2 > co1.y1 and co2.y2 < co1.y1):
        return True
    else:
        return False
```

Alles zusammenfügen: Unserer endgültiger Code zur Erkennung von Zusammenstößen

Sobald wir festgestellt haben, ob sich ein Satz unserer `x`-Koordinaten mit einem anderen überlappt, und das Gleiche für die `y`-Koordinaten getan haben, können wir Funktionen schreiben, mit denen wir feststellen, ob ein Sprite den anderen berührt hat, und wenn ja, auf welcher Seite. Das machen wir mit den Funktionen `angestoßen_links`, `angestoßen_rechts`, `angestoßen_oben` und `angestoßen_unten`.

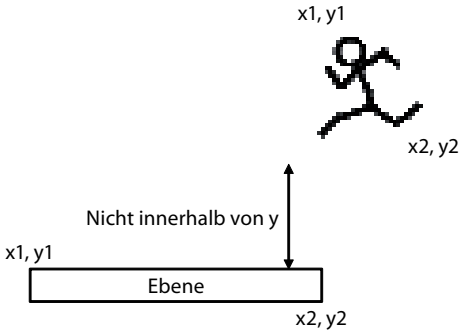
Die Funktion »angestoßen_links«

Hier ist der Code für die Funktion `angestoßen_links`, den Du unter die beiden `innerhalb`-Funktionen schreibst, die wir gerade erzeugt haben:

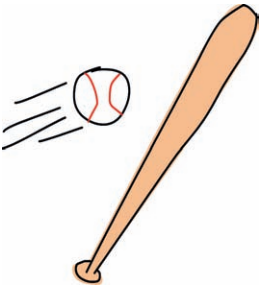
```
❶ def angestoßen_links(co1, co2):
❷     if innerhalb_y(co1, co2):
❸         if co1.x1 <= co2.x2 and co1.x1 >= co2.x1:
❹             return True
❺     return False
```

Diese Funktion sagt uns, ob die linke Seite (der `x1`-Wert) des ersten Koordinaten-Objekts ein anderes Koordinaten-Objekt berührt hat.

Die Funktion nimmt zwei Parameter auf: `co1` (das erste Koordinaten-Objekt) und `co2` (das zweite Koordinaten-Objekt). Wie Du in ❶ sehen kannst, prüfen wir, ob sich die beiden Koordinaten-Objekte vertikal überlappen, indem wir in ❷ die `innerhalb_y`-Funktion einsetzen. Schließlich ist es sinnlos zu prüfen, ob Herr Strichmann eine Ebene berührt hat, wenn er weit über ihr schwebt:



In ❸ prüfen wir, ob der Wert der äußersten linken Position des ersten Koordinaten-Objekts (`co.x1`) die `x2`-Position des zweiten Koordinaten-Objekts (`co2.x2`) berührt hat – das heißt, ob sie weniger oder gleich der `x2`-Position ist. Wir prüfen auch, ob sie nicht schon über die `x1`-Position herausragt. Falls sie die Seite berührt hat, geben wir in ❹ `True` (wahr) zurück. Falls keine der `if`-Anweisungen zutrifft, geben wir in ❺ `False` (falsch) zurück.



Die Funktion »angestoßen_rechts«

Die Funktion `angestoßen_rechts` sieht fast genauso aus wie `angestoßen_links`:

```
    def angestoßen_links(co1, co2):
❶       if innerhalb_y(co1, co2):
❷           if co1.x2 <= co2.x1 and co1.x2 >= co2.x2:
❸               return True
❹       return False
```

Wie schon bei der Funktion `angestoßen_links` prüfen wir in ❶ mit der Funktion `innerhalb_y`, ob sich die `y`-Koordinaten überlappen. Anschließend überprüfen wir in ❷, ob der `x2`-Wert zwischen den `x1`- und `x2`-Positionen des zweiten Koordinaten-Objekts liegt, und geben (falls ja) in ❸ `True` zurück. Ansonsten geben wir in ❹ `False` zurück.

Die Funktion »angestoßen_oben«

Die Funktion `angestoßen_oben` ähnelt stark den beiden Funktionen, die wir gerade hinzugefügt haben.

```
    def angestoßen_oben(co1, co2):
❶       if innerhalb_x(co1, co2):
❷           if co1.y1 <= co2.y2 and co1.y1 >= co2y1:
                return True
        return False
```

Diesmal besteht der Unterschied darin, dass wir mit der `innerhalb_x`-Funktion in ❶ prüfen, ob sich die Koordinaten horizontal überlappen. Als Nächstes schauen wir in ❷, ob die oberste Position der ersten Koordinate (`co1.y1`) sich mit der `y2`-Position der zweiten, nicht aber mit deren `y1`-Position überlappt. Falls sie es tut, geben wir `True` zurück (was bedeutet, dass die erste Koordinate die zweite berührt hat).

Die Funktion »angestoßen_unten«

Dir war natürlich schon klar, dass eine dieser vier Funktionen ein wenig anders sein würde, und sie ist es auch. Hier ist nun die Funktion `angestoßen_unten`:

```
    def angestoßen_unten(y, co1, co2):
❶       if innerhalb_x(co1, co2):
❷           y_calc = co1.y2 + y
❸           if y_calc >= co2.y1 and y_calc <= co2.y2:
❹               return True
❺       return False
```

Diese Funktion nimmt einen weiteren Parameter `y` auf, einen Wert, den wir der `y`-Position der ersten Koordinate hinzufügen. In ❶ schauen wir, ob sich die Koordinaten horizontal überlappen (wie wir es schon bei `angestoßen_oben` getan haben). Als Nächstes fügen wir den Wert des `y`-Parameters zu der `y2`-Position der Koordinate hinzu und speichern das Ergebnis in der Variable `y_calc` in ❷. Falls in ❸ der neu berechnete Wert zwischen den `y1`- und `y2`-Werten der zweiten Koordinate liegt, geben wir in ❹ `True` zurück, da die Unterseite der Koordinate `co1` die Oberseite der Koordinate `co2` berührt hat. Falls jedoch keine der `if`-Anweisungen zutrifft, geben wir in ❺ `False` zurück.

Da Herr Strichmann von einer Ebene fallen könnte, benötigen wir einen weiteren `y`-Parameter. Im Gegensatz zu den anderen `angestoßen`-Funktionen müssen wir testen können, ob er auf den Boden prallen *würde*, statt ob er es bereits getan hat. Wenn er von einer Ebene herunterlaufen und weiter in der Luft schweben würde, wäre unser Spiel ziemlich unrealistisch. Während er also läuft, prüfen wir daher, ob er mit etwas links oder rechts zusammengestoßen ist. Wenn wir dage-

gen unter ihm prüfen, schauen wir, ob er an die Ebene stößt; falls nicht, muss er herunterfallen!

17.5 Erzeugen der Sprite-Klasse

Die Elternklasse für unsere Spiel-Elemente werden wir `Sprite` nennen. Diese Klasse wird zwei Funktionen bereitstellen: `move`, um den Sprite zu bewegen, und `Koordinaten`, um die aktuelle Position des Sprites auf dem Monitor zurückzugeben. Hier ist der Code für die `Sprite`-Klasse:

```
    class Sprite:
❶       def __init__(self, spiel):
❷           self.spiel = spiel
❸           self.spielende = False
❹           self.koordinaten = None
❺       def move(self):
❻           pass
❼       def koords(self):
❽           return self.koordinaten
```

Die in ❶ definierte `__init__`-Funktion der `Sprite`-Klasse nimmt einen einzigen Parameter auf: `spiel`. Bei diesem Parameter handelt es sich um das Objekt `spiel`. Dieses Objekt brauchen wir, damit jeder Sprite, den wir erzeugen, auf die Liste der anderen Sprites im Spiel zugreifen kann. In ❷ speichern wir den `Spiel`-Parameter als Objekt-Variable.

In ❸ speichern wir die Objekt-Variable `spielende`, die wir verwenden, um das Ende des Spiels anzuzeigen (in dem Moment, wenn es auf `False` gesetzt wird). Die letzte Objekt-Variable, `koordinaten` in ❹, wird auf Nichts (`None`) gesetzt.

Die in ❺ definierte Funktion `move` macht in dieser Elternklasse nichts, sodass wir das Schlüsselwort `pass` in ❻ in diesem Funktionskörper benutzen. Die Funktion `koords` in ❼ gibt einfach in ❽ die Objekt-Variable `koordinaten` zurück.

Unsere `Sprite`-Klasse hat also eine Funktion `move`, die nichts macht, und eine Funktion `koords`, die keine Koordinaten zurückgibt. Das klingt nicht gerade sinnvoll, oder? Wir wissen jedoch, dass jede Klasse, die `Sprite` als Elternklasse hat, immer die Funktionen `move` und `koords` enthält. Wenn wir also in der Hauptschleife des Spiels durch die Liste der Sprites laufen, können wir die Funktion `move` aufrufen, ohne Fehlermeldungen zu produzieren. Warum? Weil jeder Sprite diese Funktion enthält.

Achtung!

Klassen mit Funktionen, die nicht sehr viel tun, kommen beim Programmieren häufig vor. Sie sind in gewisser Hinsicht eine Art Absprache oder Vertrag, durch den sichergestellt ist, dass alle Kinder einer Klasse die gleiche Funktionalität aufweisen, auch wenn in einigen Fällen die Funktionen der Kinderklassen nichts machen.

17.6 Die Ebenen hinzufügen

Jetzt kommen wir zu den Ebenen. Wir werden die Klasse für unser Ebenenobjekt `EbenenSprite` nennen, und sie wird eine Kinderklasse von `Sprite` sein. Die `__init__`-Funktion dieser Klasse wird einen `Spiel`-Parameter (genau wie die Elternklasse `Sprite`), ein Bild, `x`- und `y`-Positionen sowie die Breite und Höhe des Bildes aufnehmen. Hier ist der Code für die Klasse `EbenenSprite`:

```
❶ class EbenenSprite(Sprite):
❷     def __init__(self, spiel, photo_image, x, y, width, height):
❸         Sprite.__init__(self, spiel)
❹         self.photo_image = photo_image
❺         self.image = spiel.canvas.create_image(x, y, \
                  image=self.photo_image, anchor='nw')
❻         self.koordinaten = Koordinaten(x, y, x + width, y + height)
```

Wenn wir in ❶ die Klasse `EbenenSprite` definieren, geben wir ihr einen einzigen Parameter: den Namen ihrer Elternklasse (`Sprite`). Die `__init__`-Funktion in ❷ hat sieben Parameter: `self`, `spiel`, `photo_image`, `x`, `y`, `width` (Breite) und `height` (Höhe).

In ❸ rufen wir die `__init__`-Funktion der Elternklasse (`Sprite`) auf und benutzen `self` und `spiel` als Parameter-Werte, weil die `__init__`-Funktion der `Sprite`-Klasse außer dem Schlüsselwort `self` nur einen Parameter aufnimmt: `spiel`.

Wenn wir zu diesem Zeitpunkt ein `EbenenSprite`-Objekt erzeugen würden, hätte es alle Objekt-Variablen seiner Elternklasse (`spiel`, `spielende` und `koordinaten`), weil wir die `__init__`-Funktion in `Sprite` aufgerufen haben.

In ❹ speichern wir den Parameter `photo_image` als Objekt-Variable, und in ❺ benutzen wir die Variable `canvas` des Objekts `spiel`, um das Bild mit `create_image` zu zeichnen.

Zum Schluss erstellen wir ein `Koordinaten`-Objekt mit den Parametern `x` und `y` als seine beiden ersten Argumente. Anschließend fügen wir noch die Parameter `width` und `height` für die beiden Argumente in ❻ hinzu.

Obwohl die Variable `koordinaten` in der Elternklasse `Sprite` auf `None` gesetzt ist, haben wir sie in unserer Kinderklasse `EbenenSprite` in ein echtes `Koordinaten`-

Objekt umgewandelt, da sie den tatsächlichen Aufenthaltsort des Ebenen-Bildes auf dem Monitor enthält.

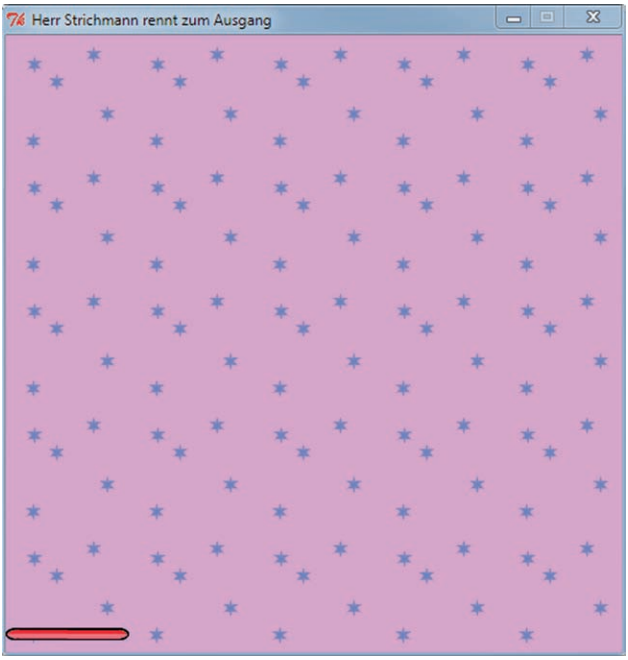
Ein Ebenen-Objekt hinzufügen

Fügen wir dem Spiel jetzt eine Ebene hinzu, um zu sehen, wie es aussieht. Ändere dazu die letzten beiden Zeilen der Spieldatei (*Strichmännchenspiel.py*) wie folgt:

```
❶ s = Spiel()
❷ Ebene1 = EbenenSprite(s, PhotoImage(file="Ebene1.gif"), \
      0, 480, 100, 10)
❸ s.sprites.append(ebene1)
❹ s.Hauptschleife()
```

Wie Du siehst, wurden die Zeilen ❶ und ❹ nicht verändert, aber in ❷ haben wir ein Objekt in der Klasse `EbenenSprite` erzeugt und ihm die Variable unseres Spiels (`s`) zusammen mit dem `PhotoImage`-Objekt (das unser erstes Ebenen-Bild *Ebene1.gif* verwendet) übergeben. Wir übergeben ihm auch die Position, auf der wir die Ebene zeichnen wollen (0 Pixel zur Seite und 480 Pixel nach unten, also fast am Boden der Leinwand), sowie die Höhe und Breite unseres Bilds (100 Pixel zur Seite und 10 Pixel hoch). In ❸ fügen wir diesen Sprite der Liste von Sprites in unserem Objekt `spiel` hinzu.

Wenn Du das Spiel jetzt laufen lässt, solltest Du eine Ebene sehen, die unten links im Fenster gezeichnet wird:

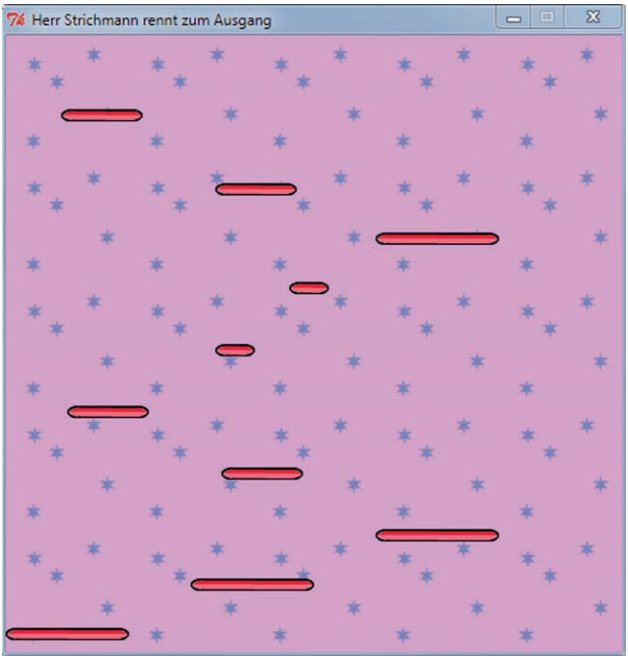


Einen ganzen Haufen Ebenen hinzufügen

Jetzt fügen wir einen ganzen Haufen von Ebenen hinzu. Jede Ebene wird unterschiedliche x- und y-Positionen haben, sodass sie über das ganze Fenster verteilt werden. Hier ist der Code dazu:

```
s = Spiel()
Ebene1 = EbenenSprite(s, PhotoImage(file="Ebene1.gif"), \
    0, 480, 100, 10)
Ebene2 = EbenenSprite(s, PhotoImage(file="Ebene1.gif"), \
    150, 440, 100, 10)
Ebene3 = EbenenSprite(s, PhotoImage(file="Ebene1.gif"), \
    300, 400, 100, 10)
Ebene4 = EbenenSprite(s, PhotoImage(file="Ebene1.gif"), \
    300, 160, 100, 10)
Ebene5 = EbenenSprite(s, PhotoImage(file="Ebene2.gif"), \
    175, 350, 66, 10)
Ebene6 = EbenenSprite(s, PhotoImage(file="Ebene2.gif"), \
    50, 300, 66, 10)
Ebene7 = EbenenSprite(s, PhotoImage(file="Ebene2.gif"), \
    170, 120, 66, 10)
Ebene8 = EbenenSprite(s, PhotoImage(file="Ebene2.gif"), \
    45, 60, 66, 10)
Ebene9 = EbenenSprite(s, PhotoImage(file="Ebene3.gif"), \
    170, 250, 32, 10)
Ebene10 = EbenenSprite(s, PhotoImage(file="Ebene3.gif"), \
    230, 200, 32, 10)
s.sprites.append(Ebene1)
s.sprites.append(Ebene2)
s.sprites.append(Ebene3)
s.sprites.append(Ebene4)
s.sprites.append(Ebene5)
s.sprites.append(Ebene6)
s.sprites.append(Ebene7)
s.sprites.append(Ebene8)
s.sprites.append(Ebene9)
s.sprites.append(Ebene10)
s.Hauptschleife()
```

Wir erzeugen jede Menge `EbenenSprite`-Objekte und speichern sie als Variablen `Ebene1`, `Ebene2`, `Ebene3` usw. bis zu `Ebene10`. Anschließend fügen wir jede Ebene der Variable `Sprites` hinzu, die wir in unserer `Spiel`-Klasse erzeugt haben. Wenn Du das Spiel jetzt laufen lässt, sollte es in etwa so aussehen:



Wir haben die Grundlagen für unser Spiel gelegt! Jetzt können wir unsere Hauptfigur, Herrn Strichmann, hinzufügen.

17.7 Was Du gelernt hast

In diesem Kapitel hast Du die `Spiel`-Klasse erzeugt und wie eine Art Tapete den Hintergrund auf den Monitor gezeichnet. Du hast gelernt, wie man mit den Funktionen `innerhalb_x` und `innerhalb_y` bestimmt, ob sich eine horizontale oder vertikale Position innerhalb zweier anderer horizontaler oder vertikaler Positionen befindet. Diese Funktionen hast Du anschließend verwendet, um neue Funktionen zu erstellen, mit denen man bestimmt, ob ein Koordinaten-Objekt mit einem anderen zusammengestoßen ist. Diese Funktionen werden wir im nächsten Kapitel verwenden, wenn wir Herrn Strichmann animieren und feststellen müssen, ob er an eine Ebene gestoßen ist, während er sich über die Leinwand bewegt hat.

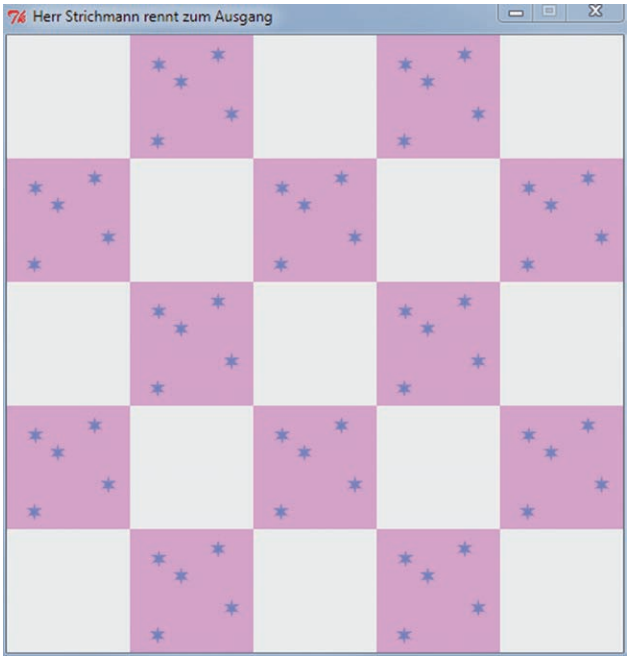
Wir haben ebenso die Elternklasse `Sprite` und deren erste Kinderklasse `EbenenSprite` erzeugt, mit der wir die Ebenen auf die Leinwand gezeichnet haben.

17.8 Programmier-Puzzles

In den folgenden Programmier-Puzzles kannst Du ein wenig mit dem Spielhintergrundbild experimentieren. Deine Lösung kannst Du unter *www.dpunkt.de/python* überprüfen.

#1: Schachbrett

Versuche, die `Spiel`-Klasse so zu ändern, dass das Hintergrundbild wie ein Schachbrett gezeichnet wird:

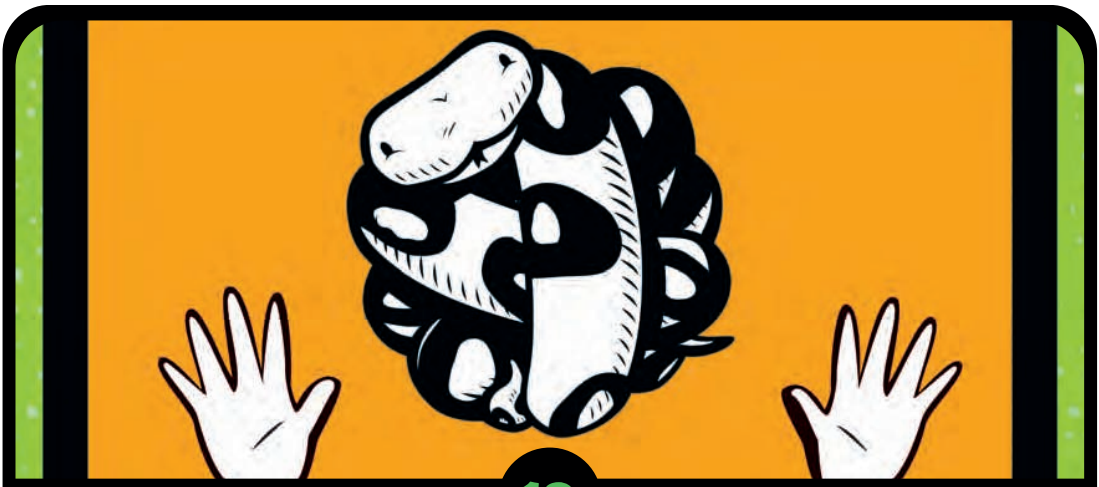


#2: Zwei-Bilder-Schachbrett

Wenn Du herausgefunden hast, wie man den Schachbrett-Effekt erzeugt, versuche es mit zwei abwechselnden Bildern. Erstelle ein zweites Hintergrundbild (mit Deinem Grafikprogramm), und ändere die `Spiel`-Klasse so, dass sie ein Schachbrettmuster aus zwei sich abwechselnden Bildern (statt eines Bildes und dem leeren Hintergrund) darstellt.

#3: Regal und Lampe

Damit das Bild hübscher aussieht, kannst Du andere Hintergrundbilder erstellen. Erzeuge dazu eine Kopie des Hintergrundbildes, und zeichne darauf ein einfaches Regal. Oder Du könntest einen Tisch mit einer Lampe oder ein Fenster dazu zeichnen. Verteile die Bilder durch Ändern der `Spiel`-Klasse auf dem Bildschirm, sodass sie drei oder vier verschiedene Hintergrundbilder lädt und anzeigt.



18 Herrn Strichmann erschaffen

In diesem Kapitel werden wir die Hauptfigur unseres Spiels *Herr Strichmann rennt zum Ausgang* erschaffen. Da Herr Strichmann nach links und rechts laufen und springen muss, anhalten soll, wenn er gegen eine Ebene läuft, und hinfallen soll, wenn er über die Kante einer Ebene läuft, wird dies der komplizierteste Code, den wir bis jetzt geschrieben haben. Um das Strichmännchen nach links und rechts laufen zu lassen, werden wir mit Ereignisbindungen an die linke und rechte Pfeiltaste arbeiten, und wir werden es durch Druck auf die Leertaste springen lassen.

18.1 Das Strichmännchen initialisieren

Die `__init__`-Funktion unserer neuen Strichmännchen-Klasse hat viel Ähnlichkeit mit den anderen Klassen, die wir bis jetzt erstellt haben. Wir geben unserer neuen Klasse zunächst einen Namen: `StrichFigurSprite`. Wie schon bei den vorherigen Klassen hat auch diese eine Elternklasse: `Sprite`.

```
class StrichFigurSprite:
    def __init__(self, spiel):
        Sprite.__init__(self, spiel)
```

Dieser Code sieht aus wie der Code, den wir in der `EbenenSprite`-Klasse in Kapitel 16 geschrieben haben, nur dass wir keine zusätzlichen Parameter verwenden (außer `self` und `spiel`). Dies liegt daran, dass wir im Gegensatz zur `EbenenSprite`-Klasse nur ein einziges `StrichFigurSprite`-Objekt im Spiel einsetzen.

Die Strichmännchen-Bilder laden

Da wir eine Menge Ebenen-Objekte auf dem Monitor haben, die jeweils unterschiedlich große Bilder verwenden können, übergeben wir das Ebenen-Bild als Parameter der `__init__`-Funktion der `Ebenen-Sprites`. (Das ist, als ob man sagen würde: »Ebenen-Sprite, nimm dieses Bild, um Dich selbst auf dem Monitor zu zeichnen.«) Da es aber nur ein einziges Strichmännchen auf dem Monitor gibt, ist es nicht sinnvoll, das Bild außerhalb des Sprites zu laden und es dann als Parameter zu übergeben. Die Klasse `StrichFigurSprite` wird daher ihre eigenen Bilder laden.

Die nächsten paar Zeilen der `__init__`-Funktion tun genau das: Sie laden die drei linken Bilder (die wir verwenden werden, um das Strichmännchen nach links laufen zu lassen) und die drei rechten Bilder (um das Strichmännchen nach rechts laufend zu animieren). Wir müssen die Bilder schon an dieser Stelle laden, weil sie nicht erst immer dann geladen werden sollen, wenn das Strichmännchen auf dem Monitor dargestellt werden soll (dies würde zu lange dauern und das Spiel sehr verlangsamen).

```
    class StrichFigurSprite:
        def __init__(self, spiel):
            Sprite.__init__(self, spiel)
❶           self.bilder_links = [
                PhotoImage(file="Figur-L1.gif"),
                PhotoImage(file="Figur-L2.gif"),
                PhotoImage(file="Figur-L3.gif")
            ]
❷           self.bilder_rechts = [
                PhotoImage(file="Figur-R1.gif"),
                PhotoImage(file="Figur-R2.gif"),
                PhotoImage(file="Figur-R3.gif")
            ]
❸           self.bild = spiel.canvas.create_image(200, 470, \
                    image=self.bilder_links[0], anchor='nw')
```

Dieser Code lädt jedes der drei linken Bilder, mit denen wir das Strichmännchen nach links laufen lassen, und die drei Bilder, mit denen wir die Animation des Strichmännchens nach rechts vornehmen.

In ❶ und ❷ erzeugen wir die Objekt-Variablen `bilder_links` und `bilder_rechts`. Beide enthalten eine Liste mit den `PhotoImage`-Objekten, die wir in Kapitel 16 erzeugt haben und in denen das Strichmännchen nach links oder rechts zeigt. Mit `bilder_links[0]` malen wir in ❸ das erste Bild mit der Funktion der Leinwand `create_image` an der Position (200, 470), wodurch das Strichmänn-

chen in der Mitte des Spielfensters und am Boden der Leinwand erscheint. Die Funktion `create_image` gibt eine Zahl zurück, die das Bild auf der Leinwand identifiziert. Wir speichern diese ID-Nummer für später in der Objekt-Variablen `image`.

Variablen einrichten

Im nächsten Teil der `__init__`-Funktion werden weitere Variablen eingerichtet, die wir später im Code verwenden.

```
        self.bilder_rechts = [
            PhotoImage(file="Figur-R1.gif"),
            PhotoImage(file="Figur-R2.gif"),
            PhotoImage(file="Figur-R3.gif")
        ]
        self.bild = spiel.canvas.create_image(200, 470, \
            image=self.bilder_links[0], anchor='nw')
❶       self.x = -2
❷       self.y = 0
❸       self.aktuelles_bild = 0
❹       self.aktuelles_bild_plus = 1
❺       self.springen_zähler = 0
❻       self.letzte_zeit = time.time()
❼       self.koordinaten = Koordinaten()
```

In ❶ und ❷ speichern die Variablen `x` und `y` den Umfang der Koordinaten, den die Strichmännchen horizontal (`x1` und `x2`) und vertikal (`y1` und `y2`) beim Bewegen auf dem Monitor zurücklegen.

Wie Du schon in Kapitel 14 gelernt hast, fügen wir zu den `x`- und `y`-Positionen der Objekte Werte hinzu, um sie mit dem Modul `tkinter` über die Leinwand zu bewegen. Indem wir x auf -2 und y auf 0 setzen, ziehen wir von der x-Position später 2 ab und fügen der vertikalen Position nichts hinzu, damit das Strichmännchen nach links rennt.

Achtung!

Denke daran, dass eine negative x-Zahl eine Bewegung nach links auf der Leinwand verursacht und dass eine positive x-Zahl eine Bewegung nach rechts bedeutet. Eine negative y-Zahl steht für eine Bewegung nach oben und eine positive y-Zahl für eine Bewegung nach unten.

In ❸ erzeugen wir die Objekt-Variable `aktuelles_bild`, um die Index-Position des Bildes, wie es gerade auf dem Monitor angezeigt wird, zu speichern. Unsere Liste von Bildern, die nach links zeigen (`bilder_links`), enthält *Figur-L1.gif*, *Figur-L2.gif* und *Figur-L3.gif*. Diese haben die Index-Positionen 0, 1 und 2.

In ❹ enthält die Variable `aktuelles_bild_plus` die Zahl, die wir der in `aktuelles_bild` gespeicherten Index-Position hinzufügen, um die nächste Index-Position zu erhalten. Wenn das Bild beispielsweise an Index-Position 0 angezeigt wird, zählen wir 1 hinzu, um das nächste Bild an Index-Position 1 zu erhalten, und zählen wieder 1 hinzu, um das letzte Bild der Liste an Index-Position 2 zu bekommen. (Wie man diese Variable zur Animation nutzt, wirst Du im nächsten Kapitel sehen.)

Die Variable `springen_zähler` in ❺ ist ein Zähler, den wir verwenden, während das Strichmännchen springt. Die Variable `letzte_zeit` speichert den letzten Zeitpunkt, an dem wir das Bild während der Animation unseres Strichmännchens verändert haben. Die aktuelle Zeit speichern wir mit der Funktion `time` aus dem Modul `time` in ❻.

In ❼ setzen wir die Objekt-Variable `koordinaten` auf ein Objekt der `Koordinaten`-Klasse, und zwar ohne Initialisierungsparameter (`x1`, `y1`, `x2` und `y2` sind alle 0). Im Gegensatz zu den Ebenen verändern sich die Koordinaten des Strichmännchens, sodass wir deren Werte später setzen.

Bindung an die Tasten

Im letzten Teil der `__init__`-Funktion verbindet die Funktion `bind` eine Taste mit dem Teil unseres Codes, der durchgeführt werden soll, sobald die entsprechende Taste gedrückt wird.

```
        self.springen_zähler = 0
        self.letzte_zeit = time.time()
        self.koordinaten = Koordinaten()
        game.canvas.bind_all('<KeyPress-Left>', self.nach_links)
        game.canvas.bind_all('<KeyPress-Right>', self.nach_rechts)
        game.canvas.bind_all('<space>', self.springen)
```

Wir verbinden `'<KeyPress-Left>'` mit der Funktion `nach_links`, `'<KeyPress-Right>'` mit der Funktion `nach_rechts` und `<space>` mit der Funktion `springen`. Jetzt müssen wir diese Funktionen erzeugen, damit sich das Strichmännchen bewegt.

18.2 Das Strichmännchen nach links und rechts bewegen

Die Funktionen `nach_links` und `nach_rechts` sorgen dafür, dass das Strichmännchen nicht springt, und setzen den Wert der Objekt-Variablen `x` so, dass es sich nach links oder rechts bewegt. (Falls unsere Figur springt, würde unser Spiel es nicht ermöglichen, die Richtung in der Luft zu ändern.)

```
        game.canvas.bind_all('<KeyPress-Left>', self.nach_links)
        game.canvas.bind_all('<KeyPress-Right>', self.nach_rechts)
        game.canvas.bind_all('<space>', self.springen)

❶   def nach_links(self, evt):
❷       if self.y == 0:
❸           self.x = -2

❹   def nach_rechts(self, evt):
❺       if self.y == 0:
❻           self.x = 2
```

Sobald der Spieler die linke Pfeiltaste drückt, ruft Python die Funktion `nach_links` auf und übergibt ein Objekt, das als Parameter Informationen darüber enthält, was der Spieler gemacht hat. So ein Objekt nennt man *Ereignisobjekt*, und wir geben ihm den Parameter-Namen `evt`.

Achtung!

Das Ereignisobjekt ist für unsere Zwecke nicht wichtig, muss aber als Parameter unserer Funktionen (in ❶ und ❹) enthalten sein, da Python es dort erwartet und ansonsten eine Fehlermeldung ausgibt. Ein Ereignis-Objekt enthält normalerweise Dinge wie x- und y-Positionen der Maus (Maus-Ereignis), Code, der für eine bestimmte Taste steht (Tastatur-Ereignis), und andere Informationen. Bei diesem Spiel nützt uns keine dieser Informationen etwas, sodass wir sie einfach ignorieren können.

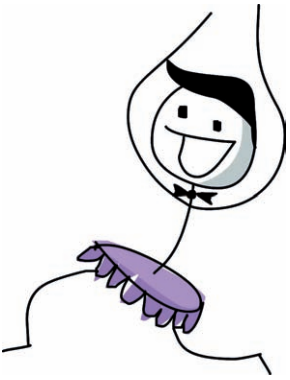
Um zu prüfen, ob das Strichmännchen gerade springt, prüfen wir in ❷ und ❺ die `y`-Objekt-Variable. Falls der Wert nicht 0 ist, springt das Strichmännchen. Wenn in diesem Fall der Wert von `y` auf 0 steht, setzen wir `x` auf -2, um in ❸ nach links zu rennen, oder wir setzen ihn in ❻ auf 2, um nach rechts zu rennen. Mit den Werten -1 oder 1 würde sich das Strichmännchen nicht schnell genug über den Monitor bewegen. (Sobald die Animation Deines Strichmännchens funktioniert, versuche einmal diesen Wert zu ändern, um den Unterschied zu sehen.)

18.3 Das Strichmännchen springen lassen

Die Funktion `springen` ist den Funktionen `nach_links` und `nach_rechts` sehr ähnlich.

```
    def nach_rechts(self, evt):
        if self.y == 0:
            self.x = 2

    def springen(self, evt):
❶       if self.y == 0:
❷           self.y = -4
❸           self.springen_zähler = 0
```

Diese Funktion nimmt den Parameter `evt` (das Ereignis-Objekt) auf, den wir ignorieren können, da wir keine weitere Informationen über das Ereignis benötigen. Wenn diese Funktion aufgerufen wird, wissen wir, dass die Leertaste gedrückt wurde.



Da wir möchten, dass unser Strichmännchen nur springt, falls es das gerade nicht tut, prüfen wir in ❶, ob `y` gleich 0 ist. Falls das Strichmännchen nicht springt, setzen wir in ❷ `y` auf -4 (um es vertikal nach oben zu bewegen) und setzen `springen_zähler` in ❸ auf 0. Den `springen_zähler` benutzen wir, um sicherzustellen, dass das Strichmännchen nicht endlos springt. Stattdessen lassen wir es eine bestimmte Anzahl Sprünge machen und lassen es dann so herunterkommen, als ob die Schwerkraft es wieder herunterziehen würde. Den Code dafür fügen wir im nächsten Kapitel hinzu.

18.4 Was wir bis jetzt erreicht haben

Wir wollen uns zunächst einen Überblick über die Definitionen der Klassen und Funktionen verschaffen, die wir bis jetzt in unserem Spiel haben, und schauen, wo sie sich in Deiner Datei befinden sollten.

Ganz oben in Deinem Programm sollten die `import`-Anweisungen stehen, gefolgt von den `Spiel`- und `Koordinaten`-Klassen. Die `Spiel`-Klasse wird zur Erzeugung eines Objekts verwendet, das als Hauptsteuerung Deines Spiels dient, und die Objekte der `Koordinaten`-Klasse sind dazu da, die Positionen der Dinge in Deinem Spiel zusammenzuhalten (wie etwa die Ebenen und Herrn Strichmann):

```
from tkinter import *
import random
import time
class Spiel:
    ...
class Koordinaten:
    ...
```

Als Nächstes sollten bei Dir die `innerhalb`-Funktionen kommen (die sagen, ob die Koordinaten eines Sprites »innerhalb« des Bereichs eines anderen Sprites liegen). Dann folgen die Elternklasse `Sprite` (die Elternklasse aller Sprites in unserem Spiel), die Klasse `EbenenSprite` und der Anfang der `StrichFigurSprite`-Klasse. Die Klasse `EbenenSprite` wurde zur Erzeugung von Ebenen-Objekten verwendet, über die unser Strichmännchen springen soll, und wir haben ein Objekt der `StrichFigurSprite`-Klasse erzeugt, das für die Hauptfigur in unserem Spiel steht:

```
def innerhalb_x(co1, co2):
    ...
def innerhalb_y(co1, co2):
    ...
class Sprite:
    ...
class EbenenSprite(Sprite):
    ...
class StrichFigurSprite(Sprite):
    ...
```

Am Ende Deines Programms sollte der Code stehen, der alle bis jetzt erstellten Objekte in Deinem Spiel erzeugt: das `Spiel`-Objekt an sich und die Ebenen. In der letzten Zeile steht das, was wir die Hauptschleifen-Funktion (`Hauptschleife`) nennen:

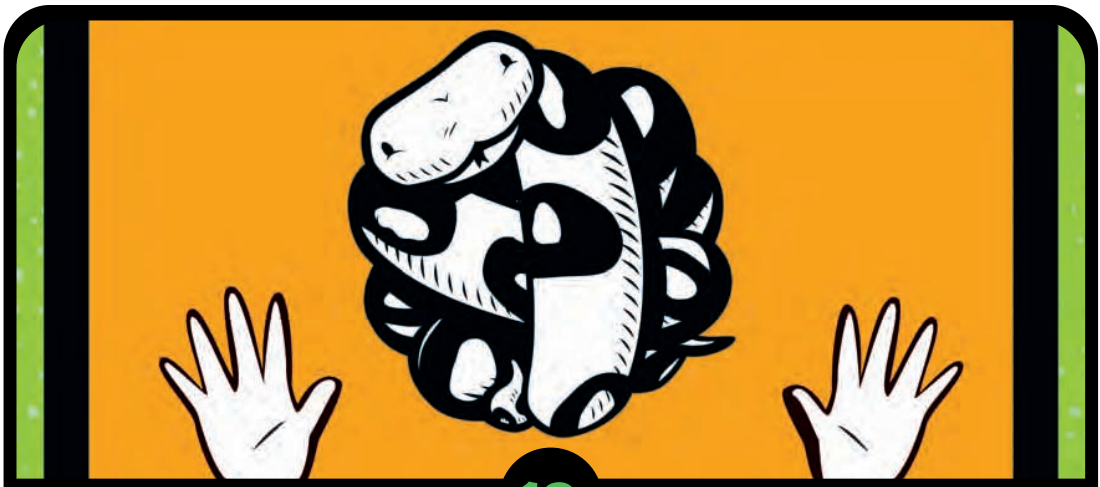
```
s = Spiel()
Ebene1 = EbenenSprite(s, PhotoImage(file="Ebene1.gif"), \
    0, 480, 100, 10)
...
s.sprites.append(ebene1)
...
s.Hauptschleife()
```

Falls Dein Code ein bisschen anders aussieht oder Du Schwierigkeiten hast, ihn ans Laufen zu bringen, kannst Du zum Ende von Kapitel 19 vorblättern, wo Du das vollständige Listing des gesamten Spiels findest.

18.5 Was Du gelernt hast

In diesem Kapitel haben wir mit der Arbeit an der Klasse für unser Strichmännchen begonnen. Wenn wir zu diesem Zeitpunkt ein Objekt dieser Klasse erzeugen würden, würde es nicht viel tun, außer die Bilder zu laden, die es für die Animation des Strichmännchens benötigt, und ein paar Objekt-Variablen für später einzurichten.

Diese Klasse enthält ein paar Funktionen zum Ändern der Werte in diesen Objekt-Variablen aufgrund von Tastatur-Ereignissen (wenn ein Spieler die linke oder rechte Pfeiltaste oder Leertaste drückt). Im nächsten Kapitel werden wir unser Spiel zu Ende programmieren. Wir werden die Funktionen zum Anzeigen und Animieren des Strichmännchens in der `StrichFigurSprite`-Klasse schreiben und Herrn Strichmann auf dem Monitor bewegen. Wir werden auch den Ausgang (die Tür) erzeugen, die er zu erreichen versucht.



19 Abschluss des Spiels mit Herrn Strichmann

In den letzten drei Kapiteln haben wir unser Spiel entwickelt: Herr Strichmann rennt zum Ausgang. Wir haben zunächst die Grafiken erstellt und danach den Code geschrieben, um das Hintergrundbild, die Ebenen und das Strichmännchen hinzuzufügen. In diesem Kapitel unternehmen wir die letzten Schritte, um das Strichmännchen zu animieren, und fügen die Tür hinzu.

Das vollständige Listing des Spiels findest Du am Ende dieses Kapitels. Falls Du nicht mehr weiter weißt oder Dir beim Schreiben dieses Codes etwas unklar ist, vergleiche Deinen Code mit dem Listing, um zu sehen, was Du falsch gemacht hast.

19.1 Animation des Strichmännchens

Bis jetzt haben wir die grundlegende Klasse für unser Strichmännchen erstellt, die Bilder geladen, die wir verwenden werden, und Tasten mit einigen Funktionen verbunden. Wenn Du das Spiel jetzt jedoch laufen lässt, macht unser Code nichts besonders Interessantes.

In den folgenden Abschnitten fügen wir der `StrichFigurSprite`-Klasse, die wir in Kapitel 18 erzeugt haben, die

restlichen Funktionen hinzu: `animieren`, `move` und `coords`. Die Funktion `animieren` wird die unterschiedlichen Strichmännchen-Bilder zeichnen; `move` wird festlegen, wohin sich die Figur bewegen soll, und `coords` wird die aktuelle Position des Strichmännchens zurückgeben. (Anders als bei den Ebenen-Sprites müssen wir die Position des Strichmännchens immer wieder neu berechnen, da es sich auf dem Bildschirm umher bewegt.)

Die Funktion animieren erstellen

Als Erstes erstellen wir die Funktion `animieren`, die wir benötigen, um die Bewegung zu erfassen und das Bild entsprechend zu ändern.

Bewegung erfassen

Wir möchten nicht, dass das Strichmännchen-Bild sich in unserer Animation zu schnell ändert, da seine Bewegungen ansonsten nicht realistisch aussehen würden. Stell Dir dazu einfach ein Daumenkino vor, das Du auf die Ecken eines Notizblocks gezeichnet hast – wenn Du die Seiten zu schnell blätterst, hast Du eventuell nicht den vollen Effekt.

Die erste Hälfte der Funktion `animieren` prüft, ob das Strichmännchen nach links oder rechts rennt. Sie benutzt die Variable `letzte_zeit`, um zu entscheiden, ob das aktuelle Bild verändert werden soll. Diese Variable wird uns dabei helfen, die Geschwindigkeit unserer Animation zu steuern. Die Funktion schreiben wir hinter die `springen`-Funktion, die wir in Kapitel 18 unserer `StrichFigurSprite`-Klasse hinzugefügt haben.

```
    def springen(self, evt):
        if self.y == 0:
            self.y = -4
            self.springen_zähler = 0

    def animieren(self):
❶       if self.x != 0 and self.y == 0:
❷           if time.time() - self.letzte_zeit > 0.1:
❸               self.letzte_zeit = time.time()
❹               self.aktuelles_bild += self.aktuelles_bild_plus
❺           if self.aktuelles_bild >= 2:
❻               self.aktuelles_bild_plus = -1
❼           if self.aktuelles_bild <= 0:
❽               self.aktuelles_bild_plus = 1
```

Mit der `if`-Anweisung in ❶ prüfen wir, ob `x` nicht 0 ist, um dadurch festzustellen, ob sich das Strichmännchen bewegt (nach links oder rechts), und wir schauen, ob `y` gleich 0 ist, um zu bestimmen, dass das Strichmännchen nicht springt. Falls diese `if`-Anweisung wahr ist, müssen wir unser Strichmännchen animieren; wenn

nicht, steht es still und muss nicht animiert werden. Wenn sich das Strichmännchen nicht bewegt, fallen wir aus dieser Funktion heraus, und der restliche Code dieses Listings wird ignoriert.

In ❷ berechnen wir die Zeit, die nach dem letzten Aufruf der Funktion `animieren` vergangen ist, indem wir den Wert der Variable `letzte_zeit` von der aktuellen Zeit mit `time.time()` abziehen. Diese Berechnung wird zur Entscheidung benötigt, ob das nächste Bild in der Sequenz gezeichnet werden soll oder nicht. Falls das Ergebnis größer als eine Zehntelsekunde (0.1) ist, geht es mit dem Codeblock in ❸ weiter. Die Variable `letzte_zeit` setzen wir auf die aktuelle Zeit. Dadurch setzen wir quasi die Stoppuhr zurück auf null für den nächsten Bildwechsel.

In ❹ fügen wir den Wert der Objekt-Variablen `aktuelles_bild_plus` der Variablen `aktuelles_bild` hinzu, die die Index-Position des aktuell angezeigten Bildes speichert. Da wir in Kapitel 18 die Variable `aktuelles_bild_plus` in der `__init__`-Funktion des Strichmännchens schon erzeugt haben, ist beim ersten Aufruf der Funktion `animieren` der Wert der Variable schon auf 1 gesetzt.

In ❺ prüfen wir, ob der Wert der Index-Position in `aktuelles_bild` größer oder gleich 2 ist. Falls ja, ändern wir den Wert von `aktuelles_bild_plus` in ❻ auf –1. Der Vorgang in ❼ ist ähnlich – sobald wir 0 erreichen, müssen wir mit dem Zählen von vorne anfangen, wie wir es in ❽ tun.

Achtung!

Falls Dir unklar ist, wie Du diesen Code einrücken sollst, hier ein Hinweis: Am Anfang von ❶ sind es 8 Leerzeichen, und vor ❽ sind es 20 Leerzeichen.

Damit Du besser verstehst, was in der Funktion bis jetzt passiert, stell Dir vor, Du hättest auf dem Boden eine Reihe farbiger Bauklötze ausgelegt. Du bewegst Deinen Finger von einem Bauklotz zum nächsten, und jeder Klotz, auf den Dein Finger zeigt (1, 2, 3, 4 usw.), hat eine Nummer (die Variable `aktuelles_bild`). Die Nummer des Platzes (Dein Finger zeigt immer auf einen einzigen Klotz) ist die Zahl, die in der Variablen `aktuelles_ bild_plus` gespeichert wird. Wenn sich Dein Finger die Reihe von Klötzen entlangbewegt, wird jedes Mal 1 hinzugezählt. Und wenn das Ende der Reihe erreicht ist und der Finger zurückwandert, wird jedes Mal 1 abgezogen (also -1 hinzugefügt).

Der Code, den wir unserer `animieren`-Funktion hinzugefügt haben, führt diesen Prozess durch, doch anstelle von farbigen Bauklötzen nimmt er die drei Strichmännchenbilder für beide Richtungen, die in einer Liste gespeichert sind. Die Index-Positionen dieser Bilder sind 0, 1 und 2. Sobald wir beim Animieren des Strichmännchens das letzte Bild erreicht haben, fangen wir an, rückwärts zu

zählen, und wenn wir wieder beim ersten Bild sind, müssen wir wieder vorwärts zählen. Als Ergebnis bekommen wir den Eindruck einer rennenden Figur.

Im Folgenden wird gezeigt, wie wir uns durch die Liste der Bilder bewegen, wobei wir die Index-Positionen in der `animieren`-Funktion berechnen.

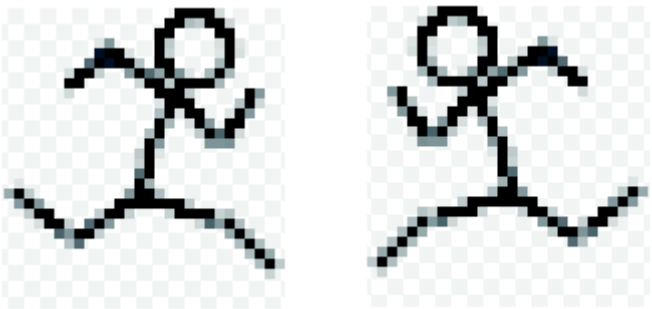
Position 0	Position 1	Position 2	Position 1	Position 0	Position 1
Aufwärts zählend	Aufwärts zählend	Aufwärts zählend	Abwärts zählend	Abwärts zählend	Aufwärts zählend
					

Das Bild ändern

In der nächsten Hälfte der Funktion `animieren` finden wir durch die berechnete Index-Position das jeweils angezeigte Bild.

```
        def animieren(self):
            if self.x != 0 and self.y == 0:
                if time.time() - self.letzte_zeit > 0.1:
                    self.letzte_zeit = time.time()
                    self.aktuelles_bild += self.aktuelles_bild_plus
                    if self.aktuelles__bild >= 2:
                        self.aktuelles_bild_plus = -1
                    if self.aktuelles__bild <= 0:
                        self.aktuelles_bild_plus = 1
❶           if self.x < 0:
❷               if self.y != 0:
❸                   self.spiel.canvas.itemconfig(self.bild, \
                            image=self.bilder_links[2])
❹               else:
❺                   self.spiel.canvas.itemconfig(self.bild, \
                            image=self.bilder_links[self.aktuelles_bild])
❻           elif self.x > 0:
❼               if self.y != 0:
❽                   self.spiel.canvas.itemconfig(self.bild, \
                            image=self.bilder_rechts[2])
❾               else:
❿                   self.spiel.canvas.itemconfig(self.bild, \
                            image=self.bilder_rechts[self.aktuelles_bild])
```

Wenn in ❶ `x` weniger als 0 beträgt, bewegt sich das Strichmännchen nach links und Python geht in den Code-Block von ❷ bis ❺, wo geprüft wird, ob `y` ungleich 0 ist (was bedeutet, dass das Strichmännchen springt). Falls `y` ungleich 0 ist (das Strichmännchen springt, bewegt sich also nach oben oder unten), benutzen wir die Leinwand-Funktion `item_config`, um das letzte Bild in unserer Liste mit nach links zeigenden Bildern in ❸ (`bilder_links[2]`) anzuzeigen. Da das Strichmännchen springt, zeigen wir es mit maximaler Schrittlänge, damit die Animation etwas realistischer aussieht:



Wenn das Strichmännchen nicht im Sprung ist (wenn `y` also gleich 0 ist), benutzt die `else`-Anweisung ab ❹ `item_config`, um das Bild in dasjenige zu ändern, dessen Indexposition der Variablen `aktuelles_bild` im Code bei ❺ entspricht.

In ❻ schauen wir, ob das Strichmännchen nach rechts rennt (`x` ist größer als 0), und Python geht dann weiter in den Block von ❼ bis ❿. Der Code ist dem Code aus dem ersten Block sehr ähnlich und prüft wieder, ob das Strichmännchen springt. Falls es springt, stellt er das entsprechende Bild dar oder verwendet die Bilder aus `bilder_rechts`.

Die Position des Strichmännchens erfassen

Da wir bestimmen müssen, wo sich das Strichmännchen auf dem Monitor befindet (weil es sich umherbewegt), unterscheidet sich die Funktion `coords` von den anderen Funktionen der `Sprite`-Klasse. Um zu erfassen, wo sich das Strichmännchen befindet, werden wir die Funktion `coords` der Leinwand verwenden und dann deren Werte benutzen, um die Werte x1, y1 sowie x2 und y2 der Variablen `koordinaten` zu setzen, die wir in der `__init__`-Funktion am Anfang von Kapitel 18 erzeugt haben. Hier siehst Du den Code, der hinter die `animieren`-Funktion gestellt werden kann:

```
            if self.x < 0:
                if self.y != 0:
                    self.spiel.canvas.itemconfig(self.bild, \
                            image=self.bilder_links[2])
                else:
                    self.spiel.canvas.itemconfig(self.bild, \
                            image=self.bilder_links[self.aktuelles_bild])
            elif self.x > 0:
                if self.y != 0:
                    self.spiel.canvas.itemconfig(self.bild, \
                            image=self.bilder_rechts[2])
                else:
                    self.spiel.canvas.itemconfig(self.bild, \
                            image=self.bilder_rechts[self.aktuelles_bild])

        def coords(self):
❶           xy = self.spiel.canvas.coords(self.bild)
❷           self.koordinaten.x1 = xy[0]
❸           self.koordinaten.y1 = xy[1]
❹           self.koordinaten.x2 = xy[0] + 27
❺           self.koordinaten.y2 = xy[1] + 30
            return self.koordinaten
```

Als wir in Kapitel 17 die `Spiel`-Klasse angelegt haben, war eine der Objekt-Variablen die Leinwand (`canvas`). In ❶ benutzen wir mit `self.spiel.canvas.coords` die `coords`-Funktion dieser `canvas`-Variablen, um die `x`- und `y`-Positionen des aktuellen Bildes zurückzugeben. Diese Funktion verwendet die Zahl, die in der Objekt-Variablen `bild` gespeichert ist, d.h. die ID-Nummer des Bildes, das auf die Leinwand gezeichnet wird.

Die daraus resultierende Liste speichern wir in der Variablen `xy`, die nun zwei Werte enthält: die `x`-Position oben links, die in der Variablen `x1` von `koordinaten` in ❷ gespeichert wird, und die y-Position oben links als Variable `y1` von `koordinaten` in ❸. Da alle Strichmännchen-Bilder 27 Pixel breit und 30 Pixel hoch sind, können wir bestimmen, wie die `x2`- und `y2`-Variablen sein sollten, indem wir die Breite in ❹ und die Höhe in ❺ zu den x- bzw. y-Zahlen addieren.

In der letzten Zeile der Funktion geben wir schließlich die Objekt-Variable `koordinaten` zurück.

Das Strichmännchen in Bewegung versetzen

Die letzte Funktion der Strichfigur-Sprite-Klasse `move` ist dafür zuständig, unsere Spielfigur tatsächlich über den Monitor zu bewegen. Sie muss uns auch sagen können, ob die Figur an etwas gestoßen ist.

Der Beginn der Funktion »move«

Hier siehst Du den Code für den ersten Teil der Funktion `move` – er folgt hinter `coords`:

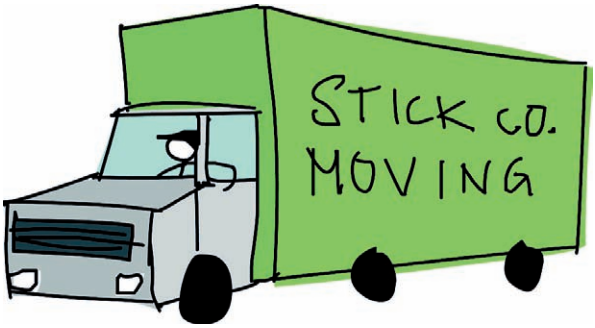
```
    def coords(self):
        xy = self.spiel.canvas.coords(self.image)
        self.koordinaten.x1 = xy[0]
        self.koordinaten.y1 = xy[1]
        self.koordinaten.x2 = xy[0] + 27
        self.koordinaten.y2 = xy[1] + 30
        return self.koordinaten

    def move(self):
❶       self.animieren()
❷       if self.y < 0:
❸           self.springen_zähler += 1
❹           if self.springen_zähler > 20:
❺               self.y = 4
❻       if self.y > 0:
❼           self.springen_zähler -= 1
```

Der Teil in ❶ dieser Funktion ruft die Funktion `animieren` auf, die wir vorher in diesem Kapitel erzeugt haben, und ändert falls nötig das aktuell angezeigte Bild. In ❷ schauen wir, ob der Wert von `y` kleiner als 0 ist. Falls ja, wissen wir, dass das Strichmännchen springt, da ein negativer Wert es nach oben bewegen würde, (Denk daran, dass 0 ganz oben auf der Leinwand ist und dass die Pixelposition 500 ganz unten liegt.)

In ❸ zählen wir zu `springen_zähler` 1 hinzu, und in ❹ bzw. ❺ sagen wir, dass wir `y` auf 4 ändern, sobald `springen_zähler` den Wert 20 erreicht hat, damit das Strichmännchen wieder herunterkommt.

In ❻ prüfen wir, ob der Wert von `y` größer als 0 ist (die Figur fällt herunter), und falls er es ist, ziehen wir 1 von `springen_zähler` ab ❼, da wir – nachdem wir bis 20 aufwärts gezählt haben – wieder rückwärts zählen müssen. (Hebe Deine Hand langsam nach oben, während Du bist 20 zählst, und lass sie wieder sinken, während Du von 20 rückwärts zählst. Dadurch bekommst Du ein Gefühl dafür, wie die Berechnung des Auf- und Abspringens der Figur funktioniert.)

In den nächsten paar Zeilen der Funktion `move` rufen wir die Funktion `coords` auf, die uns sagt, wo sich unsere Figur auf dem Monitor befindet, und das Ergebnis in der Variablen `co` speichert. Anschließend erzeugen wir die Variablen `links`, `rechts`, `oben`, `unten` und `fallen`. Diese werden wir alle im Rest dieser Funktion verwenden.




```
        if self.y > 0:
            self.springen_zähler -= 1
        co = self.coords()
        links = True
        rechts = True
        oben = True
        unten = True
        fallen = True
```

Beachte, dass alle Variablen auf den booleschen Wert wahr (True) gesetzt sind. Wir werden sie als Indikatoren verwenden, um zu prüfen, ob die Figur auf dem Monitor an etwas gestoßen ist oder gerade fällt.

Hat das Strichmännchen den Boden oder die Decke der Leinwand berührt?

Der nächste Abschnitt der Funktion `move` prüft, ob unsere Figur den Boden oder die Decke der Leinwand berührt hat. Hier ist der Code:

```
        unten = True
        fallen = True
❶       if self.y > 0 and co.y2 >= self.spiel.canvas_height:
❷           self.y = 0
❸           unten = False
❹       elif self.y < 0 and co.y1 <= 0:
❺           self.y = 0
❻           oben = False
```

Wenn die Figur auf dem Monitor herunterfällt, ist `y` größer als 0, und von daher müssen wir prüfen, ob sie nicht bereits den Boden der Leinwand erreicht hat (ansonsten würde sie ja durch den Boden hindurchfallen). Deshalb schauen wir in ❶, ob ihre `y2`-Position (die Unterseite des Strichmännchens) größer oder gleich der Variable `canvas_height` im `Spiel`-Objekt ist. Falls sie es ist, setzen wir den Wert von `y` in ❷ auf 0, stoppen dadurch den Fall des Strichmännchens und setzen die Variable `unten` in ❸ auf `False`, wodurch wir dem restlichen Code sagen, dass er nicht länger schauen muss, ob das Strichmännchen den Boden berührt hat.

Der Prozess, mit dem wir feststellen, ob das Strichmännchen die oberste Kante des Monitors berührt hat, ist dem Prozess sehr ähnlich, mit dem wir prüfen, ob es den Boden erreicht hat. Dazu prüfen wir in ❹, ob das Strichmännchen springt (`y` ist kleiner als 0), und prüfen dann, ob seine `y1`-Position kleiner oder gleich 0 ist – ob es also oben an der Leinwand angestoßen ist. Falls beide Bedingungen wahr sind, setzen wir `y` in ❺ gleich 0, um die Bewegung zu stoppen. Wir setzen in ❻ gleichzeitig die Variable oben auf `True`, um dem restlichen Code mitzuteilen, dass er nicht mehr prüfen muss, ob das Strichmännchen oben angestoßen ist.

Hat das Strichmännchen die Seite der Leinwand berührt?

Wir durchlaufen fast exakt den gleichen Prozess wie beim vorherigen Code, um zu ermitteln, ob das Strichmännchen die linke oder rechte Seite der Leinwand berührt hat:

```
            elif self.y < 0 and co.y1 <= 0:
                self.y = 0
                oben = False
❶           if self.x > 0 and co.x2 >= self.spiel.canvas_width:
❷               self.x = 0
❸               rechts = False
❹           elif self.x < 0 and co.x1 <= 0:
❺               self.x = 0
❻               links = False
```

Der Code in ❶ basiert auf der Erkenntnis, dass das Strichmännchen nach rechts rennt, falls `x` größer als 0 ist. Wir wissen auch, ob er schon vor die rechte Wand gerannt ist, indem wir nachschauen, ob die `x2`-Position (`co.x2`) größer oder gleich der Breite der Leinwand ist, die in `spiel.canvas_width` gespeichert ist. Falls beide Anweisungen wahr sind, setzen wir `x` gleich 0 (und beenden so das Rennen des Strichmännchens) und setzen die Variable `rechts` in ❸ auf falsch.

Mit anderen Sprites zusammenstoßen

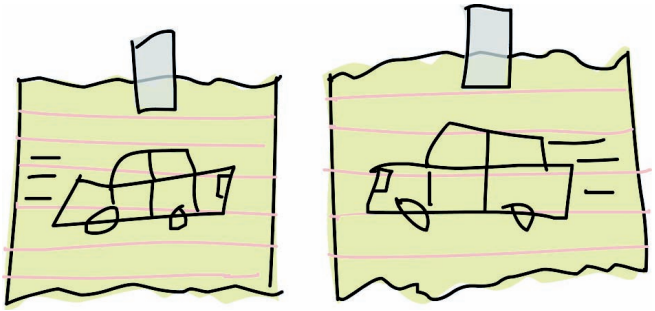
Sobald wir geprüft haben, ob die Figur die Seiten des Spielfensters berührt hat, müssen wir noch schauen, ob sie noch gegen etwas anderes gestoßen ist. Mit dem folgenden Code schleifen wir durch die Liste von Sprite-Objekten, die im Objekt `spiel` gespeichert sind, um zu prüfen, ob das Strichmännchen irgendeines von ihnen berührt hat.

```
            elif self.x < 0 and co.x1 <= 0:
                self.x = 0
                links = False
❶           for sprite in self.spiel.sprites:
❷               if sprite == self:
❸                   continue
❹               sprite_co = sprite.coords()
❺               if oben and self.y < 0 and angestoßen_oben(co, sprite_co):
❻                   self.y = -self.y
❼                   oben = False
```

In ❶ laufen wir in einer Schleife durch die Liste der Sprites und weisen jedes Sprite dabei der Variablen `sprite` zu. Der Code in ❷ besagt: Wenn der Sprite gleich `self` ist (das ist, als ob man sagen würde »falls dieser Sprite das Gleiche ist wie ich«), müssen wir nicht prüfen, ob das Strichmännchen irgendwo angestoßen ist, da es sich nur selbst berührt hätte. Falls die Variable `sprite` gleich `self` ist, gehen wir mit `continue` zum nächsten Sprite in der Liste.

Als Nächstes bekommen wir durch den Aufruf der Funktion `coords` in ❹ die Koordinaten des neuen Sprites und speichern das Ergebnis in der Variablen `sprite_co`. Anschließend überprüft der Code in ❺ Folgendes:

- Das Strichmännchen hat die Decke der Leinwand noch nicht berührt (die Variable `oben` ist immer noch wahr).
- Das Strichmännchen springt (der Wert von `y` ist kleiner als 0).
- Die Oberseite des Strichmännchens ist mit einem Sprite in der Liste zusammengestoßen (durch Verwendung der Funktion `angestoßen_oben`, die wir in Kapitel 17 erzeugt haben).



Falls alle diese Bedingungen wahr sind, möchten wir, dass der Sprite beginnt herunterzufallen, und so kehren wir den Wert von `y` in ❻ mit einem Minus (-) um. Die Variable `oben` wird in ❼ auf `False` gesetzt, denn wenn die Strichfigur einmal oben angestoßen ist, müssen wir nicht weiter auf Berührungen hin prüfen.

Auftreffen mit der Unterseite

Der nächste Teil der Schleife prüft, ob die Unterseite unserer Figur etwas berührt hat:

```
                if oben and self.y < 0 and angestoßen_oben(co, sprite_co):
                    self.y = -self.y
                    oben = False
❶               if unten and self.y > 0 and angestoßen_unten(self.y, \
                        co, sprite_co):
❷                   self.y = sprite_co.y1 - co.y2
❸                   if self.y < 0:
❹                       self.y = 0
❺                   oben = False
❻                   unten = False
```

In ❶ gibt es drei ähnliche Prüfungen: ob die Variable `unten` noch gesetzt ist, ob die Figur fällt (`y` ist größer als 0) und ob die Unterseite unserer Figur den Sprite berührt hat. Falls alle Prüfungen wahr ergeben, ziehen wir den unteren y-Wert (`y2`) des Strichmännchens vom oberen y-Wert des Sprites (`y1`) in ❷ ab. Kommt Dir das etwas merkwürdig vor? Schauen wir uns an, warum wir das so machen.

Stell Dir vor, unsere Spielfigur ist von einer Ebene heruntergefallen. Jedes Mal, wenn die Funktion `Hauptschleife` läuft, bewegt sie sich 4 Pixel auf dem Monitor nach unten und der Fuß des Strichmännchens ist 3 Pixel über einer anderen Ebene. Nehmen wir an, die Unterseite des Strichmännchens (`y2`) befindet sich an Position 57 und die Oberseite der Ebene (`y1`) an Position 60. In diesem Fall würde die Funktion `angestoßen_unten` `True` zurückgeben, da ihr Code den Wert von `y` (der 4 beträgt) zu der `y2`-Variablen des Strichmännchens hinzuzählt, was 61 ergäbe.

Wir möchten jedoch nicht, dass Herr Strichmann mit dem Fallen aufhört, sobald es so aussieht, als ob er eine Ebene oder den Boden des Spielfensters berührt hat. Das würde nämlich so aussehen, als würde er einen großen Sprung von einer Stufe machen und mitten in der Luft knapp über dem Boden anhalten. Das wäre zwar ein hübsches Kunststück, sieht aber in unserem Spiel nicht so gut aus. Wenn wir stattdessen den `y2`-Wert (von 57) der Figur von dem `y1`-Wert (von 60) der Ebene abziehen, bekommen wir 3. Das ist der Betrag, um den das Strichmännchen fallen sollte, damit es ordentlich oben auf der Ebene landet.

In ❸ sorgen wir dafür, dass die Berechnung keine negative Zahl ergibt. Falls sie es tut, setzen wir in ❹ `y` gleich 0. (Wenn wir die Zahl negativ lassen würden, würde das Strichmännchen wieder nach oben fliegen, und das wollen wir in diesem Spiel nicht.)

Zum Schluss setzen wir `oben` in ❺ und `unten` in ❻ auf falsch und müssen so nicht länger prüfen, ob das Strichmännchen oben oder unten angestoßen oder gegen einen anderen Sprite geprallt ist.

Wir machen noch eine weitere Prüfung in Richtung unten, um zu sehen, ob das Strichmännchen über die Kante einer Ebene gelaufen ist. Hier siehst Du den Code für diese `if`-Anweisung:

```
            if self.y < 0:
                self.y = 0
            unten = False
            oben = False
        if unten and fallen and self.y == 0 \
                and co.y2 < self.spiel.canvas_height \
                and angestoßen_unten(1, co, sprite_co):
            fallen = False
```

Damit die Variable `fallen` auf `False` gesetzt wird, müssen fünf Prüfungen wahr (`True`) ergeben:

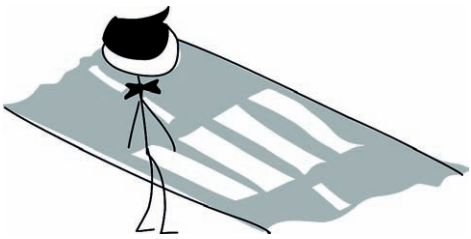
- Wir müssen immer noch prüfen, ob `unten` auf True steht.
- Wir müssen prüfen, ob das Strichmännchen fallen sollte (`fallen` steht noch auf `True`).
- Das Strichmännchen fällt noch nicht (`y` ist 0).
- Die Unterseite des Sprites hat noch nicht den Boden des Monitors berührt (ist geringer als die Höhe der Leinwand).
- Das Strichmännchen hat die Oberseite einer Ebene berührt (`angestoßen_unten` gibt `True` zurück).

Anschließend setzen wir die Variable `fallen` auf `False`.

Überprüfung links und rechts

Wir haben geprüft, ob das Strichmännchen einen Sprite, den Boden oder die Decke berührt hat. Nun müssen wir noch prüfen, ob es die linke oder rechte Seite berührt hat:

```
            if unten and fallen and self.y == 0 \
                    and co.y2 < self.spiel.canvas_height \
                    and angestoßen_unten(1, co, sprite_co):
                fallen = False
❶           if links and self.x < 0 and angestoßen_links(co, sprite_co):
❷               self.x = 0
❸               links = False
❹           if rechts and self.x > 0 and angestoßen_rechts(co, sprite_co):
❺               self.x = 0
❻               rechts = False
```



In ❶ prüfen wir, ob wir noch weiter nach Berührungen links Ausschau halten sollten (`links` steht noch auf `True`) und ob sich das Strichmännchen nach links bewegt (`x` ist weniger als 0). Wir prüfen mit der Funktion `angestoßen_links`, ob das Strichmännchen mit einem Sprite zusammengestoßen ist. Falls diese Bedingungen alle wahr sind, setzen wir in ❷ `x` auf 0 (damit das Strichmännchen aufhört zu rennen) und setzen `links` in ❸ auf `False`, damit wir nicht länger nach Berührungen auf der linken Seite suchen müssen.

Wie in ❹ zu sehen ist, ähnelt der Code dem für Berührungen auf der rechten Seite. Wir setzen in ❺ `x` wieder auf 0 und `rechts` in ❻ auf `False`, damit wir nicht mehr auf Zusammenstöße auf der rechten Seite hin prüfen müssen. Nachdem wir in alle vier Richtungen auf Berührungen prüfen können, sollte unsere `for`-Schleife so aussehen:

```
        elif self.x < 0 and co.x1 <= 0:
            self.x = 0
            links = False
        for sprite in self.spiel.sprites:
            if sprite == self:
                continue
            sprite_co = sprite.coords()
            if oben and self.y < 0 and angestoßen_oben(co, sprite_co):
                self.y = -self.y
                oben = False
            if unten and self.y > 0 and angestoßen_unten(self.y, \
                    co, sprite_co):
                self.y = sprite_co.y1 - co.y2
                if self.y < 0:
                    self.y = 0
                unten = False
                oben = False
            if unten and fallen and self.y == 0 \
                    and co.y2 < self.spiel.canvas_height \
                    and angestoßen_unten(1, co, sprite_co):
                fallen = False
            if links and self.x < 0 and angestoßen_links(co, sprite_co):
                self.x = 0
                links = False
            if rechts and self.x > 0 and angestoßen_rechts(co, sprite_co):
                self.x = 0
                rechts = False
```

Jetzt müssen wir der Funktion `move` nur noch ein paar weitere Zeilen hinzufügen:

```
            if rechts and self.x > 0 and angestoßen_rechts(co, sprite_co):
                self.x = 0
                rechts = False
❶       if fallen and unten and self.y == 0 \
                and co.y2 < self.spiel.canvas_height:
❷           self.y = 4
❸       self.spiel.canvas.move(self.bild, self.x, self.y)
```

In ❶ prüfen wir, ob die Variablen `fallen` und `unten` beide auf `True` stehen. Falls ja, sind wir mit unserer Schleifen durch jeden Ebenen-Sprite in der Liste gelaufen, ohne auf den Boden zu treffen.

Die letzte Prüfung in dieser Zeile bestimmt, ob der Wert für die Unterseite unserer Figur weniger als die Leinwandhöhe beträgt – ob sie also über dem Boden (der Leinwand) ist. Falls das Strichmännchen an nichts gestoßen ist und sich über dem Boden befindet, steht es in der Luft und sollte daher anfangen zu fallen (es ist also über das Ende einer Ebene hinausgelaufen). Damit es über das Ende einer jeden Ebene rennt, setzen wir in ❷ `y` gleich 4.

In ❸ bewegen wir das Bild anhand der Werte, die wir in den Variablen `x` und `y` gesetzt haben, über den Monitor. Die Tatsache, dass wir in einer Schleife durch die Sprites gelaufen sind und dabei auf Berührungen hin geprüft haben, kann bedeuten, dass wir beide Variablen auf 0 gesetzt haben, da das Strichmännchen sowohl die linke Wand als auch den Boden berührt hatte. In diesem Fall bewirkt der Aufruf der Funktion `move` durch die Leinwand nichts.

Es kann auch sein, dass Herr Strichmann über das Ende einer Ebene gelaufen ist. Falls das passiert, wird `y` auf 4 gesetzt und Herr Strichmann fällt nach unten.

Puh, das war eine lange Funktion!

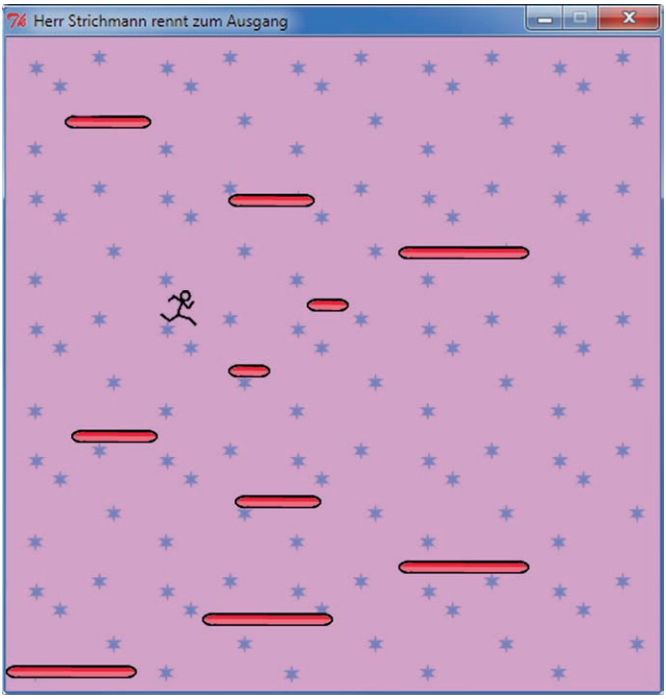
19.2 Testen unseres Strichmännchen-Sprites

Weiter oben haben wir die Klasse `StrichFigurSprite` erzeugt, und jetzt probieren wir sie aus, indem wir die folgenden zwei Zeilen vor dem Aufruf der Funktion `Hauptschleife` einfügen:

```
❶ sf = StrichFigurSprite(s)
❷ s.sprites.append(sf)
  s.Hauptschleife()
```

In ❶ erzeugen wir ein Strichfigur-Sprite-Objekt und setzen es gleich der Variablen `sf`. Wie wir es schon bei den Ebenen getan haben, fügen wir in ❷ diese neue Variable der Liste von Sprites im `Spiel`-Objekt hinzu.

Lass nun Dein Programm laufen. Du wirst sehen, dass Herr Strichmann rennen, von Ebene zu Ebene springen und fallen kann!



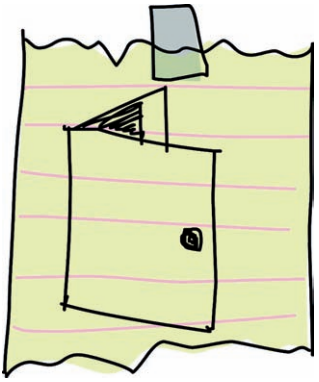
19.3 Die Tür!

Die einzige Sache, die bei unserem Spiel noch fehlt, ist die Ausgangstür. Wir runden das Spiel ab, indem wir einen Sprite für die Tür erstellen, Code zum Erkennen der Tür hinzufügen und unserem Programm ein `Tür`-Objekt geben.

Die Klasse TürSprite erzeugen

Du hast es Dir sicher schon gedacht – wir müssen noch eine weitere Klasse erzeugen: `TürSprite`. Hier ist der Anfang des Codes:

```
    class TürSprite(Sprite):
❶       def __init__(self, spiel, photo_image, x, y, width, height):
❷           Sprite.__init__(self, spiel)
❸           self.photo_image = photo_image
❹           self.bild = spiel.canvas.create_image(x, y, \
                    image=self.photo_image, anchor='nw')
❺           self.koordinaten = Coords(x, y, x + (width / 2), y + height)
❻           self.spielende = True
```



Wie man in ❶ sieht, hat die Klasse `TürSprite` Parameter für `self`, für ein `spiel`-Objekt, für ein `photo_image`-Objekt, für die `x`- und `y`-Koordinaten und für die Breite (`width`) und Höhe (`height`) des Bildes. In ❷ rufen wir wieder `__init__` auf, wie wir es schon bei unseren anderen Sprite-Klassen getan haben.

In ❸ speichern wir den Parameter `photo_image` mit einer Objekt-Variablen mit dem gleichen Namen ab, wie schon in `EbenenSprite`. Wir erzeugen das angezeigte Bild mit der Funktion der Leinwand `create_image` und speichern die zurückgegebene ID-Nummer dieser Funktion in der Objekt-Variablen `image` in ❹.

In ❺ setzen wir die Koordinaten des `TürSprite` auf die x- und y-Parameter (die zu den `x1`- und `y1`-Positionen der Tür werden) und errechnen dann die `x2`- und `y2`-Positionen. Die `x2`-Position erhalten wir durch die Addition der halben Breite (die Variable `width` geteilt durch zwei) zum Parameter `x`. Wenn `x` beispielsweise 10 beträgt (die `x1`-Koordinate also 10 ist) und die Breite 40 beträgt, wäre die `x2`-Koordinate 30 (10 plus die Hälfte von 40).

Warum machen wir nun diese kleine verwirrende Berechnung? Weil Herr Strichmann hier vor der Tür anhalten soll – anders als bei den Ebenen, bei denen wir möchten, dass Herr Strichmann aufhört zu rennen, sobald er gegen eine Ebene stößt. (Es würde nicht gut aussehen, wenn Herr Strichmann an der Tür aufhören würde zu rennen.) Du wirst es sehen, wenn Du das Spiel spielst und es bis zur Tür schaffst.

Im Gegensatz zur `x1`-Position ist die `y1`-Position leichter zu berechnen. Wir zählen nur den Wert der Variablen `height` zum `y`-Parameter hinzu, und das war's.

In ❻ setzen wir die Objekt-Variable `spielende` auf `True`. Dies bedeutet, dass das Spiel enden soll, sobald das Strichmännchen die Tür erreicht.

Die Tür erkennen

Jetzt müssen wir den Code der Funktion `move` in der `StrichfigurSprite`-Klasse ändern, der festlegt, was passiert, wenn das Strichmännchen links oder rechts an einen Sprite stößt. Hier ist die erste Änderung:

```
if links and self.x < 0 and angestoßen_links(co, sprite_co):
    self.x = 0
    links = False
    if sprite.spielende:
        self.spiel.rennen = False
```

Wir prüfen, ob der Sprite, gegen den das Strichmännchen gestoßen ist, eine `spielende`-Variable hat, die auf `True` steht. Falls ja, setzen wir die Variable `rennen` auf `False`, und alles kommt zum Stehen – wir haben das Spielende erreicht.

Die gleichen Zeilen fügen wir dem Code hinzu, der nach Zusammenstößen auf der rechten Seite schaut. Hier ist der Code:

```
        if rechts and self.x > 0 and angestoßen_rechts(co, sprite_co):
            self.x = 0
            rechts = False
            if sprite.spielende:
                self.spiel.rennen = False
```

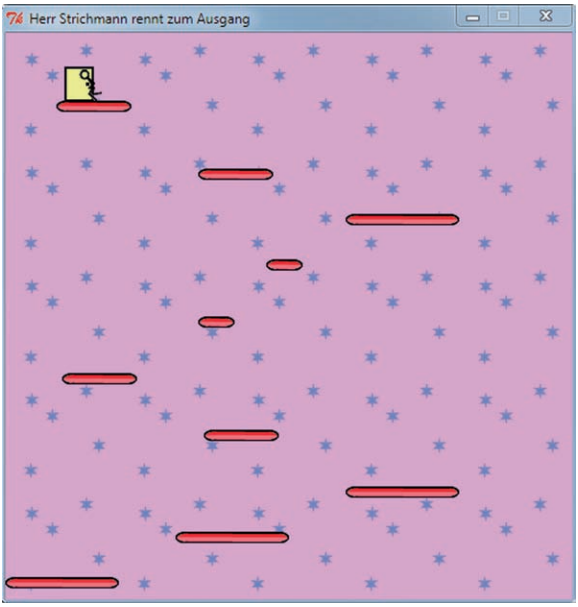
Das Tür-Objekt hinzufügen

Das letzte Element, das wir dem Spiel-Code hinzufügen, ist ein Objekt für die Tür. Wir fügen es vor der Hauptschleife ein. Genau vor dem Strichmännchen-Objekt erzeugen wir ein Tür-Objekt und fügen es der Liste von Sprites hinzu. Hier ist der Code dazu:

```
s.sprites.append(Ebene7)
s.sprites.append(Ebene8)
s.sprites.append(Ebene9)
s.sprites.append(Ebene10)
tür = TürSprite(s, PhotoImage(file="Tuer1.gif"), 45, 30, 40, 35)
s.sprites.append(tür)
sf = StrichFigurSprite(s)
s.sprites.append(sf)
s.Hauptschleife()
```

Mit der Variablen `s` für unser `spiel`-Objekt erzeugen wir ein Tür-Objekt, auf das ein PhotoImage folgt (das Türbild, das wir in Kapitel 16 erzeugt haben). Wir setzen die `x`- und `y`-Parameter auf 45 und 30, um die Tür auf eine Ebene ganz oben im Spielfeld zu setzen, und setzen Breite (`width`) und Höhe (`height`) auf 40 und 35. Dann fügen wir das Tür-Objekt der Liste von Sprites hinzu, wie wir es mit den anderen Sprites in unserem Spiel auch getan haben.

Das Ergebnis siehst Du, sobald Herr Strichmann die Tür erreicht hat. Statt neben der Tür hört er direkt vor der Tür auf zu rennen, wie es hier zu sehen ist:



19.4 Das fertige Spiel

Das vollständige Listing unseres Spiels enthält etwas mehr als 200 Zeilen Code. Im Folgenden siehst Du den gesamten Code des Spiels. Falls Du Schwierigkeiten hast, Dein Spiel ans Laufen zu bringen, vergleichst Du jede Funktion (und jede Klasse) mit diesem Listing und siehst nach, was Du verkehrt gemacht hast.

```
from tkinter import *
import random
import time

class Spiel:
    def __init__(self):
        self.tk = Tk()
        self.tk.title("Herr Strichmann rennt zum Ausgang")
        self.tk.resizable(0, 0)
        self.tk.wm_attributes("-topmost", 1)
        self.canvas = Canvas(self.tk, width=500, height=500, \
                highlightthickness=0)
        self.canvas.pack()
        self.tk.update()
        self.canvas_height = 500
        self.canvas_width = 500
        self.bg = PhotoImage(file="Hintergrund.gif")
        w = self.bg.width()
        h = self.bg.height()
```

```
        for x in range(0, 5):
            for y in range(0, 5):
                self.canvas.create_image(x * w, y * h, \
                        image=self.bg, anchor='nw')
        self.sprites = []
        self.rennen = True

    def Hauptschleife(self):
        while 1:
            if self.rennen == True:
                for sprite in self.sprites:
                    sprite.move()
            self.tk.update_idletasks()
            self.tk.update()
            time.sleep(0.01)

class Koordinaten:
    def __init__(self, x1=0, y1=0, x2=0, y2=0):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

def innerhalb_x(co1, co2):
    if (co1.x1 > co2.x1 and co1.x1 < co2.x2) \
            or (co1.x2 > co2.x1 and co1.x2 < co2.x2) \
            or (co2.x1 > co1.x1 and co2.x1 < co1.x2) \
            or (co2.x2 > co1.x1 and co2.x2 < co1.x1):
        return True
    else:
        return False

def innerhalb_y(co1, co2):
    if (co1.y1 > co2.y1 and co1.y1 < co2.y2) \
            or (co1.y2 > co2.y1 and co1.y2 < co2.y2) \
            or (co2.y1 > co1.y1 and co2.y1 < co1.y2) \
            or (co2.y2 > co1.y1 and co2.y2 < co1.y1):
        return True
    else:
        return False

def angestoßen_links(co1, co2):
    if innerhalb_y(co1, co2):
        if co1.x1 <= co2.x2 and co1.x1 >= co2.x1:
            return True
    return False
```

```
def angestoßen_rechts(co1, co2):
    if innerhalb_y(co1, co2):
        if co1.x2 >= co2.x1 and co1.x2 <= co2.x2:
            return True
    return False

def angestoßen_oben(co1, co2):
    if innerhalb_x(co1, co2):
        if co1.y1 <= co2.y2 and co1.y1 >= co2.y1:
            return True
    return False

def angestoßen_unten(y, co1, co2):
    if innerhalb_x(co1, co2):
        y_calc = co1.y2 + y
            if y_calc >= co2.y1 and y_calc <= co2.y2:
        return True
    return False

class Sprite:
    def __init__(self, spiel):
        self.spiel = spiel
        self.spielende = False
        self.koordinaten = None
    def move(self):
        pass
    def coords(self):
        return self.koordinaten

class EbenenSprite(Sprite):
    def __init__(self, spiel, photo_image, x, y, width, height):
        Sprite.__init__(self, spiel)
        self.photo_image = photo_image
        self.bild = spiel.canvas.create_image(x, y, \
                image=self.photo_image, anchor='nw')
        self.koordinaten = Koordinaten(x, y, x + width, y + height)

class StrichFigurSprite(Sprite):
    def __init__(self, spiel):
        Sprite.__init__(self, spiel)
        self.bilder_links = [
            PhotoImage(file="Figur-L1.gif"),
            PhotoImage(file="Figur-L2.gif"),
            PhotoImage(file="Figur-L3.gif")
        ]
        self.bilder_rechts = [
            PhotoImage(file="Figur-R1.gif"),
            PhotoImage(file="Figur-R2.gif"),
            PhotoImage(file="Figur-R3.gif")
        ]
```

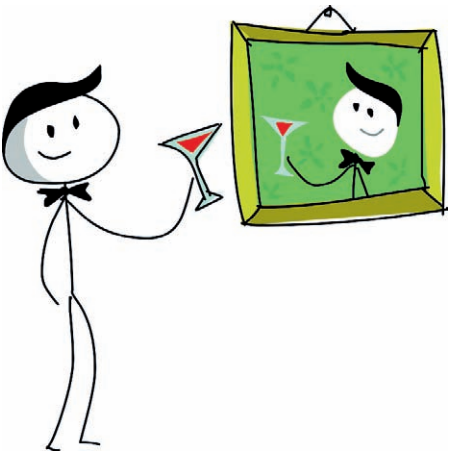
```
        self.bild = spiel.canvas.create_image(200, 470, \
                image=self.bilder_links[0], anchor='nw')
    self.x = -2
    self.y = 0
    self.aktuelles_bild = 0
    self.aktuelles_bild_plus = 1
    self.springen_zähler = 0
    self.letzte_zeit  = time.time()
    self.koordinaten = Koordinaten()
    spiel.canvas.bind_all('<KeyPress-Left>', self.nach_links)
    spiel.canvas.bind_all('<KeyPress-Right>', self.nach_rechts)
    spiel.canvas.bind_all('<space>', self.springen)

    def nach_links(self, evt):
        if self.y == 0:
            self.x = -2

    def nach_rechts(self, evt):
        if self.y == 0:
            self.x = 2

    def springen(self, evt):
        if self.y == 0:
            self.y = -4
            self.springen_zähler = 0

    def animieren(self):
        if self.x != 0 and self.y == 0:
            if time.time() - self.letzte_zeit  > 0.1:
                self.letzte_zeit  = time.time()
                self.aktuelles_bild += self.aktuelles_bild_plus
                if self.aktuelles_bild >= 2:
                    self.aktuelles_bild_plus = -1
                if self.aktuelles_bild <= 0:
                    self.aktuelles_bild_plus = 1
        if self.x < 0:
            if self.y != 0:
                self.spiel.canvas.itemconfig(self.bild, \
                        image=self.bilder_links[2])
            else:
                self.spiel.canvas.itemconfig(self.bild, \
                        image=self.bilder_links[self.aktuelles_bild])
            elif self.x > 0:
                if self.y != 0:
                    self.spiel.canvas.itemconfig(self.bild, \
                        image=self.bilder_rechts[2])
                else:
                    self.spiel.canvas.itemconfig(self.bild, \
                       image=self.bilder_rechts[self.aktuelles_bild])
```

```
    def coords(self):
        xy = list(self.spiel.canvas.coords(self.bild))
        self.koordinaten.x1 = xy[0]
        self.koordinaten.y1 = xy[1]
        self.koordinaten.x2 = xy[0] + 27
        self.koordinaten.y2 = xy[1] + 30
        return self.koordinaten

    def move(self):
        self.animieren()
        if self.y < 0:
            self.springen_zähler += 1
            if self.springen_zähler > 20:
                self.y = 4
        if self.y > 0:
            self.springen_zähler -= 1
        co = self.coords()
        links = True
        rechts = True
        oben = True
        unten = True
        fallen = True
        if self.y > 0 and co.y2 >= self.spiel.canvas_height:
            self.y = 0
            unten = False
        elif self.y < 0 and co.y1 <= 0:
            self.y = 0
            oben = False

        if self.x > 0 and co.x2 >= self.spiel.canvas_width:
            self.x = 0
            rechts = False
        elif self.x < 0 and co.x1 <= 0:
            self.x = 0
            links = False
        for sprite in self.spiel.sprites:
            if sprite == self:
                continue
            sprite_co = sprite.coords()
            if oben and self.y < 0 and angestoßen_oben(co, sprite_co):
                self.y = -self.y
                oben = False
            if unten and self.y > 0 and angestoßen_unten(self.y, co, \
                        sprite_co):
                self.y = sprite_co.y1 - co.y2
                if self.y < 0:
                    self.y = 0
                unten = False
                oben = False
```

```
            if unten and fallen and self.y == 0
                    and co.y2 < self.spiel.canvas_height \
                    and angestoßen_unten(1, co, sprite_co):
                fallen = False
            if links and self.x < 0 and angestoßen_links(co, \
                    sprite_co):
                self.x = 0
                links = False
                if sprite.spielende:
                    self.spiel.rennen = False
            if rechts and self.x > 0 and angestoßen_rechts(co, \
                    sprite_co):
                self.x = 0
                rechts = False
                if sprite.spielende:
                    self.spiel.rennen = False
        if fallen and unten and self.y == 0 \
                and co.y2 < self.spiel.canvas_height:
            self.y = 4
        self.spiel.canvas.move(self.bild, self.x, self.y)

class TürSprite(Sprite):
    def __init__(self, spiel, photo_image, x, y, width, height):
        Sprite.__init__(self, spiel)
        self.photo_image = photo_image
        self.image = spiel.canvas.create_image(x, y, \
                image=self.photo_image, anchor='nw')
        self.koordinaten = Koordinaten(x, y, x + \
                (width / 2), y + height)
        self.spielende = True

s = Spiel()
Ebene1 = EbenenSprite(s, PhotoImage(file="Ebene1.gif"), \
    0, 480, 100, 10)
Ebene2 = EbenenSprite(s, PhotoImage(file="Ebene1.gif"), \
    150, 440, 100, 10)
Ebene3 = EbenenSprite(s, PhotoImage(file="Ebene1.gif"), \
    300, 400, 100, 10)
Ebene4 = EbenenSprite(s, PhotoImage(file="Ebene1.gif"), \
    300, 160, 100, 10)
Ebene5 = EbenenSprite(s, PhotoImage(file="Ebene2.gif"), \
    175, 350, 66, 10)
Ebene6 = EbenenSprite(s, PhotoImage(file="Ebene2.gif"), \
    50, 300, 66, 10)
Ebene7 = EbenenSprite(s, PhotoImage(file="Ebene2.gif"), \
    170, 120, 66, 10)
Ebene8 = EbenenSprite(s, PhotoImage(file="Ebene2.gif"), \
    45, 60, 66, 10)
```

```
Ebene9 = EbenenSprite(s, PhotoImage(file="Ebene3.gif"), \
    170, 250, 32, 10)
Ebene10 = EbenenSprite(s, PhotoImage(file="Ebene3.gif"), \
    230, 200, 32, 10)
s.sprites.append(Ebene1)
s.sprites.append(Ebene2)
s.sprites.append(Ebene3)
s.sprites.append(Ebene4)
s.sprites.append(Ebene5)
s.sprites.append(Ebene6)
s.sprites.append(Ebene7)
s.sprites.append(Ebene8)
s.sprites.append(Ebene9)
s.sprites.append(Ebene10)
Tür = TürSprite(s, PhotoImage(file="Tuer1.gif"), 45, 30, 40, 35)
s.sprites.append(Tür)
sf = StrichFigurSprite(s)
s.sprites.append(sf)
s.Hauptschleife()
```

19.5 Was Du gelernt hast

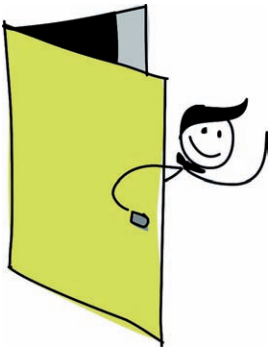


In diesem Kapitel haben wir unser Spiel *Herr Strichmann rennt zum Ausgang* abgeschlossen. Wir haben eine Klasse für unser animiertes Strichmännchen erzeugt und Funktionen geschrieben, mit denen wir es auf dem Bildschirm bewegen und währenddessen animieren. (Dazu lassen wie ein Bild in das nächste übergehen, um die Illusion von Bewegung zu erzeugen.) Wir haben eine einfache Kollisionserkennung verwendet, um zu prüfen, ob das Strichmännchen die linke oder rechte Seite der Leinwand oder einen anderen Sprite (wie etwa eine Ebene oder die Tür) berührt hat. Einen solchen Kollisions-Code haben wir auch hinzugefügt, um festzustellen, ob das Strichmännchen die Decke oder den Boden berührt hat, und um dafür zu sorgen, dass es ordentlich herunterfällt, sobald es über die Kante einer Ebene hinausgelaufen ist. Wir haben dann noch Code hinzugefügt, durch den das Spiel beendet wird, sobald Herr Strichmann die Tür erreicht hat.

19.6 Programmier-Puzzles

Wir können noch jede Menge Dinge tun, um das Spiel zu verbessern. Im Moment ist es noch sehr einfach gehalten, und wir können Code hinzufügen, damit es professioneller aussieht und interessanter zu spielen ist. Versuche, die folgenden Funktionalitäten hinzuzufügen, und überprüfe Deinen Code unter *www.dpunkt.de/python*.

#1: »Du hast gewonnen!«



Füge den Text »Du hast gewonnen!« für den Fall hinzu, dass das Strichmännchen die Tür erreicht hat, damit die Spieler sehen können, dass sie gewonnen haben. Du hast etwas Ähnliches schon mit dem Text »Game over« in dem Spiel Bounce! gemacht, das wir in Kapitel 15 abgeschlossen haben.

#2: Animation der Tür

In Kapitel 16 haben wir für die Tür zwei Bilder erstellt: ein Bild einer geschlossenen Tür und ein Bild einer offenen Tür. Sobald Herr Strichmann die Tür erreicht, soll das Bild der geschlossenen Tür gegen das der offenen Tür ausgetauscht werden, Herr Strichmann soll verschwinden, und anschließend soll wieder das Bild der geschlossenen Tür gezeigt werden. Dadurch entsteht der Eindruck, dass Herr Strichmann hinausgeht und die Tür hinter sich schließt. Dies erreichst Du, indem Du die `TürSprite`-Klasse und die `StrichfigurSprite`-Klasse änderst.

#3: Sich bewegende Ebenen

Versuche, eine neue Klasse namens `BeweglicheEbenenSprite` hinzuzufügen. Diese Ebene sollte sich seitlich hin und her bewegen, damit es schwieriger wird, Herrn Strichmann die Tür ganz oben erreichen zu lassen.

20 Wie geht es jetzt weiter?

Auf Deiner Tour durch Python hast Du einige grundlegende Konzepte des Programmierens kennengelernt, sodass es Dir jetzt viel leichter fallen wird, Dich in andere Programmiersprachen einzuarbeiten. Auch wenn Python äußerst nützlich ist, ist eine einzige Sprache nicht immer das beste Werkzeug für alle Aufgaben. Scheue Dich daher nicht davor, andere Arten des Programmierens auszuprobieren. Im Folgenden schauen wir uns einige Alternativen für die Spiele- und Grafikprogrammierung an und werfen dann einen kurzen Blick auf die gebräuchlichsten Programmiersprachen.

20.1 Spiele- und Grafikprogrammierung

Wenn Du mehr mit Spielen oder Grafikprogrammierung machen möchtest, hast Du viele Möglichkeiten. Hier sind nur einige davon:

- BlitzBasic (*http://www.blitzbasic.com/*), das eine besondere Version der Programmiersprache BASIC verwendet, die speziell zum Schreiben von Spielen entwickelt wurde
- Adobe Flash, eine Animations-Software, die für Browser entwickelt wurde und ihre eigene Programmiersprache *ActionScript* verwendet (*http://www.adobe.com/devnet/actionscript.html*)
- Alice (*http://www.alice.org/*), eine 3D-Entwicklungsumgebung
- Scratch (*http://scratch.mit.edu/*), ein Tool zur Entwicklung von Spielen
- Unity3D (*http://unity3d.com/*), noch ein Tool zur Spiele-Entwicklung

Wenn Du im Internet recherchierst, wirst Du jede Menge Quellen finden, die Dir beim Einstieg in jedes dieser Tools helfen. Wenn Du dagegen mit Python weiterspielen möchtest, kannst Du PyGame verwenden – das Python-Modul, das für die Spiele-Entwicklung entworfen wurde. Lass uns diese Möglichkeit näher beleuchten.

20.2 PyGame

PyGame Reloaded (oder `pygame2`) ist die Version von PyGame, die mit Python 3 funktioniert (frühere Versionen funktionieren nur mit Python 2).

Mit PyGame ein Spiel zu schreiben ist etwas komplizierter als mit `tkinter`. In Kapitel 13 haben wir in `tkinter` beispielsweise mit diesem Code ein Bild angezeigt:

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
myimage = PhotoImage(file='c:\\test.gif')
canvas.create_image(0, 0, anchor=NW, image=myimage)
```

Mit PyGame sieht der gleiche Code so aus (statt einer *.gif*-Datei lädt er eine *.bmp*-Datei):

```
import sys
import time
import pygame2
import pygame2.sdl.constants as constants
import pygame2.sdl.image as image
import pygame2.sdl.video as video
❶ video.init()
❷ img = image.load_bmp("c:\\test.bmp")
❸ screen = video.set_mode(img.width, img.height)
❹ screen.fill(pygame2.Color(255, 255, 255))
❺ screen.blit(img, (0, 0))
❻ screen.flip()
❼ time.sleep(10)
❽ video.quit()
```

Nach dem Import der `pygame2`-Module rufen wir die `__init__`-Funktion des PyGame-`video`-Moduls in ❶ auf, was ein wenig an das Erzeugen der Leinwand im `tkinter`-Beispiel erinnert. Mit der Funktion `load_mode` laden wir in ❷ ein BMP-Bild, erzeugen mit der Funktion `set_mode` ein `screen`-Objekt und übergeben die Breite und die Höhe des geladenen Bildes als Parameter in ❸. In der nächsten (optionalen) Zeile säubern wir den Bildschirm, indem wir ihn in ❹ mit Weiß auf-

füllen und mit der Funktion `blit` des `screen`-Objekts das Bild in ❺ darstellen. Die Parameter für diese Funktion sind das `img`-Objekt und ein Tupel, das die Position enthält, an der wir das Bild darstellen wollen (0 Pixel zur Seite und 0 Pixel nach unten).

PyGame verwendet einen *Off-Screen-Puffer*. Bei der Off-Screen-Technik werden die Grafiken in einem unsichtbaren Bereich des Computerspeichers verarbeitet und anschließend vollständig in den sichtbaren Bereich (auf Deinen Monitor) kopiert. Durch das Off-Screen-Verfahren wird das Flimmern reduziert, das entstehen kann, wenn viele Objekte auf einmal auf dem Monitor aufgebaut werden. Das Kopieren aus dem Off-Screen-Puffer in die sichtbare Anzeige wird durch die Funktion `flip` in ❻ durchgeführt.

Anschließend lassen wir unser Programm in ❼ für 10 Sekunden pausieren, da das Fenster im Gegensatz zur `tkinter`-Leinwand sofort geschlossen wird, falls wir es nicht daran hindern. Mit `video.init` räumen wir in ❽ auf, damit PyGame sich ordentlich schließt. Es gibt noch sehr viel mehr über PyGame zu sagen, aber dieses Beispiel gibt Dir einen ersten Eindruck.

20.3 Programmiersprachen

Falls Du an weiteren Programmiersprachen interessiert bist: Aktuell sind Java, C/C++, C#, PHP, Objective-C, Perl, Ruby und JavaScript beliebt. Wir unternehmen einen kleinen Rundgang durch diese Sprachen und schauen uns an, wie unser kleines »Hallo Welt«-Programm (wie die Python-Version, mit der wir in Kapitel 2 angefangen haben) in jeder von ihnen aussieht. Keine dieser Sprachen richtet sich jedoch an Programmieranfänger, und sie unterscheiden sich deutlich von Python. Zu den angegebenen englischsprachigen Websites findest Du unter *www.dpunkt.de/python* entsprechende Quellen in deutscher Sprache.

Java

Java (*http://www.oracle.com/technetwork/java/index.html*) ist eine mäßig komplizierte Programmiersprache mit einer großen mitgelieferten Bibliothek von Modulen (*packages* genannt). Im Internet findet man jede Menge kostenlose Dokumentationen. Java kannst Du auf den meisten Betriebssystemen einsetzen. Java ist auch die Sprache, die auf Android-Mobiltelefonen verwendet wird.

Hier ein Beispiel für *Hallo Welt* in Java:

```
public class Hallo Welt {
    public static final void main(String[] args) {
        System.out.println("Hallo Welt");
    }
}
```

C/C++

C (*http://www.cprogramming.com/*) und C++ (*http://www.stroustrup.com/C++.html*) sind komplizierte Programmiersprachen, die unter allen Betriebssystemen verwendet werden. Beide gibt es sowohl in freien als auch in kommerziellen Versionen. Beide Sprachen (und das gilt für C++ vielleicht noch etwas mehr als für C) sind schwer zu erlernen. Du wirst beispielsweise feststellen, dass Du einige Dinge dort manuell kodieren musst, die Python direkt anbietet (wie etwa dem Computer zu sagen, dass er ein Teil des Speicherinhalts in einem Objekt speichern soll). Viele der kommerziell erhältlichen Spiele und Konsolen sind in der einen oder anderen Form von C oder C++ programmiert. Hier ist ein Beispiel für *Hallo Welt* in C:

```
#include <stdio.h>
int main ()
{
    printf ("Hallo Welt\n");
}
```

Ein Beispiel in C++ könnte folgendermaßen aussehen:

```
#include <iostream>
int main()
{
    std::cout << "Hallo Welt\n";
    return 0;
}
```

C#

C# (*http://msdn.microsoft.com/de-de/vcsharp/aa336706.aspx*), was »C sharp« ausgesprochen wird, ist eine mäßig komplizierte Sprache für Windows, die Java sehr ähnelt. Sie ist ein bisschen leichter zu lernen als C und C++.

Hier siehst Du ein Beispiel für *Hallo Welt* in C#:

```
public class Hallo
{
    public static void Main()
    {
        System.Console.WriteLine("Hallo Welt");
    }
}
```

PHP

PHP (*http://www.php.net*) ist eine Programmiersprache für Webseiten. Dafür benötigst Du einen Webserver (Software, die Webseiten an den Browser liefert), auf dem PHP installiert ist. Sämtliche Software, die man dafür benötigt, ist für alle wesentlichen Betriebssysteme kostenlos erhältlich. Um mit PHP arbeiten zu können, musst Du HTML (eine einfache Sprache zum Aufbau von Webseiten) erlernen. Eine kostenlose Anleitung findest Du unter *http://php.net/manual/de/tutorial.php* und eine für HTML unter *http://www.w3schools.com/html/*.

Eine HTML-Seite, die »Hallo Welt« anzeigt, könnte so aussehen:

```
<html>
    <body>
        <p>Hallo Welt</p>
    </body>
</html>
```

Eine Seite in PHP, die das Gleiche macht, könnte so ausschauen:

```
<?php
echo "Hallo Welt\n";
?>
```

Objective-C

Objective-C (*http://classroomm.com/objective-c/)* ist der Programmiersprache C sehr ähnlich (es handelt sich in der Tat um eine Erweiterung von C) und wird hauptsächlich auf Apple-Computern verwendet. Es ist auch die Programmiersprache für das iPhone und das iPad.

Hier siehst Du ein Beispiel von *Hallo Welt* in Objective-C:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSLog (@"Hallo Welt");

    [pool drain];
return 0;
}
```

PERL

Die Programmiersprache Perl (*http://www.perl.org/)* ist kostenlos für alle wesentlichen Betriebssysteme verfügbar. Sie wird meist bei der Entwicklung von Webseiten (ähnlich wie PHP) verwendet.

Hier ist ein Beispiel für *Hallo Welt* in Perl:

```
print("Hello World\n");
```

Ruby

Ruby (*http://www.ruby-lang.org/de/*) ist eine frei erhältliche Programmiersprache für alle wesentlichen Betriebssysteme. Sie wird meist bei der Erstellung von Webseiten, vor allem im Framework *Ruby on Rails* benutzt. (Ein Framework ist eine Gruppe von Bibliotheken, die die Entwicklung bestimmter Arten von Anwendungen unterstützt.)

Hier siehst Du ein Beispiel für *Hallo Welt* in Ruby:

```
puts "Hallo Welt"
```

JavaScript

JavaScript (*https://developer.mozilla.org/de/docs/JavaScript*) ist eine Programmiersprache, die normalerweise innerhalb von Webseiten verwendet wird, aber auch immer mehr bei der Spieleprogrammierung eingesetzt wird. Die Syntax ist im Prinzip die gleiche wie bei Java, man findet jedoch mit JavaScript leichter den Einstieg. Eine einfache HTML-Seite, die ein JavaScript-Programm enthält, kannst Du mit einem Browser laufen lassen, ohne dass Du dazu eine Shell, Befehlszeile oder sonst etwas benötigst. Einen guten Einstieg in JavaScript könntest Du in der Codeacademy unter *http://www.codecademy.com* finden.

Wie ein *Hallo Welt*-Beispiel in JavaScript aussieht, hängt davon ab, ob Du es in einem Browser oder in einer Shell laufen lässt. In einer Shell würde unser Beispiel so aussehen:

```
print('Hello World');
```

In einem Browser könnte so aussehen:

```
<html>
    <body>
        <script type="text/javascript">
            alert("Hallo Welt");
        </script>
    </body>
</html>
```

20.4 Abschließende Worte

Egal ob Du nun bei Python bleibst oder eine andere Programmiersprache ausprobierst (und es gibt noch viel mehr als die Sprachen, die hier aufgelistet sind), die Konzepte, die Du in diesem Buch entdeckt hast, sind überall nützlich. Selbst falls Du nicht mit dem Programmieren weitermachst: Wenn Du die grundsätzlichen Ideen verstanden hast, hilft Dir das bei vielen anderen Tätigkeiten, egal ob in der Schule oder später im Berufsleben.

Viel Glück und viel Spaß beim Programmieren!

Anhang

Python-Schlüsselwörter

In Python (und in den meisten anderen Programmiersprachen) sind *Schlüsselwörter* Begriffe mit besonderer Bedeutung. Sie werden als Teil der Programmiersprache selbst gebraucht und dürfen daher für nichts anderes verwendet werden. Wenn Du zum Beispiel versuchst, Schlüsselwörter als Variablen zu benutzen oder sie sonst falsch verwendest, bekommst Du von der Python-Konsole merkwürdige (manchmal lustige, manchmal verwirrende) Fehlermeldungen.

Dieser Anhang beschreibt jedes der Python-Schlüsselwörter. Er soll Dir als praktisches Nachschlagewerk beim weiteren Programmieren dienen.

and

Mit dem Schlüsselwort `and` (und) werden zwei Ausdrücke innerhalb einer Anweisung (wie z.B. in einer `if`-Anweisung) verbunden, um zu sagen, dass beide Ausdrücke wahr sein müssen. Hier ein Beispiel:

```
if Alter > 10 and Alter < 20:
  print('Vorsicht Teenager!!!')
```

Dieser Code bedeutet, dass der Wert der Variable `Alter` mehr als 10 und (`and`) weniger als 20 betragen muss, damit die Mitteilung angezeigt wird.

as

Das Schlüsselwort `as` kann man verwenden, um einem importierten Modul einen anderen Namen zu geben. Stell Dir zum Beispiel vor, Du hättest ein Modul mit einem sehr langen Namen:

```
Ich_bin_ein_Python-Modul_das_wenig_bringt
```

Es wäre sehr nervig, diesen Modulnamen jedes Mal zu tippen, wenn man ihn benutzen wollte:

```
import Ich_bin_ein_Python-Modul_das_wenig_bringt
Ich_bin_ein_Python-Modul_das_wenig_bringt.mach_etwas()
Ich habe etwas gemacht das wenig bringt.
Ich_bin_ein_Python-Modul_das_wenig_bringt.mach_etwas_anderes()
Ich habe etwas anderes gemacht das wenig bringt!!
```

Stattdessen kannst Du dem Modul einen neuen und kürzeren Namen beim Importieren geben und dann einfach diesen neuen Namen verwenden (das ist ein bisschen wie ein Spitzname):

```
import Ich_bin_ein_Python-Modul_das_wenig_bringt as bringt_nichts
bringt_nichts.mach_etwas()
Ich habe etwas gemacht das wenig bringt.
bringt_nichts.mach_etwas_anderes()
Ich habe etwas anderes gemacht das wenig bringt!!
```

assert

Mit dem Schlüsselwort `assert` (engl. für »behaupten«) drückt man aus, dass ein Stück Code wahr sein muss. Man kann damit auch Fehler und Probleme im Code in meistens anspruchsvolleren Programmen aufdecken (weshalb wir `assert` in diesem Buch nicht verwendet haben). Hier ist eine einfache `assert`-Anweisung:

```
>>> meineZahl = 10
>>> assert meineZahl < 5
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    assert meineZahl < 5
AssertionError
```

In diesem Beispiel behaupten wir, dass der Wert der Variablen `meineZahl` weniger als 5 beträgt. Da er es nicht ist, gibt Python eine Fehlermeldung aus (einen AssertionError).

break

Das Schlüsselwort `break` wird verwendet, um einen Code-Ablauf zu unterbrechen. Du kannst `break` innerhalb einer `for`-Schleife verwenden:

```
Alter = 10
for x in range(1, 100):
    print('Zähle %s' % x)
    if x == Alter:
        print('Zählen beendet')
        break
```

Da die Variable `Alter` hier auf 10 gesetzt ist, gibt der Code Folgendes aus:

```
Zähle 1
Zähle 2
Zähle 3
Zähle 4
Zähle 5
Zähle 6
Zähle 7
Zähle 8
Zähle 9
Zähle 10
Zählen beendet
```

Sobald der Wert der Variablen x 10 erreicht, zeigt der Code »Zählen beendet« an und steigt aus der Schleife aus.

class

Mit dem Schlüsselwort `class` definiert man eine Art Objekt, wie etwa ein Fahrzeug, ein Tier oder eine Person. Klassen können eine Funktion namens `__init__` haben, die alle Aufgaben erledigt, die das Objekt einer Klasse benötigt. Zum Beispiel benötigt ein Objekt der Klasse `Auto` bei dessen Erzeugung die Variable `Farbe`:

```
class Auto:
    def __init__(self, Farbe):
        self.Farbe = Farbe

auto1 = Auto('rot')
auto2 = Auto('blau')
print(auto1.Farbe)
rot
print(auto2.Farbe)
blau
```

continue

Mit dem Schlüsselwort `continue` kann man zum nächsten Element einer Schleife »springen«, sodass der restliche Code eines Schleifen-Blocks nicht ausgeführt wird. Im Gegensatz zu `break` springen wir allerdings nicht aus der Schleife heraus, sondern machen mit dem nächsten Element weiter. Wenn wir beispielsweise eine Liste von Elementen hätten und Elemente, die mit *B* anfangen, auslassen wollten, könnten wir den folgenden Code benutzen:

```
❶ >>> meine_Elemente = ['Apfel', 'Avocado', 'Banane', 'Birne',
                        'Clementine', 'Cashew']
❷ >>> for Element in meine_Elemente:
❸             if Element.startswith('b'):
❹                 continue
❺             print(Element)
```

```
Apfel
Avocado
Clementine
Cashew
```

In ❶ erstellen wir unsere Liste von Elementen und benutzen dann eine `for`-Schleife, um durch unsere Elemente zu laufen und für jedes von ihnen in ❷ einen Code-Block auszuführen. Falls das Element in ❸ mit dem Buchstaben *B* beginnt, machen wir in ❹ mit dem nächsten Element weiter (`continue`). Anderenfalls geben wir in ❺ das Element aus.

def

Mit dem Schlüsselwort `def` wird eine Funktion definiert – zum Beispiel, um eine Funktion zu erzeugen, die eine Anzahl von Jahren in die entsprechende Anzahl von Minuten umwandelt:

```
>>> def Minuten(Jahre):
    return Jahre * 365 * 24 * 60
>>> Minuten(10)
5256000
```

del

Durch das Schlüsselwort `del` wird etwas entfernt. Wenn Du beispielsweise eine Wunschliste für Deinen Geburtstag hast, es Dir mit einem dieser Dinge jedoch anders überlegt hast, würdest Du wahrscheinlich eines aus der Liste streichen und etwas anderes hineinschreiben:

ferngesteuertes Auto
neues Fahrrad
~~Computerspiel~~
Roboreptile

In Python würde die Original-Liste folgendermaßen aussehen:

```
Was_ich_möchte = ['ferngesteuertes Auto', 'neues Fahrrad',
                  'Computerspiel']
```

Du könntest mit `del` und dem Index des entsprechenden Elements das Computerspiel entfernen. Anschließend könntest Du mit der Funktion `append` das neue Element hinzufügen:

```
del Was_ich_möchte[2]
Was_ich_möchte.append('Roboreptile')
```

Anschließend gibst Du die neue Liste aus:

```
print(Was_ich_möchte)
['ferngesteuertes Auto', 'neues Fahrrad', 'Roboreptile']
```

elif

Das Schlüsselwort `elif` verwendet man als Bestandteil einer `if`-Anweisung. In der Beschreibung des Schlüsselworts `if` findest Du ein Beispiel.

else

Das Schlüsselwort `else` verwendet man als Bestandteil einer `if`-Anweisung. In der Beschreibung des Schlüsselworts `if` findest Du ein Beispiel.

exept

Mit dem Schlüsselwort `exept` deckt man Probleme im Code auf. Es wird meistens in ziemlich komplizierten Programmen verwendet, und deshalb kommt es in diesem Buch nicht vor.

finally

Das Schlüsselwort `finally` verwendet man, um sicherzustellen, dass ganz bestimmter Code ausgeführt wird, sobald ein Fehler auftritt (meistens um Unordnung, die ein Stück Code hinterlassen hat, aufzuräumen). Dieses Schlüsselwort kommt in diesem Buch nicht vor, da es für fortgeschrittenes Programmieren vorgesehen ist.

for

Mit dem Schlüsselwort `for` erzeugt man eine Code-Schleife, die eine bestimmte Anzahl an Malen durchlaufen wird. Hier ein Beispiel:

```
for x in range(0, 5):
  print('x ist %s' %x)
```

Diese `for`-Schleife führt den Code-Block (die `print`-Anweisung) fünfmal aus, was folgende Ausgabe ergibt:

```
x ist 0
x ist 1
x ist 2
x ist 3
x ist 4
```

from

Wenn Du ein Modul importierst, kannst Du mit dem Schlüsselwort `from` nur denjenigen Teil importieren, den Du benötigst. Das Modul `turtle`, das wir in Kapitel 5 vorgestellt haben, enthält eine Klasse namens `Pen`, mit der wir ein `Pen`-Objekt erzeugt haben (die Leinwand, auf der sich die Schildkröte bewegt). Hier steht, wie wir das gesamte Modul `turtle` importieren und anschließend die `Pen`-Klasse benutzen:

```
import turtle
t = turtle.Pen()
```

Du könntest auch ausschließlich die Pen-Klasse importieren und sie direkt verwenden (ohne Dich überhaupt auf das Modul turtle zu beziehen):

```
from turtle import Pen
t = Pen()
```

Dies könntest Du tun, damit Du beim nächsten Mal, wenn Du ganz oben in das Programm schaust, alle von Dir verwendeten Funktionen und Klassen sehen kannst (das ist vor allem in größeren Programmen nützlich, in denen jede Menge Module importiert werden). Wenn Du Dich allerdings dafür entscheidest, wirst Du diejenigen Teile des Moduls, die Du nicht importiert hast, natürlich auch nicht nutzen können. Das Modul time beispielsweise, hat die Funktionen localtime und gmtime. Wenn Du also nur localtime importierst und dann versuchst, gmtime zu verwenden, bekommst Du eine Fehlermeldung:

```
>>> from time import localtime
>>> print(localtime())
time.struct_time(tm_year=2013, tm_mon=2, tm_mday=27, tm_hour=10,
tm_min=59, tm_sec=58, tm_wday=2, tm_yday=58, tm_isdst=0)
>>> print(gmtime())
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    print(gmtime())
NameError: name 'gmtime' is not defined
```

Die Fehlermeldung name 'gmtime' is not defined bedeutet, dass Python nichts über die Funktion gmtime weiß, da Du sie nicht importiert hast.

Wenn es ein paar Funktionen eines bestimmten Moduls gibt, die Du verwenden möchtest, Dich aber nicht auf sie mit den Modulnamen beziehen möchtest (zum Beispiel time.localtime oder time.gmtime) kannst Du mit einem Asterisk (*) alles aus dem Modul importieren:

```
>>> from time import *
>>> print(localtime())
time.struct_time(tm_year=2013, tm_mon=2, tm_mday=27, tm_hour=10,
tm_min=59, tm_sec=58, tm_wday=2, tm_yday=58, tm_isdst=0)
>>> print(gmtime())
time.struct_time(tm_year=2013, tm_mon=2, tm_mday=27, tm_hour=10,
tm_min=6, tm_sec=58, tm_wday=2, tm_yday=58, tm_isdst=0)
```

Auf diese Weise wird alles aus dem time-Modul importiert, und Du kannst Dich auf die einzelnen Funktionen durch deren Namen beziehen.

global

Was Gültigkeitsbereiche in Programmen sind, wird in Kapitel 8 erklärt. Der Gültigkeitsbereich bezieht sich auf die *Sichtbarkeit* einer Variablen. Falls eine Variable außerhalb einer Funktion definiert wird, ist sie normalerweise innerhalb der Funktion sichtbar. Falls die Variable dagegen innerhalb einer Funktion definiert

wird, kann sie außerhalb dieser Funktion nicht gesehen werden. Das Schlüsselwort `global` macht eine Ausnahme von dieser Regel. Eine Variable, die als `global` definiert wird, kann überall gesehen werden. Hier ist ein Beispiel:

```
>>> def test():
        global a
        a = 1
        b = 2
```

Was glaubst Du, wird passieren, wenn Du `print(a)` aufrufst und anschließend `print(b)`, nachdem die Funktion `test` gelaufen ist? Beim ersten Mal wird es funktionieren, am zweiten Mal gibt es eine Fehlermeldung:

```
>>> test()
>>> print(a)
1
>>> print(b)
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in <module>
    print(b)
NameError: name 'b' is not defined
```

Die Variable `a` wurde innerhalb der Funktion auf `global` gesetzt und wurde dadurch sichtbar, obwohl die Funktion schon abgeschlossen war. Dagegen ist `b` immer noch nur innerhalb der Funktion sichtbar. (Du musst das `global`-Schlüsselwort vor dem Setzen des Wertes Deiner Variable verwenden.)

if

Das Schlüsselwort `if` wird verwendet, um eine Entscheidung über etwas zu treffen. Es kann auch mit den Schlüsselwörtern `else` und `elif` (else if) benutzt werden. Eine `if`-Anweisung funktioniert so, als ob man sagt: »Wenn etwas wahr ist, führe folgenden Vorgang durch.« Hier siehst Du ein Beispiel:

```
❶ if Spielzeugpreis > 1000:
❷     print('Dieses Spielzeug ist zu teuer')
❸ elif Spielzeugpreis > 100:
❹     print('Dieses Spielzeug ist teuer')
❺ else:
❻     print('Ich kann mir dieses Spielzeug leisten')
```

Diese if-Anweisung sagt in ❶ Folgendes: Wenn der Preis für ein Spielzeug über 1000 € liegt, soll die Nachricht in ❷ angezeigt werden, dass es zu teuer ist. Falls das Spielzeug in ❸ über 100 € kosten soll, wird die Nachricht in ❹ angezeigt, das es teuer ist. Falls in ❺ keine dieser Bedingungen wahr ist, soll die Nachricht in ❻ (»Ich kann mir dieses Spielzeug leisten«) angezeigt werden.

import

Das Schlüsselwort `import` wird verwendet, um Python zu sagen, dass es ein Modul laden soll, damit man es verwenden kann. Im folgenden Beispiel sagt der Code, dass Python das Modul `sys` benutzen soll:

```
import sys
```

in

Das Schlüsselwort `in` wird in Ausdrücken verwendet, um nachzusehen, ob sich ein Element innerhalb einer Sammlung von Elementen befindet. Befindet sich zum Beispiel die Zahl 1 in einer Liste (einer Sammlung) von Zahlen?

```
>>> if 1 in [1,2,3,4]:
    print('Zahl befindet sich in der Liste')
Zahl befindet sich in der Liste
```

Hier steht, wie man herausfindet, ob sich der String `'Hose'` in der Liste von Bekleidungsstücken befindet:

```
>>> Bekleidungsliste = ['kurze Hose', 'Unterwäsche', 'Boxershorts',
                        'lange Unterhose', 'Schlüpfer']
>>> if 'Hose' in Bekleidungsliste:
  print('Hose ist auf der Liste')
else:
  print('Hose ist nicht auf der Liste')
Hose ist nicht auf der Liste
```

is

Das Schlüsselwort `is` ist ein bisschen wie der Operator (`==`), den man gebraucht, um zu prüfen, ob zwei Dinge gleich sind (zum Beispiel `10 == 10` ist wahr, und `10 == 11` ist falsch). Es gibt allerdings einen grundsätzlichen Unterschied zwischen `is` und `==`. Wenn Du zwei Dinge vergleichst, kann `==` wahr zurückgeben, wogegen `is das` eventuell nicht tut (selbst dann, wenn Du glaubst, dass die Dinge die gleichen sind). Dabei handelt es sich um ein fortgeschrittenes Programmierkonzept. Wir bleiben in diesem Buch daher beim `==`.

lambda

Mit dem Schlüsselwort `lambda` erzeugt man anonyme oder auch Inline-Funktionen. Dieses Schlüsselwort wird in fortgeschritteneren Programmen benutzt, und wir haben es in diesem Buch nicht behandelt.

not

Wenn etwas wahr ist, macht das Schlüsselwort `not` es falsch. Wenn wir beispielsweise eine Variable `x` erzeugen und sie auf den Wert `True` setzen und anschließend den Wert dieser Variablen mit `not` ausgeben, bekommen wir folgendes Ergebnis:

```
>>> x = True
>>> print(not x)
False
```

Dies ergibt so lange keinen Sinn, bis Du das Schlüsselwort in `if`-Anweisungen benutzt. Um zum Beispiel herauszufinden, ob ein Element nicht in einer Liste enthalten ist, können wir so etwas schreiben:

```
>>> Bekleidungsliste = ['kurze Hose', 'Unterwäsche', 'Boxershorts',
                        'lange Unterhose', 'Schlüpfer']
>>> if 'Hose' not in Bekleidungsliste:
    print('Du musst unbedingt eine Hose kaufen')
Du musst unbedingt eine Hose kaufen
```

or

Das `or`-Schlüsselwort verwendet man beim Verbinden zweier Bedingungen in einer Anweisung (wie z.B. in einer `if`-Anweisung), um auszudrücken, dass mindestens eine der Bedingungen wahr sein sollte. Hier ist ein Beispiel:

```
if Dino == 'Tyrannosaurus' or Dino == 'Allosaurus':
    print('Fleischfresser')
elif Dino == 'Ankylosaurus' or Dino == 'Apatosaurus':
    print('Pflanzenfresser')
```

Wenn in diesem Fall die Variable `Dino` den `Tyrannosaurus` oder `Allosaurus` enthält, zeigt das Programm »Fleischfresser« an. Falls sie `Ankylosaurus` oder `Apatosaurus` enthält, gibt das Programm »Pflanzenfresser« aus.

pass

Manchmal möchtest Du bei der Entwicklung eines Programms nur kleine Teile schreiben, um Dinge auszuprobieren. Das Problem dabei ist, dass Du keine `if`-Anweisung haben kannst, ohne dazu den Code-Block zu haben, der ausgeführt werden soll, falls der Ausdruck in der `if`-Anweisung wahr ist. Ebenso kannst Du keine `for`-Schleife ohne den Code-Block haben, der in der Schleife ausgeführt werden soll. Der folgende Code beispielweise funktioniert wunderbar:

```
>>> Alter = 15
>>> if Alter > 10:
    print('älter als 10')
älter als 10
```

Falls Du aber den Code-Block der `if`-Anweisung nicht ausfüllst, wartet IDLE auf den Code-Block und zeigt nach dem Drücken der Enter-Taste keinen Prompt:

```
>>> Alter = 15
>>> if Alter > 10:
```

In solchen Fällen kannst Du das `pass`-Schlüsselwort anwenden, um eine Anweisung zu schreiben, ohne dabei den Code-Block einzugeben, der dazugehört.

Nehmen wir beispielsweise an, dass Du eine `for`-Schleife mit einer `if`-Anweisung darin erzeugen möchtest. Vielleicht hast Du Dich noch nicht entschieden, was Du in die `if`-Anweisung schreiben möchtest – eventuell verwendest Du die `print`-Funktion, fügst ein `break` oder sonst etwas ein. Wenn Du an dieser Stelle `pass` verwendest, wird der Code trotzdem funktionieren (selbst wenn er noch nicht genau das tut, was Du möchtest). Hier siehst Du wieder unsere `if`-Anweisung, die diesmal das Schlüsselwort `pass` enthält:

```
>>> Alter = 15
>>> if Alter > 10:
        pass
```

Der folgende Code zeigt eine weitere Verwendung des Schlüsselwortes `pass`:

```
>>> for x in range(0, 7):
        print('x ist %s' % x)
        if x == 4:
            pass

x ist 0
x ist 1
x ist 2
x ist 3
x ist 4
x ist 5
x ist 6
```

Python prüft bei der Ausführung des Code-Blocks in der Schleife immer noch jedes Mal, ob die Variable `x` den Wert 4 enthält, unternimmt jedoch nichts, sodass es alle Zahlen im Bereich zwischen 0 und 7 ausgibt.

Später kannst Du den Code im Block für die `if`-Anweisung hinzufügen, indem Du das Schlüsselwort `pass` durch etwas anderes, wie zum Beispiel `break`, ersetzt:

```
>>> for x in range(1, 7):
        print('x ist %s' % x)
        if x == 5:
            break

x ist 0
x ist 1
x ist 2
x ist 3
x ist 4
x ist 5
```

Das Schlüsselwort `pass` wird meistens verwendet, wenn man eine Funktion erzeugt, den Code für die Funktion aber noch nicht schreiben möchte.

raise

Das Schlüsselwort `raise` kann man benutzen, damit es einen Fehler gibt. Das mag zwar etwas merkwürdig klingen, bei der fortgeschrittenen Programmierung kann das allerdings ziemlich nützlich sein. (Wir verwenden dieses Schlüsselwort in diesem Buch nicht.)

return

Mit dem Schlüsselwort `return` wird ein Wert von einer Funktion zurückgegeben. Du könntest zum Beispiel eine Funktion erzeugen, um die Anzahl der Sekunden zu berechnen, die Du bis zu Deinem letzten Geburtstag gelebt hast:

```
def Alter_in_Sekunden(Alter_in_Jahren):
    return Alter_in_Jahren * 365 * 24 * 60 * 60
```

Wenn Du diese Funktion aufrufst, kann der zurückgegebene Wert einer anderen Variablen zugewiesen oder angezeigt werden:

```
>>> Sekunden = Alter_in_Sekunden(9)
>>> print(Sekunden)
283824000
>>> print(Alter_in_Sekunden(12))
378432000
```

try

Mit dem Schlüsselwort `try` fängt ein Code-Block an, der mit den Schlüsselwörtern `exept` und `finally` endet. Man verwendet diese `try/exept/finally`-Codeblöcke, um mit Fehlern in einem Programm umzugehen, sodass das Programm dem Benutzer eine hilfreiche Meldung anzeigt statt eines unfreundlichen Python-Fehlers. Diese Schlüsselwörter werden in diesem Buch nicht verwendet.

while

Das Schlüsselwort `while` ist ein bisschen wie `for`, nur dass eine `for`-Schleife durch einen Bereich (von Zahlen) zählt, eine `while`-Schleife aber so lange läuft, wie ein Ausdruck wahr ist. Sei vorsichtig mit `while`-Schleifen: Falls der Ausdruck immer wahr sein sollte, wird die Schleife nie enden (dies nennt man eine *Endlosschleife*). Hier siehst Du ein Beispiel:

```
>>> x = 1
>>> while x == 1:
        print('Hallo')
```

Wenn Du diesen Code ausführst, wird er unendlich laufen oder mindestens so lange, bis Du die Python-Shell schließt oder Ctrl-C drückst, um die Schleife zu unterbrechen. Der folgende Code wird dagegen neunmal »Hallo« anzeigen (wobei er jedes Mal 1 zu der Variablen `x` addiert, bis `x` nicht mehr weniger als 10 beträgt).

```
>>> x = 1
>>> while x < 10:
        print('Hallo')
        x = x + 1
```

with

Das Schlüsselwort `with` benutzt man mit einem Objekt, um einen Code-Block in ähnlicher Weise wie mit den Schlüsselwörtern `try` und `finally` zu erzeugen. Dieses Schlüsselwort wird in diesem Buch nicht verwendet.

yield

Das Schlüsselwort `yield` ist ein bisschen wie `return`, nur dass man es mit einer bestimmten Klasse von Objekten, den Generatoren, benutzt. Generatoren erzeugen Werte spontan (sie erzeugen also Werte direkt auf Anfrage), sodass sich in diesem Sinne auch die Funktion `range` wie ein Generator verhält. Das Schlüsselwort `yield` kommt in diesem Buch nicht zum Einsatz.

Glossar

Manchmal ist es so, dass Dir beim Programmieren ein neuer Begriff begegnet, mit dem Du nicht viel anfangen kannst. Solche Wissenslücken können einem beim Vorankommen richtig im Wege stehen. Für dieses Problem gibt es aber eine ganz einfache Lösung.

Für solche Fälle, bei denen Dich ein neues Wort oder ein Begriff aufhält, habe ich dieses Glossar erstellt. Darin findest Du Definitionen vieler Programmierbegriffe, die in diesem Buch verwendet werden. Schlag also einfach hier nach, wenn Du auf ein Wort stößt, das Du nicht verstehst.

Animation Das Verfahren zum Zeigen einer Reihe (Sequenz) von Bildern, die so schnell wechseln, dass es aussieht, als ob sich etwas bewege.

aufrufen Den Code einer Funktion ausführen. Wenn wir eine Funktion benutzen, sagen wir, dass wir sie »aufrufen«.

ausführen Etwas Code (wie etwa ein Programm, einen kleinen Code-Abschnitt oder eine Funktion) laufen lassen.

Bedingung Ein Ausdruck in einem Programm, der ein wenig wie eine Frage klingt. Bedingungen prüfen, ob etwas wahr oder falsch ist.

Block Eine Gruppe von Computer-Anweisungen innerhalb eines Programms.

Boolescher Wert Eine Art von Wert, der entweder wahr oder falsch ist. (In Python sind das `True` oder `False` mit großem »`T`« und »`F`«.)

Daten sind meist Informationen, die vom Computer gespeichert und bearbeitet werden.

Dialog Ein Dialog ist üblicherweise ein kleines Fenster in einer Anwendung, in dem Informationen aus dem Kontext stehen (wie etwa eine Warnung, eine Fehlermeldung oder die Aufforderung, etwas einzugeben). Wenn Du beispielsweise eine Datei öffnen möchtest, erscheint normalerweise der Datei-Dialog.

Dimensionen Im Zusammenhang mit der Grafikprogrammierung bedeuten *zweidimensional* oder *dreidimensional* die Arten, wie Bilder auf dem Computermonitor gezeigt werden. Zweidimensionale Grafiken (2D-Grafiken) sind flache Bilder auf einem Monitor, die eine Breite und eine Höhe haben – wie in den alten Zeichentrickfilmen, die Du aus dem Fernsehen kennst. Dreidimensionale Grafiken (3D-Grafiken) sind Bilder auf dem Monitor, die eine Breite, eine Höhe und den Anschein von Tiefe haben – die Art von Grafiken, denen man in realistischeren Computerspielen begegnet.

einbetten Das Ersetzen von Werten innerhalb eines Strings. Die ersetzten Werte nennt man manchmal auch *Platzhalter*.

Eltern Wenn man sich auf Klassen und Objekte bezieht, sind die Eltern die Klasse, von der Funktionen und Variablen geerbt werden. Andersherum gesagt, erbt eine Kinderklasse die Eigenschaften ihrer Elternklasse. Wenn es nicht um Python geht, sind die Eltern diejenigen, die Dir sagen, dass Du Deine Zähne putzen sollst, bevor Du abends zu Bett gehst.

Ereignis Etwas, das auftritt, während das Programm läuft. Beispiele für Ereignisse sind das Bewegen der Maus, das Drücken einer Maustaste oder Tastatureingaben.

Exception Auf Deutsch »Ausnahme«: Eine Art von Fehler, der bei der Ausführung eines Programms auftreten kann.

Fehler Wenn etwas mit einem Programm oder Deinem Computer schiefläuft, ist dies ein Fehler. Wenn Du mit Python programmierst, kannst Du alle möglichen Arten von Meldungen als Reaktion auf einen Fehler sehen. Wenn Du Deinen Code nicht richtig eingibst, kannst Du zum Beispiel einen Einrückungsfehler (`IndentationError`) bekommen.

Frame Eine Folge von Bildern, aus denen eine Animation wird.

Funktion Ein Befehl in einer Computersprache, der meistens aus einer Sammlung von Anweisungen besteht, um eine bestimmte Aktion durchzuführen.

Grad Eine Maßeinheit für Winkel.

Gültigkeitsbereich Der Teil oder Abschnitt eines Programms, in dem eine Variable »gesehen« (oder benutzt) werden kann. (Eine Variable innerhalb einer Funktion ist mitunter außerhalb der Funktion nicht sichtbar.)

hexadezimal Eine Art, Zahlen darzustellen, die vor allem bei der Programmierung verwendet wird. Hexadezimale Zahlen haben als Basis 16, was bedeutet, dass neben den Ziffern 0–9 auch noch die Buchstaben A, B, C, D, E oder F benutzt werden.

horizontal Die Richtungen nach links und rechts auf dem Monitor (dargestellt durch das x).

ID-Nummer Eine Zahl, die eindeutig etwas in einem Programm benennt. Im Modul `tkinter` von Python nimmt man die ID-Nummer, um sich auf Formen zu beziehen, die auf der Leinwand gezeichnet wurden.

Image Ein Bild auf dem Computermonitor.

Import In Python »importierst« Du ein Modul, um es in Deinem Programm zu benutzen. Man könnte auch sagen, dass Du es »einbindest« oder »dazuschaltest«, aber Python-Programmierer reden lieber vom »Importieren«.

initialisieren Wenn Du den Ausgangszustand eines Objekts einrichtest (also die Variablen in einem Objekt bei dessen Erzeugung einrichtest), nennt der Python Programmierer das »initialisieren«.

Installation Der Prozess, in dem Du Dateien einer Softwareanwendung auf Deinen Computer kopierst, damit Du die Anwendung anschließend benutzen kannst.

Instanz Die Instanz einer Klasse ist – anders gesagt – ein Objekt.

Kind Wenn es um Klassen geht, stellen wir uns die Beziehungen zwischen verschiedenen Klassen wie die Beziehungen zwischen Eltern und Kindern vor. Eine Kinderklasse erbt die Eigenschaften seiner Elternklasse.

Klasse Eine Beschreibung oder Definition einer Art von Ding. In der Welt des Programmierens ist eine Klasse eine Sammlung von Funktionen und Variablen.

Klick Druck auf eine der Maustasten, um einen Button auf dem Monitor auszuwählen, eine Menüoption aufzurufen usw.

Kollision Wenn in Computerspielen eine Figur an eine andere oder an ein Objekt auf dem Monitor stößt.

Koordinaten Die Position eines Pixels auf dem Monitor. Die Positionen werden meist als Anzahl der Pixel zur Seite (x) und nach unten (y) auf dem Monitor beschrieben.

Leinwand Ein Bereich des Monitors, auf dem gezeichnet wird. Die Leinwand (`canvas`) ist eine Klasse, die zum Modul `tkinter` gehört.

Modul Eine Gruppe von Funktionen und Variablen.

Null Das Nichtvorhandensein eines Wertes (in Python als `None` bezeichnet).

Objekt Die bestimmte Instanz einer Klasse. Wenn Du ein Objekt einer Klasse erzeugst, reserviert Python dafür einen Teil Deines Computerspeichers, um darin Informationen über ein Mitglied dieser Klasse zu speichern.

Operator Ein Element eines Computerprogramms, das man für mathematische Berechnungen oder zum Vergleichen von Werten benutzt.

Parameter Ein Wert, den man beim Aufruf einer Funktion oder bei der Erzeugung eines Objekts benutzt (wenn Python beispielsweise die Funktion __init__ aufruft). Parameter werden häufig auch als *Argumente* bezeichnet.

Pixel Ein einzelner Punkt auf Deinem Computermonitor – der kleinste Punkt, den Dein Computer zeichnen kann.

Programm Eine Gruppe von Befehlen, die einem Computer sagen, was er zu tun hat.

Schlüsselwort Ein bestimmtes Wort, das von einer Programmiersprache verwendet wird. Schlüsselwörter bezeichnet man auch als *reservierte Wörter*, was im Grunde bedeutet, dass man sie für nichts anderes benutzen kann. (Du kannst beispielsweise kein Schlüsselwort als Variablennamen verwenden.)

Schleife Ein Befehl oder eine Gruppe von Befehlen, die wiederholt werden.

Shell Ein Zugang zum Programm auf Basis der Befehlseingabezeile. Wenn wir in diesem Buch von der »Python-Shell« sprechen, meinen wir die Anwendung IDLE.

Speicher Ein Teil Deines Computers, in dem vorübergehend Informationen abgelegt werden.

Software Eine Sammlung von Computerprogrammen.

Sprite Eine Figur oder Objekt in einem Computerspiel.

String Eine Sammlung alphanumerischer Zeichen (Buchstaben, Zahlen, Satzzeichen und Leerzeichen).

Syntax Die Anordnung und Reihenfolge von Wörtern in einem Programm.

Transparenz Der Bereich eines Bildes, der nicht angezeigt wird und daher nicht überdeckt, was hinter ihm dargestellt wird.

Variable Etwas, in dem Du Werte speichern kannst. Du kannst Dir eine Variable wie ein Etikett vorstellen, das Du auf Informationen »klebst«, die im Computerspeicher stehen. Variablen sind nicht dauerhaft an einen bestimmten Wert gebunden – daher der Name »Variable«, der Wert kann sich also ändern.

vertikal Die Richtungen nach oben und unten auf dem Monitor (dargestellt durch y).

Verzeichnis Der Ort, an dem eine Gruppe von Dateien auf der Festplatte Deines Computers liegt.

Index

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z

Sonderzeichen

Ziffern