

PYTHON

Craig Richardson

PROGRAMMIEREN LERNEN MIT MINECRAFT



Craig Richardson ist ein erfahrener Python-Entwickler und -Trainer. Er hat am Raspberry Pi-Projekt mitgearbeitet und gibt neben seinen IT-Kursen an der Uni auch viele Workshops zur Python-Programmierung in Minecraft.

Craig Richardson

Python programmieren lernen mit Minecraft



dpunkt.verlag

Craig Richardson

Übersetzung: Christian Alkemper, alkemper.com

Lektorat: Boris Karnikowski

Fachlektorat: Tobias Overkamp, Solingen

Copy-Editing: Friederike Daenecke, Zülpich

Herstellung: Susanne Bröckelmann, Heidelberg

Satz: Ulrich Borstelmann, Dortmund

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:

Print 978-3-86490-373-1

PDF 978-3-86491-957-2

ePub 978-3-86491-958-9

mobi 978-3-86491-959-6

1. Auflage 2016

Translation Copyright für die deutschsprachige Ausgabe © 2016 dpunkt.verlag GmbH

Wiebinger Weg 17

69123 Heidelberg

Copyright der amerikanischen Originalausgabe © 2015 by Craig Richardson

Title of English-language original: Learn to Program with Minecraft

Published by No Starch Press, Inc.

ISBN: 978-1-59327-670-6

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

Inhaltsverzeichnis

Einleitung	xi
Warum man programmieren können sollte	xi
Warum Python?	xii
Und warum Minecraft?	xii
Was du in diesem Buch findest	xiii
Online-Ressourcen	xv
Möge das Abenteuer beginnen!	xvi
 Kapitel 1: Die Vorbereitung aufs Abenteuer	 1
Windows-PC einrichten	2
Minecraft installieren	2
Python installieren	4
Java installieren	5
Minecraft-Python-API und Spigot installieren	7
Spigot ausführen und ein Spiel erstellen	9
Der Einstieg in eine neue Welt	11
Offline spielen	12
In den Überlebensmodus umschalten	13
Mac einrichten	15
Minecraft installieren	15
Python installieren	17
Java installieren	18
Minecraft-Python-API und Spigot installieren	19
Spigot ausführen und ein Spiel erstellen	21
Der Einstieg in eine neue Welt	23
Offline spielen	23
In den Überlebensmodus umschalten	24
Raspberry Pi einrichten	25
IDLE kennenlernen	26
Python-Shell kennenlernen	27
Auftritt: Der IDLE-Texteditor	29
Shell oder Editor – wann verwendest du was?	31
Die in diesem Buch verwendeten Prompts	31
Minecraft-Python-Setup testen	32

Kapitel 2: Teleportation mit Variablen	35
Was ist ein Programm?	35
Daten in Variablen speichern	36
Programmiersprachen und ihre Struktur.	37
Syntaxregeln für Variablen	39
Variablenwerte ändern	39
Integer	40
Float-Variablen	48
Teleportation mit dem time-Modul verlangsamen	50
Debuggen	54
Was du gelernt hast	58
 Kapitel 3: Schneller bauen und reisen – Mathe macht’s möglich	 59
Ausdrücke und Anweisungen	59
Operatoren	60
Addition	61
Subtraktion	65
Mathematische Operatoren in Argumenten verwenden	67
Multiplikation	73
Division	73
Potenzrechnung	76
Klammern und die Reihenfolge von Rechenoperationen	77
Praktische Rechentricks	78
Kurzformen für Operatoren	78
Mehr Spaß mit Zufallszahlen	79
Was du gelernt hast	82
 Kapitel 4: Chatten mit Strings	 83
Was sind Strings?	84
Die Funktion print()	84
Die Funktion input()	87
Strings verknüpfen	90
Zahlen in Strings umwandeln	91
Integer und Floats verketten	92
Strings mit int() in Integer konvertieren	94
Fehlermeldung? Lieber nicht!	97
Was du gelernt hast	103

Kapitel 5: Stimmt's oder nicht? Boolesche Variablen 105

Grundlagen zu booleschen Variablen	106
Boolesche Werte verketteten	108
Vergleichsoperatoren	108
Ist gleich	109
Ist ungleich	112
Größer als/kleiner als	114
Größer oder gleich und kleiner oder gleich	116
Logische Operatoren	120
and	121
or	123
not	125
Bearbeitungsreihenfolge bei logischen Operatoren	128
Liegt meine Zahl zwischen zwei anderen?	129
Was du gelernt hast	132

Kapitel 6: Minispiele mit if-Anweisungen basteln 133

if-Anweisungen verwenden	134
else-Anweisungen	138
elif-Anweisungen	141
elif-Anweisungen verketteten	145
Verschachtelte if-Anweisungen	148
Mit if-Anweisungen einen Wertebereich testen	151
Boolesche Operatoren und if-Anweisungen	155
Was du gelernt hast	159

Kapitel 7: Tanztees und Blumenkorsos mit while-Schleifen 161

Eine einfache while-Schleife	161
Schleifen mit einer Zählervariablen steuern	166
while-Endlosschleifen	170
Kunstvolle Bedingungen	172
Boolesche Operatoren und while-Schleifen	176
Bereichsprüfung in while-Schleifen	176
Verschachtelte if-Anweisungen und while-Schleifen	179
Eine while-Schleife mit break beenden	182
while-else-Anweisungen	184
Was du gelernt hast	188

Kapitel 8: Meine geheime Superkraft: Funktionen	189
Eigene Funktionen definieren	190
Funktionen aufrufen	191
Funktionen arbeiten mit Argumenten	192
Refactoring	195
Kommentieren mit Docstrings	198
Zeilenwechsel in Argumenten	199
Rückgabewerte von Funktionen	200
if-Anweisungen und while-Schleifen in Funktionen verwenden.	204
if-Anweisungen	205
while-Schleifen	208
Lokale und globale Variablen	211
Was du gelernt hast	216
 Kapitel 9: Treffer, versenkt – dank Listen und Wörterbüchern	 217
Listen verwenden	218
Listenelemente aufrufen	219
Listenelemente ändern	220
Listen manipulieren	223
Element anhängen	223
Element einfügen	223
Elemente löschen	224
Strings als Listen behandeln	227
Tupel	228
Variablen mit Tupeln festlegen	229
Tupel zurückgeben	232
Weitere nützliche Listenfunktionen	232
Listenlänge	233
Zufälliges Element auswählen	236
Liste kopieren	238
Elemente und if-Anweisungen	240
Wörterbücher	244
Wörterbuch definieren	244
Zugriff auf Elemente in Wörterbüchern	245
Elemente im Wörterbuch ändern oder hinzufügen	248
Elemente aus Wörterbüchern löschen	249
Was du gelernt hast	252

Kapitel 10: Wie von Geisterhand – oder mit der for-Schleife **253**

Eine einfache for-Schleife	253
Die Funktion range()	256
Lustige Sachen mit range() machen	259
Weitere Listenfunktionen	260
Schleife über ein Wörterbuch erstellen	265
for-else-Schleifen	267
for-else-Schleife unterbrechen	268
Verschachtelte for-Schleifen und mehrdimensionale Listen	270
In zwei Dimensionen denken	270
Werte in 2-D-Listen aufrufen	276
2-D-Listen mit Schleifen erzeugen	279
Und jetzt in drei Dimensionen	283
3-D-Listen ausgeben	284
Werte in 3-D-Listen aufrufen	290
Was du gelernt hast	297

Kapitel 11: Gebäude speichern und laden mit Dateien und Modulen **299**

Dateien verwenden	300
Eine Datei öffnen	300
In eine Datei schreiben und sie speichern	302
Datei lesen	303
Dateien zeilenweise lesen	304
Teil 1: To-Do-Liste schreiben	305
Teil 2: To-Do-Liste anzeigen	307
Module verwenden	308
Das pickle-Modul	309
pickle importieren	309
Eine Funktion mit der from-Klausel importieren	311
Alle Funktionen mit * importieren	312
Einen Spitznamen für ein Modul erstellen	313
Teil 1: Gebäude speichern	314
Teil 2: Gebäude laden	318
Mit dem shelve-Modul viele Daten speichern	320
Datei mit shelve öffnen	320
Elemente mit shelve hinzufügen, bearbeiten und abrufen	321
Teil 1: Struktur in einer Sammlung speichern	322
Teil 2: Struktur aus einer Sammlung laden	323

Weitere Module mit pip installieren	327
pip unter Windows verwenden	327
pip auf einem Mac oder dem Raspberry Pi verwenden	328
Beispiel: Das pip-Modul Flask verwenden	329
Was du gelernt hast	332

Kapitel 12: Jetzt geht's ans Eingemachte: OOP **335**

Grundlagen der objektorientierten Programmierung	336
Eine Klasse erstellen	336
Ein Objekt erstellen	337
Auf Attribute zugreifen	338
Grundsätzliches zu Methoden	341
Mit Methoden Werte zurückgeben	347
Mehrere Objekte erstellen	350
Klassenattribute	353
Grundsätzliches zur Vererbung	356
Eine Klasse vererben	357
Neue Methoden zu untergeordneten Klassen hinzufügen	358
Methoden und Attribute überschreiben	362
Was du gelernt hast	367

Index **369**

Einleitung

Willkommen zu *Python programmieren lernen mit Minecraft*. In diesem Buch erfährst du, wie du mit einer Programmiersprache namens Python Programme schreibst und damit beeinflusst, was in deiner Minecraft-Welt passiert. Du wirst allerlei über das Programmieren erfahren und das Gelernte dann verwenden, um Häuser zu bauen, Minispiele zu erstellen und aus langweiligen Gegenständen neue und aufregende Items zu machen.

Wenn du das Buch ausgelesen hast, solltest du eigentlich alles wissen, was du brauchst, um auch die abgefahrensten Ideen umzusetzen.

Programmieren ist kreativ und fantasievoll – ebenso wie Minecraft. Mit den Fertigkeiten, die du in diesem Buch lernst, kannst du alles Mögliche basteln: Games, Apps und auch nützliche Tools. Das geht weit über die Programmierung mit Minecraft hinaus. Dieses Buch ist der Anfang einer Reise, an deren Ende du ein herausragender Programmierer und ein Meister in Sachen Minecraft sein wirst!

Warum man programmieren können sollte

Es gibt eine Reihe von Gründen dafür, sich mit dem Programmieren zu beschäftigen. Einer der wichtigsten ist: Du lernst, wie man Probleme löst. Du wirst hier herausfinden, wie du ein großes Problem in viele kleinere Teilprobleme zerlegen kannst, die für sich genommen wesentlich einfacher zu lösen sind. Bei vielen Problemen, die du lösen musst, wirst du kreativ werden und verschiedene Ansätze ausprobieren müssen.

Ein weiterer Vorteil: Du lernst logisches Denken, um die Struktur und Abläufe deiner Programme besser planen und nachvollziehen zu können. Und auch dann, wenn du gerade nicht programmierst, sind Fähigkeiten

wie Strategien zur Problemlösung, Kreativität und logisches Denken in vielen Situationen sehr wertvoll.

Eine Karriere als Programmierer ist auch sehr lohnenswert. Tag für Tag hast du dann die Aufgabe, anspruchsvolle Probleme kreativ zu lösen. Aber sogar dann, wenn du eine andere Laufbahn einschlägst, kann das Programmieren als Hobby ebenso inspirierend wie unterhaltsam sein. Kein Witz: Ich selbst habe am Anfang auch nur hobbymäßig programmiert – und heute lebe ich davon.

Vor allem aber: Programmieren macht Spaß. Richtig viel Spaß. Es gibt nur wenig, was befriedigender ist, als zu sehen, dass das Programm, das du geschrieben hast, genau das tut, was es tun soll!

Warum Python?

Und warum genau sollte man das Programmieren jetzt mit Python lernen? Python ist für Einsteiger in Sachen Computerprogrammierung hervorragend geeignet. Es ist ebenso leicht zu lesen wie zu schreiben, dabei aber leistungsfähig genug, um wirklich ernst zu nehmende Programme zu erstellen. Und: Python ist eine der beliebtesten Programmiersprachen weltweit.

Und warum Minecraft?

Minecraft ist extrem populär, denn es spornt die Kreativität an und macht gleichzeitig einen Riesenspaß. In deiner Minecraft-Welt kannst du bauen, was immer du möchtest. Lass deiner Fantasie freien Lauf. Durch die Integration von Minecraft mit deinen eigenen Python-Programmen kannst du Minecraft noch besser steuern und dir ganz neue kreative Möglichkeiten eröffnen. Du wirst Dinge tun, die mit Minecraft allein gar nicht möglich sind – etwa ein riesiges Gebäude in nur wenigen Sekunden bauen.

Manchmal ist es schwierig, den Einstieg in das Programmieren zu finden, denn man muss eine Menge Code lernen, der so gar nichts Spannendes tun will. Aber bei der Kombination von Python und Minecraft siehst du die Resultate deiner coolen Programme sofort in deiner Minecraft-Welt.

Was du in diesem Buch findest

Jedes Kapitel legt den Schwerpunkt auf ein einzelnes Konzept in Python. Auf diese Weise kannst du im Verlauf des Buchs deine Kenntnisse zur Python-Programmierung nach und nach ausbauen. Du findest in jedem Kapitel Erläuterungen zu Funktionen in Python, Beispiele, die Python in Aktion zeigen, und Minecraft-Missionen. Im Verlauf dieser Missionen schreibst du Programme, die in Interaktion mit Minecraft treten. Ich gebe jeweils ein Codegerüst vor, und deine Aufgabe besteht darin, die Lücken aufzufüllen und das Programm zu vervollständigen. So wirst du im Laufe der Zeit Fähigkeiten zur Problemlösung entwickeln, die jeder gute Programmierer kennen muss.

Sehen wir uns doch einmal an, was dich in den einzelnen Kapiteln erwartet.

- **Kapitel 1: Die Vorbereitung aufs Abenteuer.** Hier findest du Hilfe beim Einrichten von Python und Minecraft, denn ohne geeignete Ausrüstung kommt auch der stärkste Held nicht voran.
- **Kapitel 2: Teleportation mit Variablen.** Dieses Kapitel zeigt dir, wie du deinen Spieler im Handumdrehen teleportierst – ganz einfach durch Manipulation von Variablen. Du erfährst, was Variablen überhaupt sind und wie sie Werte in deinen Programmen speichern. So kannst du dir sogar selbst ganz abgefahrenere Teleportationsskills verpassen und dich auf eine magische Reise durch deine Welt begeben.
- **Kapitel 3: Schneller bauen und reisen – Mathe macht's möglich.** In diesem Kapitel wird gezeigt, wie man sich selbst Superkräfte verpasst und eigene Kreationen mit extrem hoher Geschwindigkeit zusammenbaut – mit ganz normaler Mathematik. Möchtest du ein Minecraft-Haus im Bruchteil einer Sekunde bauen? Kein Problem mit Mathe. Möchtest du mit einem einzigen Sprung in die höchsten Sphären gelangen? Auch das wird mit Rechenoperatoren zum Kinderspiel.
- **Kapitel 4: Chatten mit Strings.** Dieses Kapitel enthält alle wichtigen Informationen zu Strings. Damit bastelst du dir einen interaktiven Chat. In der Programmierung bezeichnet der Begriff *String* eine Textzeichenfolge. In Python kannst du Programme schreiben, die dir und deinen Mitspielern in Minecraft Nachrichten übermitteln.
- **Kapitel 5: Stimmt's oder nicht? Boolesche Variablen.** In diesem Kapitel wird dir gezeigt, wie du boolesche Variablen und Logik verwendest, damit deine Programme Fragen beantworten können. Anders formuliert: Du kannst deinen Programmen beibringen, dir zu sagen, ob etwas wahr oder falsch ist. Deine Minecraft-Programme

können alle Arten von Fragen beantworten: Bin ich unter Wasser? Sitze ich in einem Baum? Ist mein Haus vielleicht gar nicht so weit weg?

- **Kapitel 6: Minispiele mit if-Anweisungen basteln.** In diesem Kapitel wird die boolesche Logik auf ein höheres Niveau gehoben. Es beschreibt, wie man mit if-Anweisungen Programme erstellt, die Entscheidungen auf Grundlage der übergebenen Daten treffen. Hast du dir jemals gewünscht, dass sich in Minecraft ein Geheimgang öffnet, wenn du einen bestimmten Block an einer bestimmten Stelle einfügst? So etwas funktioniert mit if-Anweisungen!
 - **Kapitel 7: Tanztees und Blumenkorsos mit while-Schleifen.** Hier lernst du eine sehr coole Möglichkeit kennen, in deinen Programmen Dinge zu wiederholen – mit Schleifen. Du kannst Code automatisieren und damit beeindruckende Sachen zaubern. Stelle dir eine Armada von Blumen vor, die dem Spieler folgt, oder eine verzauberte Tanzfläche, die in unterschiedlichen Farben blinkt. Das sind einige meiner Lieblingsprogramme, mit denen ich vor anderen so gerne angebe.
 - **Kapitel 8: Meine geheime Superkraft: Funktionen.** Dieses Kapitel beschreibt das Bauen von Wäldern und Städten im Handumdrehen – mit Funktionen. Außerdem erfährst du, wie du dir das Leben als Programmierer einfacher machst, indem du Teile deiner Programme wiederverwendest.
 - **Kapitel 9: Treffer, versenkt – dank Listen und Wörterbüchern.** Minispiele mit Listen erstellen? Kein Problem. Listen sind ein extrem mächtiges Programmierkonzept, denn damit lassen sich wichtige Informationen an zentraler Stelle speichern. Mit Listen sorgst du dafür, dass sich dein Programm an alle Blöcke erinnert, die du mit deinem Schwert getroffen hast. Und mit einigen wenigen zusätzlichen Codezeilen machst du daraus sogar ein Minispiel. Das hat doch was!
 - **Kapitel 10: Wie von Geisterhand – oder mit der for-Schleife.** Du möchtest Strukturen (z. B. Pyramiden) bauen? Dabei hilft dir die for-Schleife. Mit for-Schleifen kannst du »Pixel Art« zeichnen oder deine Minecraft-Gebäude duplizieren. Stell dir vor, du baust dir eine prächtige Statue – und fertigst dann eine Kopie nach der anderen davon an, bis du am Ende eine ganze Armee von Statuen hast!
 - **Kapitel 11: Gebäude speichern und laden mit Dateien und Modulen.** Dieses Kapitel erklärt, wie du mit deinen Programmen Dateien anlegst und bearbeitest, um erstellte Objekte zu speichern und sie dann in anderen Welten wieder zu laden. Anders gesagt: Du wandelst deine Gebäude in Dateien um, die du dann buchstäblich überallhin
-

mitnehmen kannst. Möchtest du die unfassbar schöne Villa speichern, die du gebaut hast? Kein Problem! Speichere sie als Datei, dann hast du sie immer dabei.

- **Kapitel 12: Jetzt geht's ans Eingemachte: Objektorientierte Programmierung (OOP).** In diesem Kapitel werden einige anspruchsvollere Themen vorgestellt: Klassen, Objekte und Vererbung. Wenn du dieses Kapitel abgeschlossen hast, bist du ein echter Python-Meister. Am Anfang jeder Mission steht ein bisschen Code, mit dem ein Gebäude erstellt wird. Danach kannst du mit Klassen, Objekten und Vererbung ganz einfach Kopien und Varianten erstellen, z. B. Dörfer oder Hotels. Und das geht mit nur wenigen zusätzlichen Codezeilen.
- Die offizielle **Minecraft Wikipedia** (<http://minecraft-de.gamepedia.com>) ist eine hervorragende Referenz – nutze das Suchen-Feld auf ihrer Startseite, um deine Informationen zu recherchieren. Und setze in deinem Browser schon mal ein Lesezeichen auf diese Übersicht über die Minecraft-Blöcke und ihre IDs: <http://minecraft-de.gamepedia.com/Datenwert>. Du wirst diese Übersicht oft brauchen, wenn du in Deinem Programmcode Blöcke direkt über ihre ID ansprechen willst. Detaillierte Einzelheiten wie etwa über die Farben von Wollblöcken findest Du aber am besten über die Suchfunktion der Minecraft Wikipedia.

Online-Ressourcen

Alle Programmcodes und Ressourcen für dieses Buch sind auf der Begleitwebsite erhältlich, die du über <https://www.nostarch.com/pythonwith-minecraft/> erreichst. Lade dir den Code für die Minecraft-Missionen herunter, wenn du feststeckst und die Lösungen herausfinden willst – oder wenn du den Code bearbeiten möchtest, um selbst coole Programme zu schreiben. Du kannst auch die Setup-Dateien herunterladen. Wie das Setup funktioniert – d. h., wie du die benötigten Programme einrichten musst –, beschreibe ich ausführlich in Kapitel 1.

Da dieses Buch aus dem Englischen übersetzt wurde, noch eine Erklärung zum darin abgedruckten Programmcode (gilt nicht für die Online-Ressourcen): Da sowohl Python als auch die Minecraft-API viele englische Schlüsselwörter verwenden, wurden die Funktionsnamen und Variablen im Code nicht übersetzt (und viele Textstrings auch nicht, ich traue dir zu, dass Du auch so verstehst, was gemeint ist). Kommentare hingegen wurden übersetzt, denn sie helfen dir zu verstehen, was das jeweilige

Stück Programmcode macht. Schließlich wirst du – wieder nur im Buch – öfter dieses Zeichen am Ende von Programmzeilen sehen: ↵. Es sagt nur aus, dass die laufende Zeile an dieser Stelle aus Platzgründen umbrochen wurde. Wenn du den Code abtippst, brauchst du dieses Zeichen nicht zu beachten, sondern tippst einfach die Zeile zu Ende.

Möge das Abenteuer beginnen!

Ich hoffe, du kannst es genauso wenig erwarten wie ich, endlich loszulegen. Ich habe beim Verfassen dieses Buchs und beim Erstellen der verschiedenen Minecraft-Missionen, mit denen ich die Kunst der Programmierung veranschaulichen möchte, wirklich viel Spaß gehabt. Fangen wir also an.

Kapitel 1

Die Vorbereitung aufs Abenteuer

Damit du coole Python-Programme für deine Minecraft-Welt programmieren kannst, musst du auf dem Computer erst einmal Minecraft, Python und einige andere Dinge einrichten. In diesem Kapitel zeige ich dir, wie du alle benötigten Programme installierst und ausführst.

Du kannst Minecraft auf einem Windows-PC oder einem Mac verwenden oder die *Minecraft: Pi Edition* auf einem Raspberry-Pi-Computer benutzen. Wenn du einen Windows-PC hast, lies einfach weiter. Benutzt du einen Mac, dann springe zum Abschnitt »Mac einrichten« auf Seite 15. Für die Verwendung eines Raspberry Pi schließlich sind die notwendigen Angaben unter »Raspberry Pi einrichten« auf Seite 25 verfügbar.

Hinweis

Informationen zu anderen Plattformen und Updates dieser Anleitungen findest du (in englischer Sprache) unter <https://www.nostarch.com/pythonwithminecraft/>.

Windows-PC einrichten

Damit du Minecraft via Python steuern kannst, musst du fünf Komponenten installieren:

- Minecraft
- Python 3
- Java
- Minecraft-Python-API
- Spigot Minecraft Server

In diesem Abschnitt beschreibe ich die Installation dieser Komponenten auf deinem Computer. Mit Minecraft fangen wir an.

Minecraft installieren

Solltest du Minecraft bereits besitzen und die aktuelle Version auf deinem PC installiert haben, dann fahre fort bei »Python installieren« auf Seite 4. Wenn du nicht genau weißt, ob sich die aktuelle Minecraft-Version auf deinem Rechner befindet, dann gehe wie hier beschrieben vor, um sie zu installieren.

Hast du das Spiel noch gar nicht, dann kannst du es auf der offiziellen Minecraft-Website <https://minecraft.net/> kaufen. Bitte ggf. einen Erwachsenen, dir dabei zu helfen. Notiere dir außerdem den beim Kauf von Minecraft verwendeten Benutzernamen und das Passwort, denn diese Angaben benötigst du später für die Anmeldung.

Wenn du Minecraft gekauft hast, gehst du wie folgt vor, um es auf deinem PC zu installieren:

1. Rufe in deinem Browser die Seite <https://minecraft.net/download> auf.
 2. Klicke im Abschnitt **Minecraft for Windows** auf den Link **Minecraft.msi**, um die gleichnamige Datei herunterzuladen. Wenn du gefragt wirst, ob du die Datei speichern oder öffnen möchtest, klicke auf **Datei speichern**.
 3. Warte, bis die Datei vollständig heruntergeladen ist, und öffne sie dann. Wirst du in einem Dialogfeld gefragt, ob du diese Datei ausführen möchtest, klicke auf **Ausführen**. Keine Sorge: Diese Datei ist sicher.
 4. Nach dem Öffnen des Minecraft Setup-Assistenten klicke auf **Next**. Danach klickst du noch einmal auf **Next** und dann auf **Install**.
-

5. Unter Umständen wirst du gefragt, ob du Minecraft wirklich installieren möchtest. Und ob du das willst! Klicke auf **Yes**. Jetzt hast du ein bisschen Zeit, während Minecraft installiert wird. Ich habe während der Installation ein Glas Wasser getrunken und ein Plätzchen verzehrt.
6. Wenn die Installation abgeschlossen ist, klickst du auf **Finish**.

Nun sollte Minecraft auf deinem System installiert sein.

Und jetzt? Jetzt wollen wir natürlich Minecraft spielen. Nimm dir vorher noch ein paar Minuten Zeit, um die Einrichtung durchzuführen.

1. Um Minecraft zu öffnen, klickst du auf das **Startmenü** oder drückst auf die Windows-Taste auf deiner Tastatur. Suche Minecraft aus der Liste der angezeigten Programme heraus, und klicke auf das Icon.
2. Jetzt wird Minecraft gestartet und installiert möglicherweise erst einmal Updates.
3. Daraufhin öffnet sich das Anmeldefenster. Gib den Benutzernamen und das Passwort ein, die du beim Kauf von Minecraft verwendet hast, und klicke auf **Log In**.

Für die Beispiele in diesem Buch musst du mit der Minecraft-Version 1.9 arbeiten (ansonsten kannst du natürlich die jeweils aktuellste Version verwenden).

4. Klicke dazu im Launcher unten links auf die Schaltfläche **Edit Profile**.
5. Im Bereich **Version Selection** wird dir im Listenfeld **Use version** angezeigt, dass die aktuellste Version verwendet wird. Wähle hier die **release 1.9** aus und klicke dann rechts unten auf die Schaltfläche **Save Profile**.
6. Klicke nun auf **Play**. Bevor es richtig losgeht, wird Minecraft noch einige weitere Updates herunterladen.
7. Abschließend klickst du auf **Single Player ▶ Create New World**. Gib deiner Welt einen Namen deiner Wahl, und klicke dann auf **Create New World**. Nun wird die Welt erstellt und du kannst nach Herzenslust darin spielen.

Viel Spaß dabei! Wenn du noch nie Minecraft gespielt hast, probiere erst einmal ein bisschen herum, bis es in deiner Minecraft-Welt dunkel wird. Hüte dich vor den Monstern! Wenn du Minecraft später mit Python nutzt, wirst du eine Multiplayer-Welt verwenden, die anders sein wird als diese Welt. Wir kommen in »Spigot ausführen und ein Spiel erstellen« auf Seite 9 darauf zurück.

Doch nun wieder an die Arbeit. Jetzt wollen wir Python installieren. Um den Mauszeiger wieder aus Minecraft herauszuholen, drückst du einfach **Esc** auf deiner Tastatur. Schließe Minecraft, bevor du mit der Installation fortfährst.

Python installieren

Python ist die Programmiersprache, die du in diesem Buch erlernen wirst. Und die wollen wir jetzt installieren.

1. Rufe in deinem Browser die Seite <http://www.python.org/downloads/> auf.
2. Klicke auf die Schaltfläche **Download Python 3.5.0**. Suche den Link, der mit **Windows x86-64 executable installer** oder **Windows x86 executable installer** beschriftet ist (je nach Windows-Version) und klicke darauf, um den Installer herunterzuladen. (3.5.0 ist zu dem Zeitpunkt, da ich dies schreibe, die aktuelle Version, womöglich wird dir aber schon eine höhere Versionsnummer angezeigt. Installiere in jedem Fall die aktuellste Version.)
3. Nun wird der Download von Python gestartet. Wenn Du gefragt wirst, ob du die Datei speichern oder öffnen möchtest, klicke auf **Datei speichern**.
4. Wenn du die Datei heruntergeladen hast, doppelklicke darauf. Wirst du in einem Dialogfeld gefragt, ob du diese Datei ausführen möchtest, klicke auf **Ausführen**.
5. Nach dem Starten des Installationsprogramms markierst du auf dem Startbildschirm das Kontrollkästchen **Add Python 3.5 to Path** (siehe Bild 1.1). Dann klickst du auf **Install Now**.

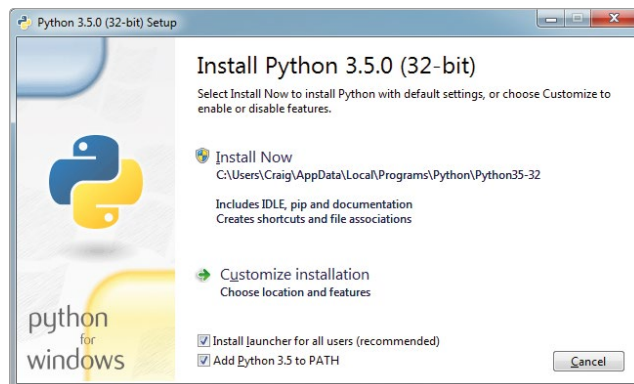


Bild 1.1: »Add Python 3.5 to PATH« muss aktiviert werden.

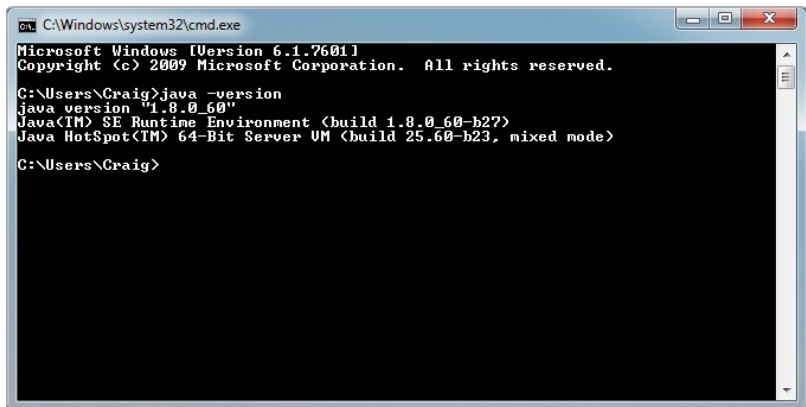
6. Möglicherweise wirst du in einem Dialogfeld gefragt, ob du die Installation des Programms auf dem Computer zulassen möchtest. Klicke auf **Yes**, und warte dann, bis die Installation von Python abgeschlossen ist. Nach dem Starten der Installation bin ich kurz aufgestanden, um das Fenster zu schließen; als ich wieder Platz nahm, war die Installation bereits beendet.
7. Klicke auf **Finish**. Python ist nun installiert.

Java installieren

Nach der erfolgreichen Installation von Minecraft und Python musst du einige Einstellungen vornehmen, damit die beiden Programme miteinander kommunizieren können. Zu diesem Zweck verwendest du ein Programm namens Spigot, das aber erst funktioniert, wenn Java auf deinem Computer installiert ist. Und genau deswegen kümmern wir uns jetzt erst mal um Java.

Zunächst überprüfen wir, ob Java nicht sogar bereits vorhanden ist.

1. Klicke auf das **Startmenü** oder drücke auf die Windows-Taste auf deiner Tastatur. In das Suchfeld gibst du dann **cmd** ein. Das gleichnamige Programm öffnet sich.
2. Nun wird dir ein Fenster mit einem schwarzen Hintergrund und einer Eingabeaufforderung angezeigt (meine heißt `C:\Users\Craig>`). An dieser Eingabeaufforderung gibst du **java -version** ein und drückst die **Eingabetaste**.
3. Wird nun eine Meldung wie die in Bild 1.2 angezeigt, dann ist Java auf deinem System installiert. In diesem Fall kannst du mit dem Abschnitt »Minecraft-Python-API und Spigot installieren« auf Seite 7 fortfahren.
4. Erhältst du hingegen eine Meldung, dass Java nicht erkannt wurde, installiere es so, wie ich es gleich zeige.



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Craig>java -version
java version "1.8.0_60"
Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)

C:\Users\Craig>
```

Bild 1.2: Meldung über ein installiertes Java nach Eingabe des Befehls `java -version`

Zur Installation von Java führst du die folgenden Schritte aus:

1. Rufe in deinem Browser die Seite <http://www.java.com/de/download/> auf.
2. Klicke auf die Schaltfläche **Kostenloser Java-Download**. Danach klickst du auf die Schaltfläche **Einverstanden und mit kostenlosem Download beginnen**.
3. Wenn du die Datei heruntergeladen hast, doppelklicke darauf. Wird daraufhin in einem Dialogfeld gefragt, ob du zulassen möchtest, dass durch das Programm Änderungen an deinem Computer vorgenommen werden, klickst du einfach auf **Ja**.
4. Nun öffnet sich der Installer. Klicke auf **Installieren**.
5. Und jetzt aufgepasst! Wenn sich eine Seite öffnet, auf der du gefragt wirst, ob du ein weiteres Programm installieren möchtest – die Ask-Suche-App, eine Yahoo!-Symbolleiste oder sonst irgendetwas –, dann entferne das Häkchen aus dem jeweiligen Kästchen, um zu vermeiden, dass dieses weitere Programm installiert wird. Denn was auch immer es ist: Du wirst es nicht brauchen.
6. Außerdem wirst du gefragt, ob Yahoo! als Startseite in deinem Browser festgelegt werden soll. Ich bezweifle, dass du das möchtest. Wähle also **Browsers Einstellungen nicht ändern** aus, und klicke dann auf **Weiter**.
7. Warte, bis Java installiert ist. Ich habe so lange eine kurze Nachricht an einen Kumpel geschrieben. Klicke auf **Schließen**, um den Vorgang abzuschließen.

Nun wollen wir sehen, ob Java ordnungsgemäß installiert wurde.

1. Klicke auf das **Startmenü**, und gib **cmd** in das Suchfeld ein. Das gleichnamige Programm wird geöffnet.
2. Im CMD-Fenster gibst du **java -version** an der Eingabeaufforderung ein und drückst die **Eingabetaste**.
3. Wird nun eine Meldung wie die in Bild 1.2 angezeigt, dann ist Java auf deinem System ordnungsgemäß installiert. Erhältst du hingegen die Fehlermeldung »Der Befehl "Java" ist entweder falsch geschrieben oder konnte nicht gefunden werden«, dann ist die Installation fehlgeschlagen. In diesem Fall solltest du noch einmal versuchen, Java zu installieren. Sollte der Fehler weiterhin auftreten, dann suche auf <http://www.java.com/de/download/help/path.xml> nach weiteren Informationen.

Das ist alles! Java ist eingerichtet und kann zur Ausführung des Minecraft-Servers verwendet werden. Diesem wollen wir uns nun zuwenden.

Minecraft-Python-API und Spigot installieren

Nächster Schritt: Wir müssen die Minecraft-Python-API und den Minecraft-Server auf deinem Computer installieren.

»API« ist die Abkürzung für *Application Programming Interface* (dt. Anwendungsprogrammierschnittstelle). Über eine solche Schnittstelle können Programme mit Anwendungen kommunizieren, die von anderen Leuten geschrieben wurden. In diesem Fall erlaubt es die Minecraft-Python-API, dass deine in Python geschriebenen Programme mit Minecraft kommunizieren können. Du könntest beispielsweise ein Python-Programm verfassen, das Minecraft über diese API anweist, einen Block im Spiel zu erstellen oder die Position des Spielers zu ändern.

Im normalen Singleplayer-Modus von Minecraft werden APIs nicht unterstützt, aber deine Programme werden mit einem Minecraft-Server interagieren, der wiederum mit APIs etwas anfangen kann. Minecraft-Server werden in der Regel online genutzt, damit viele Leute gemeinsam in derselben Welt spielen können. Du kannst den Server aber auch auf deinem eigenen Computer laufen lassen und für dich allein spielen. Multiplayer- und Singleplayer-Server gestatten die Verwendung einer API mit Minecraft. In diesem Buch werden wir einen Singleplayer-Server namens Spigot auf deinem Computer installieren.

Jetzt weißt du, wofür eine API und ein Server gut sind. Installieren wir sie also jetzt auf deinem Computer. Ich habe dafür ein praktisches Download-Paket zusammengestellt – damit gelingt die Installation im Handumdrehen. Führe einfach die folgenden Schritte aus:

1. Rufe in deinem Browser die Seite <https://www.nostarch.com/python-withminecraft/> auf, und lade die Datei *Minecraft Tools.zip* für Windows herunter.
2. Wenn der Download der Datei abgeschlossen ist, klickst du mit der rechten Maustaste darauf und wählst **Alles extrahieren** aus. Danach musst du angeben, wo die extrahierten Dateien gespeichert werden sollen. Klicke auf die Schaltfläche **Durchsuchen**, und navigiere zum Ordner *Eigene Dateien*. Klicke auf die Schaltfläche **Neuer Ordner**, und nenne den neuen Ordner *Minecraft Python*. Wähle diesen Ordner dann aus, und klicke auf **OK**. Klicke auf **Extrahieren**, um das Archiv zu entpacken.
3. Navigiere nun zum Ordner *Minecraft Python* im Ordner *Eigene Dateien*. Dort sollten die extrahierten Dateien aufgelistet sein.
4. Öffne den Ordner *Minecraft Tools* (dessen Inhalt ist in Bild 1.3 wiedergegeben).
5. Doppelklicke auf die Datei *Install_API*. Nun wird ein neues Fenster geöffnet und die Minecraft-Python-API installiert. Wird ein Warnhinweis angezeigt, dann klicke auf **Trotzdem ausführen**.
6. Nach Abschluss der Installation drückst du eine beliebige Taste, um den Vorgang zu beenden.

Hinweis

Wenn du eine Fehlermeldung erhältst, laut der »pip« nicht gefunden wurde, bedeutet dies, dass Python nicht richtig installiert wurde. Installiere Python in diesem Fall neu wie in »Python installieren« auf Seite 4 beschrieben. Vergiss dabei nicht, das Kästchen **Add Python 3.5 to Path** zu markieren.

Die Minecraft-Python-API und der Minecraft-Server sind nun installiert. Der abschließende Schritt besteht darin, den Server zu starten. Das tun wir im nächsten Abschnitt.

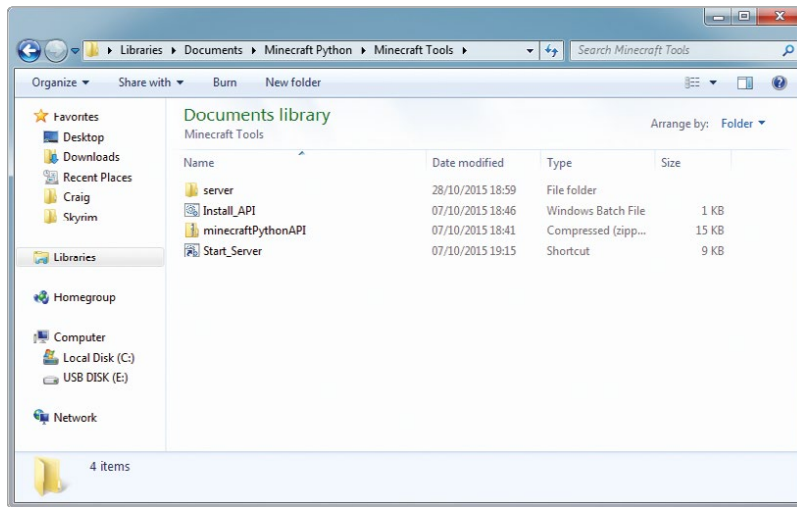
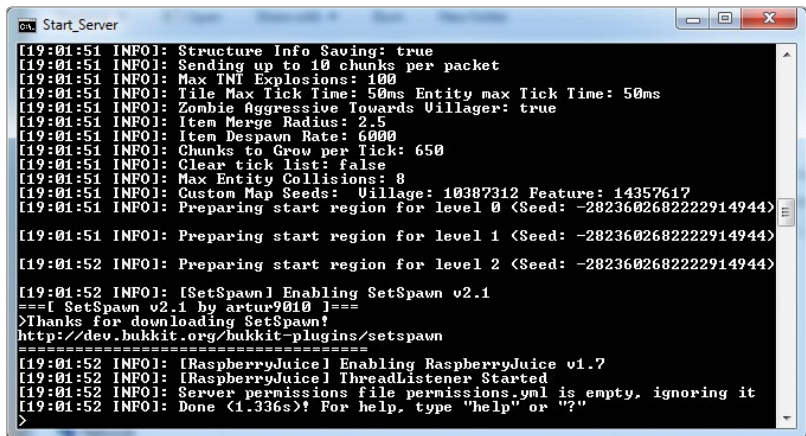


Bild 1.3: Der Ordner »Minecraft Tools«

Spigot ausführen und ein Spiel erstellen

Bei der ersten Ausführung erstellt Spigot eine Minecraft-Welt. Zum Starten von Spigot führst du die folgenden Schritte aus:

1. Navigiere zu deinem Ordner *Minecraft Python*, und öffne den Unterordner *Minecraft Tools* und darin den Ordner *Server*.
2. Doppelklicke im Ordner *Server* auf die Datei *start.bat*. Wenn du gefragt wirst, ob du das Programm wirklich ausführen möchtest, klicke auf **Ausführen**.
3. Spigot startet nun den Minecraft-Server (sollte dir eine Anfrage der Windows-Firewall angezeigt werden, so kannst du diese einfach bestätigen – wahrscheinlich musst du dein Passwort eingeben). Dabei wird ein Fenster geöffnet, in dem eine Menge Text angezeigt wird, während Spigot das Spiel für dich generiert. Ist Spigot fertig, dann sollte der Bildschirm in etwa so aussehen wie in Bild 1.4. Bitte schließe dieses Fenster nicht.



```
[19:01:51 INFO]: Structure Info Saving: true
[19:01:51 INFO]: Sending up to 10 chunks per packet
[19:01:51 INFO]: Max TNT Explosions: 100
[19:01:51 INFO]: Tile Max Tick Time: 50ms Entity max Tick Time: 50ms
[19:01:51 INFO]: Zombie Aggressive Towards Villager: true
[19:01:51 INFO]: Item Merge Radius: 2.5
[19:01:51 INFO]: Item Despawn Rate: 6000
[19:01:51 INFO]: Chunks to Grow per Tick: 650
[19:01:51 INFO]: Clear tick list: false
[19:01:51 INFO]: Max Entity Collisions: 8
[19:01:51 INFO]: Custom Map Seeds: Village: 10387312 Feature: 14357617
[19:01:51 INFO]: Preparing start region for level 0 (Seed: -2823602682222914944)
[19:01:51 INFO]: Preparing start region for level 1 (Seed: -2823602682222914944)
[19:01:52 INFO]: Preparing start region for level 2 (Seed: -2823602682222914944)
[19:01:52 INFO]: [SetSpawn] Enabling SetSpawn v2.1
===[ SetSpawn v2.1 by artur9010 ]===
>Thanks for downloading SetSpawn!
http://dev.bukkit.org/bukkit-plugins/setspawn
=====
[19:01:52 INFO]: [RaspberryJuice] Enabling RaspberryJuice v1.7
[19:01:52 INFO]: [RaspberryJuice] ThreadListener Started
[19:01:52 INFO]: Server permissions file permissions.yml is empty, ignoring it
[19:01:52 INFO]: Done (<1.336s>)? For help, type "help" or "?"
>
```

Bild 1.4: Der Spigot-Server ist bereit.

4. Öffne Minecraft, und klicke auf **Mehrspieler**.
5. Klicke auf die Schaltfläche **Server hinzufügen**.
6. Im Feld **Server Name** nennst du deinen Server **Minecraft Python World**, in das Feld **Server Address** gibst du **localhost** ein (siehe Bild 1.5). Abschließend klickst du auf **Done**.



Bild 1.5: Servereinrichtung

7. Doppelklicke auf **Minecraft Python World**. Nun öffnet sich die von Spigot generierte Welt.

Wir wollen uns die neue Minecraft-Welt auf dem Spigot-Server einmal kurz ansehen. Die Welt läuft im Kreativmodus, d. h., du kannst umherfliegen. Drücke zweimal kurz hintereinander auf die Leertaste, um zu fliegen. Wenn du länger auf die Leertaste drückst, fliegst du höher, während das Drücken der **Umschalttaste** die Flughöhe verringert. Wenn du nicht mehr weiterfliegen möchtest, drücke zweimal kurz hintereinander auf die Leertaste.

Der Einstieg in eine neue Welt

Das Erstellen einer funkelneuen Minecraft-Welt mit einem Server unterscheidet sich ein wenig von dem Erstellen einer neuen Welt im Singleplayer-Modus. Gehe wie folgt vor, um eine neue Welt anzulegen:

1. Schließe zunächst das schwarze Fenster, das sich beim Starten des Servers geöffnet hat. Navigiere dann zum Ordner *Minecraft Python*. Klicke mit der rechten Maustaste auf den Ordner *Minecraft Tools*, und klicke dann auf **Kopieren**.
2. Klicke nun mit der rechten Maustaste an beliebiger Stelle in den Ordner *Minecraft Python*, und klicke dann auf **Einfügen**. Hierdurch entsteht eine Kopie des Ordners *Minecraft Tools* mit dem Namen *Minecraft Tools - Kopie*.
3. Jetzt klickst du mit der rechten Maustaste auf den Ordner *Minecraft Tools - Kopie* und klickst dann auf **Umbenennen**. Ich habe den Ordner *New World* genannt, aber du kannst ihm einen beliebigen Namen geben.
4. Öffne nun den Ordner *New World* (oder wie auch immer du ihn genannt hast) und darin dann den Ordner *server*.
5. Im Ordner *server* wählst du die Ordner *world*, *world_nether* und *world_the_end* aus (siehe Bild 1.6). Drücke die Taste **Entf**, um diese Ordner zu löschen.

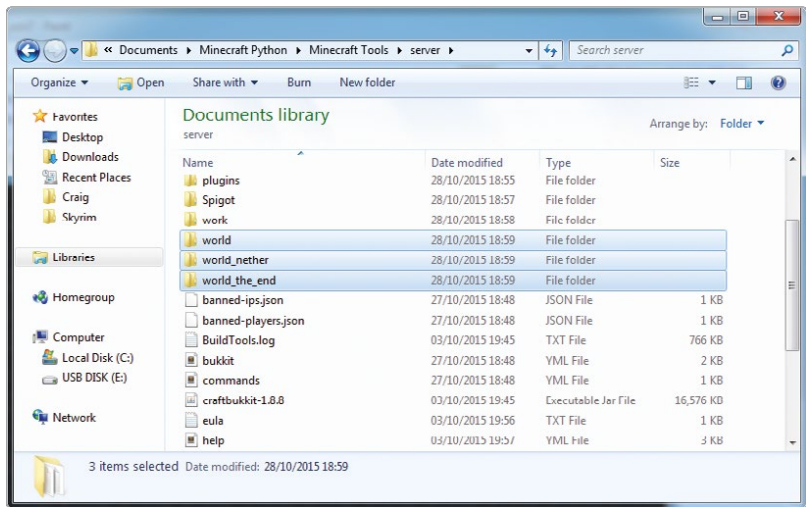


Bild 1.6: Die zu löschenden Ordner sind hervorgehoben dargestellt.

6. Doppelklicke nun im selben Ordner *server* auf die Datei *start*. (Achtung: Du musst auf die Datei *start* im Ordner *server* klicken, nicht auf die oben erwähnte Datei *Start_Server*!) Nun wird der Server neu gestartet, und es wird eine neue Welt erstellt.
7. Wenn du Minecraft und darin die Minecraft Python World jetzt öffnest, wird dir eine neu erstellte Welt angezeigt.

Du kannst diesen Vorgang zum Erstellen einer neuen Welt beliebig oft wiederholen. Zum Öffnen der ursprünglichen Welt doppelklickst du dagegen auf die Datei *Start_Server* im Ordner *Minecraft Tools*.

Du kannst eine Welt löschen und durch eine neue ersetzen, indem du die Ordner *world*, *world_nether* und *world_the_end* im Ordner der Welt löschst, die du ersetzen willst.

Offline spielen

Wenn du nicht über eine Internetverbindung verfügst, erhältst du bei dem Versuch, aus deinem Minecraft-Spiel heraus eine Verbindung mit dem Minecraft-Server herzustellen, eine Fehlermeldung. Das Problem kannst du beheben, indem du die Servereigenschaften änderst. Zunächst musst du ggf. das Serverfenster schließen. Danach navigierst du zum Ordner *Minecraft Python* und öffnest darin den Ordner *Minecraft Tools* und in diesem wiederum den Ordner *server* (du solltest die Option **Anzeige**

der Dateinamenserweiterung im Explorer aktiviert haben, damit du auch **server.properties** und nicht nur **server** im Explorer siehst). Dann öffnest du die Datei *server.properties* in einem Texteditor (wie etwa dem Windows-Editor) und setzt den Parameter `online-mode` (Bild 1.7) von `true` auf `false`. Speichere die Änderungen, und navigiere zurück zum Ordner *Minecraft Tools*. Dort doppelklickst du auf *Start_Server*, um den Server neu zu starten. Jetzt kannst du auch offline spielen.

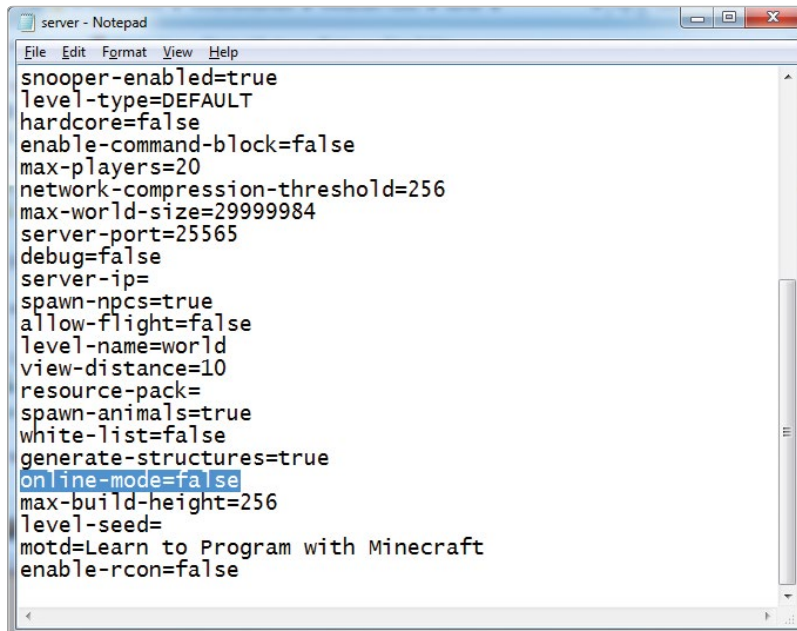


Bild 1.7: Stelle den markierten Parameter von `true` auf `false` um.

In den Überlebensmodus umschalten

Ich habe als Standardmodus für deinen Minecraft-Server den Kreativmodus festgelegt. So kannst du selbst geschriebene Python-Programme einfacher ausführen, denn du musst nicht darauf achten, ob der Spieler Gesundheitspunkte verliert, Hunger bekommt oder angegriffen wird.

Du kannst allerdings einige Programme spaßeshalber im Überlebensmodus ausprobieren. Das Umschalten zwischen Kreativ- und Überlebensmodus ist eigentlich ganz einfach.

Führe die folgenden Schritte aus, um den Server vom Kreativ- in den Überlebensmodus umzuschalten:

1. Öffne den Ordner *Minecraft Tools*. Öffne darin den Ordner *server*.
2. Suche nach der Datei *server.properties*, und öffne diese in einem Texteditor (z. B. dem Windows-Editor).
3. Suche in der Datei nach der Zeile *gamemode=1*, und ersetze diese durch *gamemode=0* (Bild 1.8).

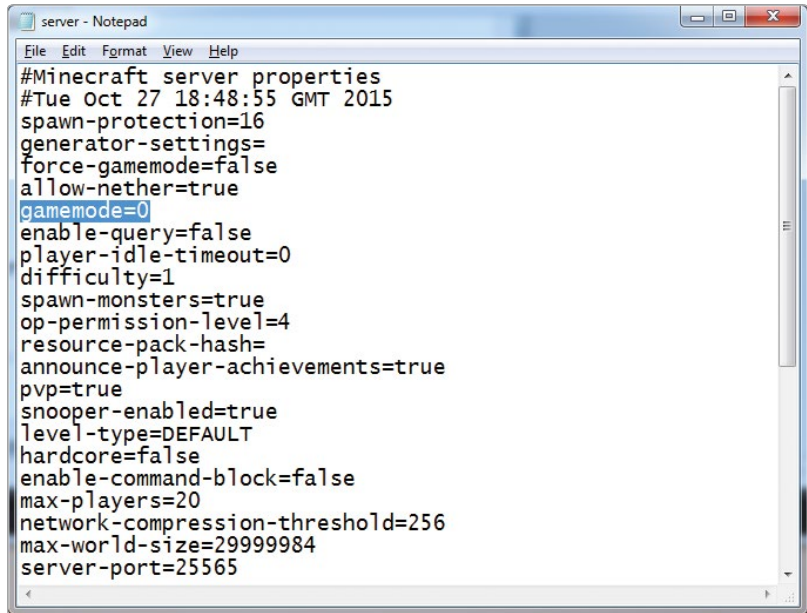


Bild 1.8: Aktiviere den Überlebensmodus durch Umstellen von *gamemode* auf 0.

4. Speichere und schließe die Datei.
5. Starte den Server durch einen Doppelklick auf die Datei *Start_Server* im Ordner *Minecraft Tools*. Wenn du nun am Spiel **Minecraft Python World** teilnimmst, läuft dieses im Überlebensmodus.

Du kannst auch jederzeit auf den Kreativmodus zurückschalten. Gehe so vor, wie ich gerade beschrieben habe; nur musst du statt *gamemode=0* in Schritt 3 *gamemode=1* in die Datei *server.properties* schreiben und natürlich den Server neu starten.

Die Einrichtung deines PCs ist damit abgeschlossen. Als Nächstes wirst du IDLE kennenlernen – das ist unsere Umgebung für das Schreiben von Code. Lies weiter unter »IDLE kennenlernen« auf Seite 26.

Mac einrichten

Damit du unter OS X Minecraft über Python steuern kannst, musst du fünf Komponenten installieren:

- Minecraft
- Python 3
- Java Development Kit
- Minecraft-Python-API
- Spigot Minecraft Server

In diesem Abschnitt beschreibe ich die Installation dieser Komponenten auf deinem Mac. Mit Minecraft fangen wir an.

Minecraft installieren

Solltest du Minecraft bereits besitzen und die aktuelle Version auf deinem Mac installiert haben, dann fahre fort bei »Python installieren« auf Seite 17. Weißt du nicht genau, ob sich die aktuelle Minecraft-Version auf deinem Rechner befindet, dann gehe wie hier beschrieben vor, um sie zu installieren.

Hast du das Spiel noch gar nicht, dann kannst du es auf der offiziellen Minecraft-Website <https://minecraft.net/> kaufen. Bitte ggf. einen Erwachsenen, dir dabei zu helfen. Notiere dir außerdem den beim Kauf von Minecraft verwendeten Benutzernamen und das Passwort, denn diese Angaben benötigst du später für die Anmeldung.

Wenn du Minecraft gekauft hast, gehst du wie folgt vor, um es auf deinem Mac zu installieren:

1. Rufe in deinem Browser die Seite <https://minecraft.net/download> auf.
2. Klicke im Abschnitt **Minecraft for Mac OS X** auf den Link **Minecraft.dmg**, um die gleichnamige Datei herunterzuladen. (Wird der Bereich **Minecraft for Mac OS X** nicht angezeigt, dann klicke auf **Show All Platforms**.)
3. Warte, bis die Datei vollständig heruntergeladen ist (ich habe derweil einen kurzen Blick aus dem Fenster geworfen), und öffne sie dann. Wenn sich das Fenster öffnet, ziehe das Minecraft-Symbol auf den Ordner *Programme* (Bild 1.9).

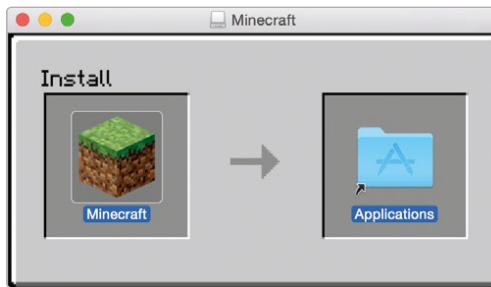


Bild 1.9: Ziehe das Minecraft-Symbol zur Installation auf den Ordner Programme.

Nun sollte Minecraft auf deinem System installiert sein.

Und jetzt? Jetzt wollen wir natürlich Minecraft spielen. Nimm dir vorher noch ein paar Minuten Zeit, um die Einrichtung durchzuführen. (Es kann sein, dass du noch Java installieren musst. Mache dann zunächst einfach ohne den ersten Spieletest weiter und installiere Python und dann Java. Danach steht einem Spiel nichts mehr im Wege.)

1. Zum Öffnen von Minecraft klickst du auf das Finder-Symbol im Dock, um den Dateibrowser zu öffnen.
2. Klicke in der Seitenleiste auf **Programme**.
3. Suche Minecraft im Ordner *Programme* (Bild 1.10). Doppelklicke darauf, und wähle **Öffnen** aus.
4. Unter Umständen wirst du dann gefragt, ob du Minecraft öffnen möchtest, wo es doch aus dem Internet heruntergeladen wurde. Klicke auf **Öffnen**.
5. Jetzt wird Minecraft gestartet und installiert möglicherweise erst einmal Updates.
6. Daraufhin öffnet sich das Anmeldefenster. Gib den Benutzernamen und das Passwort ein, die du beim Kauf von Minecraft verwendet hast, und klicke auf **Log In**.

Für die Beispiele in diesem Buch musst du mit der Minecraft-Version 1.9 arbeiten (ansonsten kannst du natürlich die jeweils aktuellste Version verwenden).

7. Klicke dazu im Launcher unten links auf die Schaltfläche **Edit Profile**.
8. Im Bereich **Version Selection** wird dir im Listenfeld **Use version** angezeigt, dass die aktuellste Version verwendet wird. Wähle hier die **release 1.9** aus und klicke dann rechts unten auf die Schaltfläche **Save Profile**.

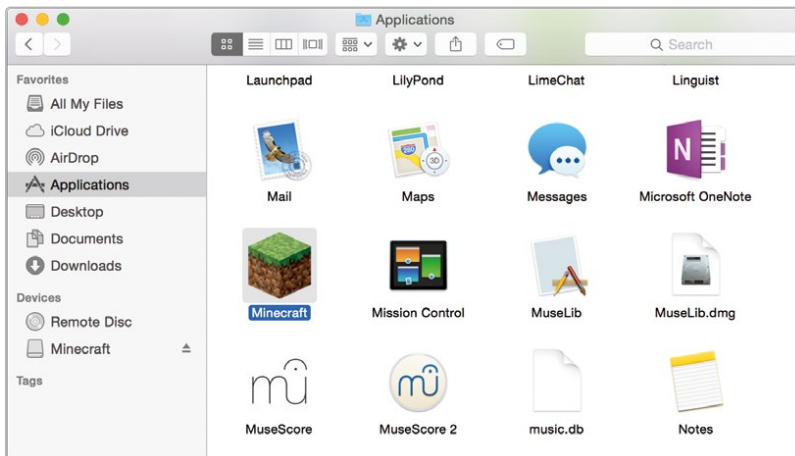


Bild 1.10: Minecraft im Ordner Programme

9. Klicke nun auf **Play**. Bevor es richtig losgeht, wird Minecraft noch einige weitere Updates herunterladen.
10. Abschließend klickst du auf **Single Player ▶ Create New World**. Gib deiner Welt einen Namen deiner Wahl, und klicke dann auf **Create New World**. Nun wird die Welt erstellt und du kannst nach Herzenslust darin spielen.

Viel Spaß dabei! Wenn du noch nie Minecraft gespielt hast, probiere erst einmal ein bisschen herum, bis es in deiner Minecraft-Welt dunkel wird. Hüte dich vor den Monstern! Wenn du Minecraft später mit Python nutzt, wirst du eine Multiplayer-Welt verwenden, die anders sein wird als diese Welt. Wir kommen in »Spigot ausführen und ein Spiel erstellen« auf Seite 21 darauf zurück.

Doch nun wieder an die Arbeit. Jetzt wollen wir Python installieren. Um den Mauszeiger wieder aus Minecraft herauszuholen, drückst du einfach **Esc** auf deiner Tastatur. Schließe Minecraft, bevor du mit der Installation fortfährst.

Python installieren

Python ist die Programmiersprache, die du in diesem Buch erlernen wirst. Und die wollen wir jetzt installieren.

1. Rufe in deinem Browser die Seite <https://www.python.org/downloads/mac-osx/> auf.

2. Klicke auf den Link **Latest Python 3 Release – Python 3.5.0**. (Dies ist zu dem Zeitpunkt, wo ich dies schreibe, die aktuelle Version, womöglich wird dir aber schon eine höhere Versionsnummer angezeigt. Installiere in jedem Fall die aktuellste Version.) Nun wird der Download von Python gestartet.
3. Wenn du die Datei heruntergeladen hast, doppelklicke darauf.
4. Nun öffnet sich der Installer. Klicke dreimal auf **Continue**. Du wirst aufgefordert, den Bedingungen des Softwarelizenzvertrags zuzustimmen. Klicke auf **Agree**.
5. Klicke auf **Install**, und warte dann, bis die Installation von Python abgeschlossen ist. Ich habe die Zeit genutzt, um einen Blick auf die Wettervorhersage zu werfen.
6. Klicke auf **Schließen**. Python ist nun installiert.

Java installieren

Nach der erfolgreichen Installation von Minecraft und Python musst du einige Einstellungen vornehmen, damit die beiden Programme miteinander kommunizieren können. Zu diesem Zweck verwendest du ein Programm namens Spigot, das aber erst funktioniert, wenn du das aktuelle Java Development Kit (JDK) auf deinem Computer installiert hast. Und genau deswegen werden wir das jetzt tun.

1. Rufe in deinem Browser die Seite <http://www.oracle.com/technetwork/java/javase/downloads/index.html> auf, und klicke auf den Button **Java Download**.
2. Wähle **Accept License Agreement** aus, und klicke dann auf **Mac OS X x64**.
3. Wenn du die Datei heruntergeladen hast, doppelklicke darauf.
4. Nun öffnet sich der Installer. Doppelklicke auf das Symbol **Installieren**.
5. Gib dein Passwort auf Anforderung ein.
6. Warte, bis Java installiert ist. Klicke auf **Schließen**, um den Vorgang abzuschließen.

Nun wollen wir sehen, ob das JDK ordnungsgemäß installiert wurde.

1. Klicke auf **Systemeinstellungen**.
 2. Das Java-Symbol sollte in den Systemeinstellungen angezeigt werden (Bild 1.11).
-

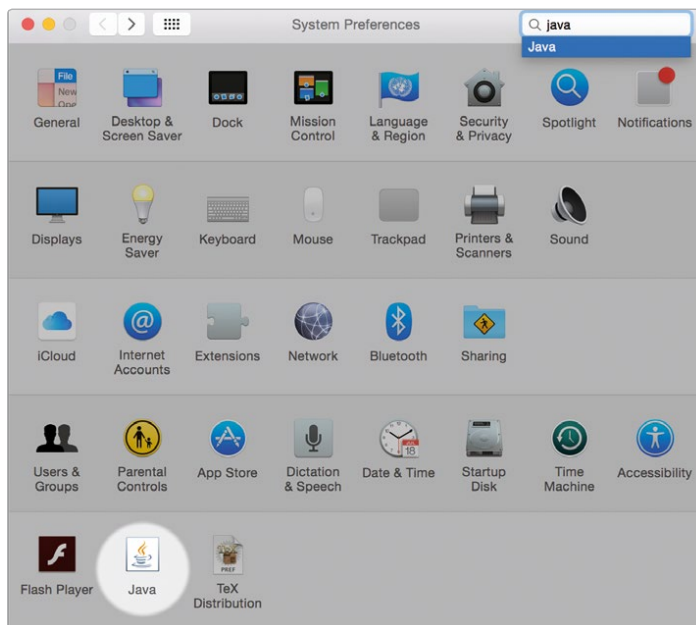


Bild 1.11: Java ist installiert!

Das ist alles! Java ist eingerichtet und kann zur Ausführung des Minecraft-Servers verwendet werden. Diesem wollen wir uns nun zuwenden.

Minecraft-Python-API und Spigot installieren

Nächster Schritt: Wir müssen die Minecraft-Python-API und den Minecraft-Server auf deinem Computer installieren.

»API« ist die Abkürzung für *Application Programming Interface* (dt. Anwendungsprogrammierschnittstelle). Über eine solche Schnittstelle können Programme mit Anwendungen kommunizieren, die von anderen Leuten geschrieben wurden. In diesem Fall erlaubt es die Minecraft-Python-API, dass deine in Python geschriebenen Programme mit Minecraft kommunizieren können. Du könntest beispielsweise ein Python-Programm verfassen, das Minecraft über diese API anweist, einen Block im Spiel zu erstellen oder die Position des Spielers zu ändern.

Im normalen Singleplayer-Modus von Minecraft werden APIs nicht unterstützt, aber deine Programme werden mit einem Minecraft-Server interagieren, der wieder mit APIs etwas anfangen kann. Minecraft-Ser-

ver werden in der Regel online genutzt, damit viele Leute gemeinsam in derselben Welt spielen können. Du kannst den Server aber auch auf deinem eigenen Computer laufen lassen und allein spielen. Multiplayer- und Singleplayer-Server gestatten die Verwendung einer API mit Minecraft. In diesem Buch werden wir einen Singleplayer-Server namens Spigot auf deinem Computer installieren.

Jetzt weißt du, wofür eine API und ein Server gut sind. Installieren wir sie also jetzt auf deinem Computer. Ich habe dafür ein praktisches Download-Paket zusammengestellt – damit gelingt die Installation im Handumdrehen. Führe einfach die folgenden Schritte aus:

1. Rufe in deinem Browser die Seite <https://www.nostarch.com/pythonwithminecraft/> auf, und lade die Datei *MinecraftTools Mac.zip* herunter.
 2. Wenn der Download der Datei abgeschlossen ist, öffne den Ordner *Downloads*, und klicke auf **Im Finder zeigen**.
 3. Im Finder klickst du bei gedrückter **control**-Taste auf die Datei und wählst **MinecraftTools Mac.zip kopieren** aus.
 4. Navigiere zum Ordner *Dokumente*. Wähle über die Menüleiste **Ablage ▶ Neuer Ordner** aus. Nenne diesen neuen Ordner *MinecraftPython*. Achte dabei darauf, dass der Ordnername kein Leerzeichen enthält.
 5. Öffne den Ordner *MinecraftPython*. Klicke bei gedrückter **control**-Taste in den Ordner, und wähle **Objekt einsetzen** aus. Die Datei *MinecraftTools Mac.zip* wird nun hierher kopiert.
 6. Klicke bei gedrückter **control**-Taste und wähle **Öffnen mit ▶ Archivierungsprogramm**. Nach dem Öffnen der ZIP-Datei wird ein neuer Ordner namens *MinecraftTools* erstellt.
 7. Öffne den Ordner *MinecraftTools*. Bild 1.12 zeigt die Dateien, die im Ordner enthalten sind.
 8. Klicke mit gedrückter **control**-Taste auf die Datei *Install_API.command*, und wähle dann **Öffnen** aus. Nun wird ein neues Fenster geöffnet. Gib dein Passwort ein, um die Minecraft-Python-API zu installieren.
-

Hinweis

Wenn eine Fehlermeldung angezeigt wird, laut der *Install_API.com* nicht geöffnet werden kann, weil sie von einem nicht verifizierten Entwickler stammt, klicke auf **Systemeinstellungen** und dann auf **Sicherheit**. Hier siehst du die Meldung »Install_API.com wurde nicht geöffnet, weil es von einem nicht verifizierten Entwickler stammt.« Klicke auf **Dennoch öffnen**. Nun sollte sich das Fenster wie vorgesehen öffnen.

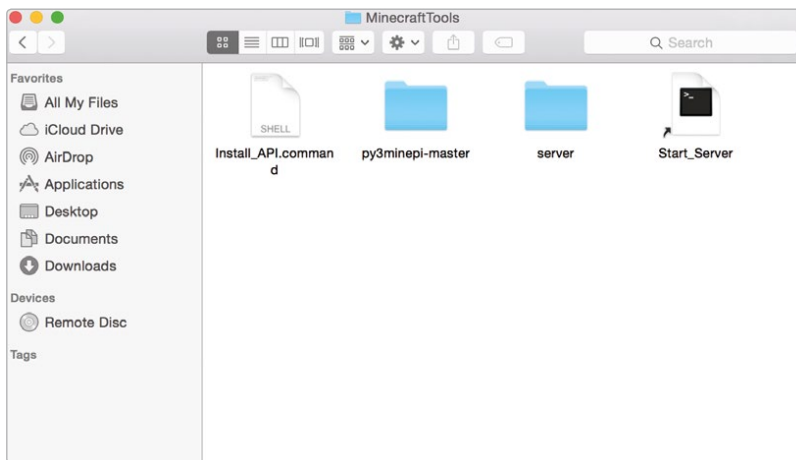


Bild 1.12: Inhalt des Ordners MinecraftTools

9. Wenn die Installation abgeschlossen ist, schließe das Fenster.

Die Minecraft-Python-API und der Minecraft-Server sind nun installiert. Der abschließende Schritt besteht darin, den Server zu starten. Das tun wir im nächsten Abschnitt.

Spigot ausführen und ein Spiel erstellen

Bei der ersten Ausführung erstellt Spigot eine Minecraft-Welt. Zum Starten von Spigot führst du die folgenden Schritte aus:

1. Navigiere zu deinem Ordner *MinecraftPython*, und öffne den Unterordner *MinecraftTools*.

2. Klicke im Ordner *MinecraftTools* bei gedrückter **control**-Taste auf die Datei *Start_Server*, und wähle **Öffnen** aus. Wenn eine Fehlermeldung angezeigt wird, wechsle zu den **Systemeinstellungen** und dann zu **Sicherheit**, und klicke auf **Dennoch öffnen**.
3. Spigot startet nun den Minecraft-Server. Dabei wird ein Fenster geöffnet, in dem eine Menge Text angezeigt wird, während Spigot das Spiel für dich generiert. Wenn der Vorgang beendet ist, darfst du dieses Fenster nicht schließen!
4. Öffne Minecraft, und klicke auf **Mehrspieler**.
5. Klicke auf die Schaltfläche **Add Server**.
6. Im Feld **Server Name** nennst du deinen Server **Minecraft Python World**, in das Feld **Server Address** gibst du **localhost** ein (siehe Bild 1.13). Abschließend klickst du auf **Done**.
7. Doppelklicke auf **Minecraft Python World**. Nun öffnet sich die von Spigot generierte Welt.

Wir wollen uns die neue Minecraft-Welt auf dem Spigot-Server einmal kurz ansehen. Die Welt läuft im Kreativmodus, d. h., du kannst umherfliegen. Drücke zweimal kurz hintereinander auf die Leertaste, um zu fliegen. Wenn du länger auf die Leertaste drückst, fliegst du höher, während das Drücken der **Umschalttaste** die Flughöhe verringert. Wenn du nicht mehr weiterfliegen möchtest, drücke zweimal kurz hintereinander auf die Leertaste.

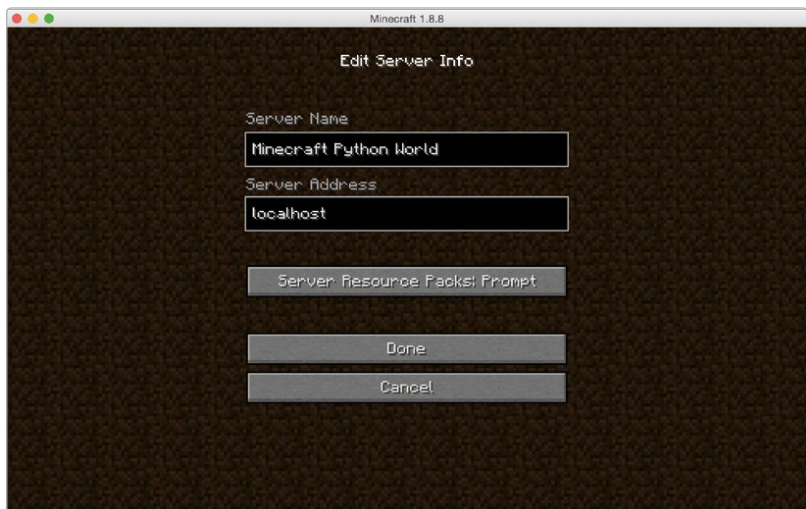


Bild 1.13: Füge den Server hinzu, sodass du in Zukunft schnell und einfach darauf zugreifen kannst.

Der Einstieg in eine neue Welt

Das Erstellen einer funkelnagelneuen Minecraft-Welt mit einem Server unterscheidet sich ein wenig von dem Erstellen einer neuen Welt im Singleplayer-Modus. Gehe wie folgt vor, um eine neue Welt anzulegen:

1. Schließe zunächst das schwarze Fenster, das sich beim Starten des Servers geöffnet hat. Klicke dann den Ordner *MinecraftTools* einmal an und drücke **command-C**.
1. Drücke dann **command-V**. Hierdurch entsteht eine Kopie des Ordners *MinecraftTools* mit dem Namen *Minecraft Tools - Kopie*.
2. Klicke bei gedrückter **control-Taste** auf den Ordner *MinecraftTools - Kopie*, und klicke dann auf **Umbenennen**. Ich habe den Ordner *New World* genannt, aber du kannst ihm einen beliebigen Namen geben.
3. Öffne nun den Ordner *New World* (oder wie auch immer du ihn genannt hast) und darin dann den Ordner *server*.
4. Im Ordner *server* wählst du die Ordner *world*, *world_nether* und *world_the_end* aus. Drücke zum Löschen dieser Ordner **Umschalttaste+Löschentaste**.
5. Navigiere zurück zum Ordner *New World*, und klicke auf die Datei *Start_Server*. Nun wird der Server neu gestartet, und es wird eine neue Welt erstellt.
6. Wenn du nun Minecraft und darin die **Minecraft Python World** öffnest, wird dir eine neu erstellte Welt angezeigt.

Du kannst diesen Vorgang zum Erstellen einer neuen Welt beliebig oft wiederholen. Zum Öffnen der ursprünglichen Welt doppelklickst du dagegen auf die Datei *Start_Server* im Ordner *MinecraftTools* (und nicht im Ordner *New World*).

Du kannst eine Welt löschen und durch eine neue ersetzen, indem du die Ordner *world*, *world_nether* und *world_the_end* im Ordner der Welt löschst, die du ersetzen möchtest.

Offline spielen

Wenn du nicht über eine Internetverbindung verfügst, erhältst du bei dem Versuch, aus deinem Minecraft-Spiel heraus eine Verbindung mit dem Minecraft-Server herzustellen, eine Fehlermeldung. Das Problem kannst du beheben, indem du die Servereigenschaften änderst. Zunächst musst du ggf. das Serverfenster schließen. Danach navigierst du zum Ordner *MinecraftPython* und öffnest darin den Ordner *MinecraftTools* und

in diesem wiederum den Ordner *server*. Dann öffnest du die Datei *server.properties* in einem Texteditor wie TextEdit und setzt den Parameter *online-mode* von *true* auf *false* (siehe Bild 1.7 auf Seite 13). Speichere die Änderungen, und navigiere zurück zum Ordner *MinecraftTools*. Dort doppelklickst du auf *Start_Server*, um den Server neu zu starten. Jetzt kannst du auch offline spielen.

In den Überlebensmodus umschalten

Ich habe als Standardmodus für deinen Minecraft-Server den Kreativmodus festgelegt. So kannst du selbst geschriebene Python-Programme einfacher ausführen, denn du musst nicht darauf achten, ob der Spieler Gesundheitspunkte verliert, Hunger bekommt oder angegriffen wird.

Du kannst allerdings einige Programme spaßeshalber im Überlebensmodus ausprobieren. Das Umschalten zwischen Kreativ- und Überlebensmodus ist eigentlich ganz einfach.

Führe die folgenden Schritte aus, um den Server vom Kreativ- in den Überlebensmodus umzuschalten:

1. Öffne den Ordner *MinecraftTools*. Öffne darin den Ordner *server*.
2. Suche nach der Datei *server.properties*, und öffne sie in einem Texteditor (z. B. TextEdit).
3. Suche in der Datei nach der Zeile *gamemode=1*, und ersetze sie durch *gamemode=0* (siehe Bild 1.8 auf Seite 14).
4. Speichere und schließe die Datei.
5. Starte den Server durch einen Doppelklick auf die Datei *Start_Server* im Ordner *MinecraftTools*. Wenn du nun am Spiel **Minecraft Python World** teilnimmst, läuft dieses im Überlebensmodus.

Du kannst auch jederzeit auf den Kreativmodus zurückschalten. Gehe so vor, wie ich gerade beschrieben habe; nur musst du statt *gamemode=0* in Schritt 3 *gamemode=1* in die Datei *server.properties* schreiben.

Die Einrichtung deines Mac ist damit abgeschlossen. Als Nächstes wirst du IDLE kennenlernen – das ist unsere Umgebung für das Schreiben von Code. Lies weiter unter »IDLE kennenlernen« auf Seite 26.

Raspberry Pi einrichten

Melde dich bei deinem Raspberry Pi an, und starte den Desktop mit dem Befehl `startx`. (Falls du die aktuelle Version des Raspberry-Pi-Betriebssystems verwendest, brauchst du den Befehl nicht einzugeben.)

Je nach Version deines Raspberry Pi sind auf ihm bereits zwei oder drei Python-Versionen installiert. Für die Zwecke dieses Buchs musst du die aktuellste Python-Version 3 verwenden.

Standardmäßig ist auf dem Raspberry-Pi-Computer eine abgespeckte Minecraft-Version namens »Minecraft: Pi Edition« installiert. Insofern findest du auf dem Pi bereits alles vor, was du für die Programmierung deiner Minecraft-Welt mit Python benötigst. Falls du mit dem Raspberry Pi noch keine Erfahrung hast, findest du auf der offiziellen Website <http://www.raspberrypi.org> eine Reihe von Anleitungen für die ersten Schritte.

Verwendest du ein älteres (d. h. vor August 2014 entstandenes) Image auf der SD-Karte, dann wirst du unter Umständen feststellen, dass Minecraft gar nicht installiert ist. Dies lässt sich aber bei Bedarf schnell und unkompliziert nachholen. Zunächst musst du mit deinem Raspberry Pi eine Verbindung mit dem Internet herstellen. Einen Leitfaden zum Verbinden des Raspberry Pi mit dem Internet findest du unter <http://www.raspberrypi.org>.

Nachdem du erfolgreich eine Verbindung hergestellt hast, führst du die folgenden Schritte aus:

1. Doppelklicke auf dem Desktop auf **LXTerminal**.
2. Nach dem Öffnen von LXTerminal gibst du folgenden Befehl ein:

```
$ sudo apt-get update
```

3. Wenn das Update abgeschlossen ist, fährst du mit dem folgenden Befehl fort:

```
$ sudo apt-get install minecraft-pi
```

4. Warte nun ab, bis die Installation abgeschlossen ist. Minecraft ist dann installiert.

Die Raspberry-Pi-Version von Minecraft weist im Vergleich zur Desktopversion einige Einschränkungen auf. Die Spielwelt ist deutlich kleiner, und eine Menge Blöcke und andere Funktionen (wie etwa der Überlebensmodus) sind nicht vorhanden. Trotzdem lassen sich die fantasti-

schen Programme, die du in diesem Buch kennenlernen wirst, samt und sonders auch auf dieser Plattform ausführen.

Bevor wir fortfahren, wollen wir einen Ordner erstellen, in dem du deine Python-Programme speichern kannst. Klicke auf der Taskleiste auf das Dateibrowsersymbol. Öffne den Ordner *Documents*, und klicke dann mit der rechten Maustaste auf den Hintergrund des Dateibrowsers. Aus dem Menü wählst du nun **Create New... ▶ Folder** aus. Nenne den Ordner *Minecraft Python*, und klicke dann auf **OK**.

Hinweis

Wenn du einen Original-Raspberry-Pi einsetzt, wirst du feststellen, dass die Programme aus diesem Buch aufgrund der Beschränkungen des Raspberry Pi relativ langsam laufen. Solche Geschwindigkeitsprobleme sind beim Raspberry Pi 2 deutlich seltener zu beobachten.

Klicke zum Öffnen von Minecraft auf das Startmenü oben links auf dem Desktop. (Bei der älteren Version des Raspberry-Pi-Betriebssystems befindet sich das Startmenü unten links.) Wechsle zu **Games**, und klicke auf **Minecraft**. Minecraft wird nun geöffnet. Wenn du Minecraft zum ersten Mal startest, musst du nun auf **Create World** klicken.

Grundsätzlich solltest du die Fenstergröße nicht ändern, da dies Probleme verursachen kann.

Manchmal werden Fenster und Dialoge, die sich öffnen (z. B. zur Bestätigung, dass du eine Datei in Python speichern möchtest), durch das Minecraft-Fenster verdeckt. In diesem Fall musst du, um andere Fenster zu benutzen, lediglich Minecraft minimieren. Wenn Probleme auftreten, versuche nach der Installation von Minecraft zunächst, deinen Raspberry Pi neu zu starten.

IDLE kennenlernen

Nun haben wir eigentlich alles installiert und eingerichtet, was wir brauchen. Wenden wir uns also IDLE zu. IDLE ist die Software, mit der du deine Python-Programme entwickelst und ausführst. Zudem ist IDLE bereits

Bestandteil deiner Python-Installation, muss also nicht separat installiert werden. Öffnen wir nun IDLE.

- **Windows** Öffne das Startmenü, und gib **IDLE** in das Suchfeld ein.
- **Mac** Öffne den Ordner *Programme*, und klicke auf das IDLE-Symbol.
- **Raspberry Pi** Wähle unter **Menu ▶ Entwicklung** das IDLE Symbol, das mit Python 3 beschriftet ist.

Nun öffnet sich ein IDLE-Fenster (Bild 1.14). Dieses Fenster ist die so genannte *Python-Shell*. Diese Shell ist so genial, dass es mir schier die Schuhe ausgezogen hat, als ich das Programmieren mit Python erlernte!

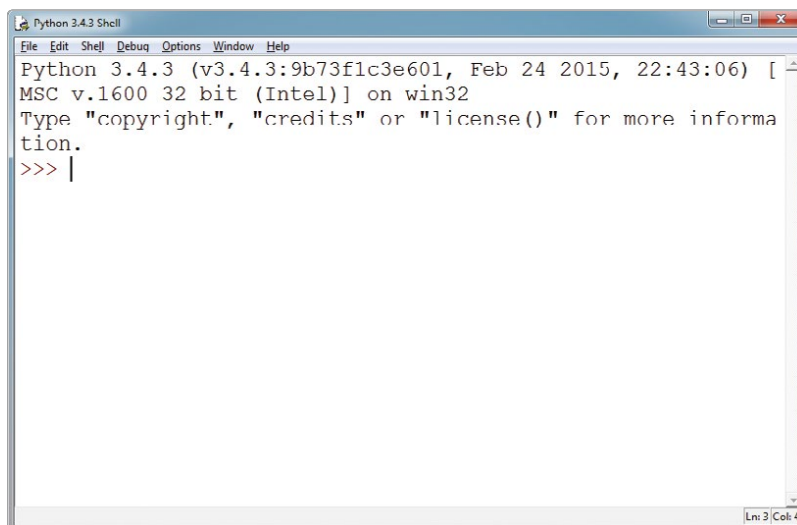


Bild 1.14: IDLE-Fenster zur Programmierung in Python

Python-Shell kennenlernen

In der Python-Shell schreibst du Programme, die dann zeilenweise ausgeführt werden. Du kannst eine Codezeile schreiben, diese direkt ausführen und sehen, was passiert, und dann die nächste Zeile schreiben. Dieses Verhalten ist super, denn du kannst ein bisschen herumprobieren und deinen Code ganz einfach testen.

Am Anfang der Zeile stehen in diesem Fenster normalerweise drei größer-als-Zeichen (>>>). Dies ist der *Befehlsprompt*. Durch Anzeigen des Befehlsprompts teilt die Python-Shell dir mit, dass sie zur Annahme

von Befehlen bereit ist. Beginnen wir mit einem ganz einfachen Befehl: Python soll zwei Zahlen addieren.

Klicke neben dem Prompt in die Python-Shell, und gib `2 + 2` ein. Achtung: Den Prompt selbst (`>>>`) musst du nicht eintippen. Das Ganze sollte dann so aussehen:

```
>>> 2 + 2
```

Nach Eingabe des Befehls drückst du die **Eingabetaste**. Die Python-Shell gibt nun das Ergebnis aus – in diesem Fall 4:

```
>>> 2 + 2
4
```

Du kannst in der Shell auch Texte verarbeiten. Gib den folgenden Code in die Python-Shell ein, und drücke die **Eingabetaste**:

```
>>> "W" + "o" * 5
Wooooo
```

Dieser Code gibt also das Wort Wooooo aus. Die Zahl am Ende des Befehls bestimmt, wie viele o im Wort vorhanden sein sollen. Indem du den Wert änderst, kannst du beeinflussen, wie lang das Wort wird. Nimm doch mal 20 oder irgendeine andere Zahl, die dir gerade einfällt:

```
>>> "W" + "o" * 20
Wooooooooooooooooooooo
```

Wooooooooooooooooooooo! Die Python-Shell kann eine Menge Spaß bereiten.

Du wirst vielleicht bemerkt haben, dass IDLE den Code unterschiedlich einfärbt. Dies bezeichnet man als *Syntaxhervorhebung*. Sie soll die Lesbarkeit des Codes verbessern. In diesem Buch verwenden wir die gleiche Farbgebung wie in IDLE, d. h., die Farben stimmen mit jenen überein, die du beim Programmieren sehen wirst.

Damit kommen wir zum Texteditor von IDLE.

Auftritt: Der IDLE-Texteditor

Wenn du längere Programme schreiben möchtest, ist die Shell ungeeignet. Und hier kommt der IDLE-Texteditor ins Spiel! Anders als die Shell führt der Editor nicht jede Codezeile unmittelbar nach der Eingabe aus, sondern dein ganzes Programm läuft ab, wenn du das festlegst.

Klicke in der Menüleiste von IDLE auf **File**, und wähle **New File** aus. Nun öffnet sich ein neues Fenster, das so ähnlich aussieht wie das Fenster in Bild 1.15. Das ist der Texteditor.

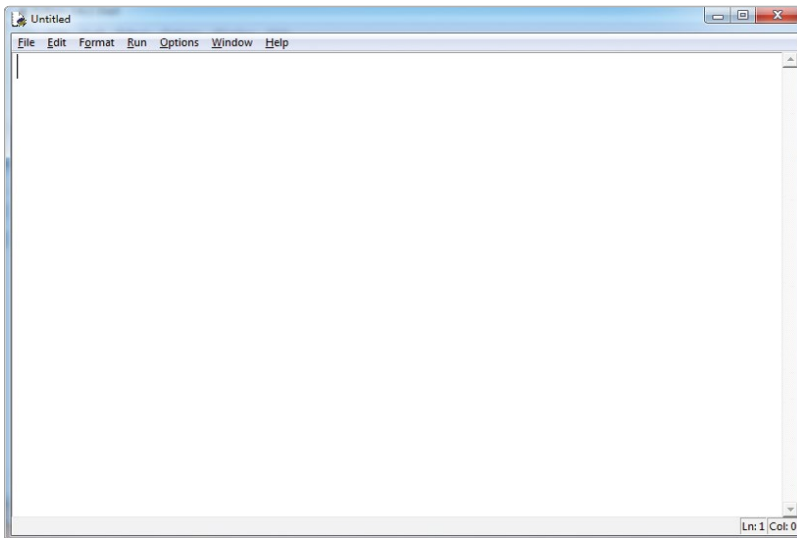


Bild 1.15: Der IDLE-Texteditor

Du denkst jetzt bestimmt: »Moment! Der Texteditor sieht doch genau so aus wie die IDLE-Python-Shell.« Das ist richtig. Und doch gibt es einen sehr wichtigen Unterschied: Das neue Fenster hat keinen Prompt (`>>>`) am Anfang jeder Zeile.

Wir wollen einmal sehen, was das bedeutet. Gib in die erste Zeile des Texteditors den folgenden Code ein, und drücke die **Eingabetaste**:

```
print(2 + 2)
```

Das Drücken der **Eingabetaste** hat eigentlich nichts bewirkt, außer dass eine neue Zeile eingefügt wurde. Hattest du etwas anderes erwartet? Da der Texteditor den Code beim Drücken der **Eingabetaste** nicht ausführt, kannst du vor dem Starten deines Programms beliebig viele Programm-

zeilen eingeben. Das wollen wir jetzt mal machen. Füge ein paar Zeilen hinzu, damit unsere Datei am Ende wie folgt aussieht:

```
print(2 + 2)
print("W" + "o" * 20)
print("PYTHON!")
print("<3s")
print("Minecraft")
```

Bevor du deinen Python-Code aus dem IDLE-Texteditor ausführst, musst du ihn erst einmal speichern. Hierzu klickst du auf **File > Save As**. Erstelle einen Unterordner in deinem Ordner *Minecraft Python*, und nenne diesen neuen Ordner *Setting Up*. Speichere dann das obige Programm als *pythonLovesMinecraft.py* im Ordner *Setting Up*.

Als Nächstes wollen wir das Programm ausführen. Klicke im Menü auf **Run > Run Module**. Das Shell-Fenster öffnet sich, und dein Programm wird darin ausgeführt. Die Ausgabe siehst du in Bild 1.16.

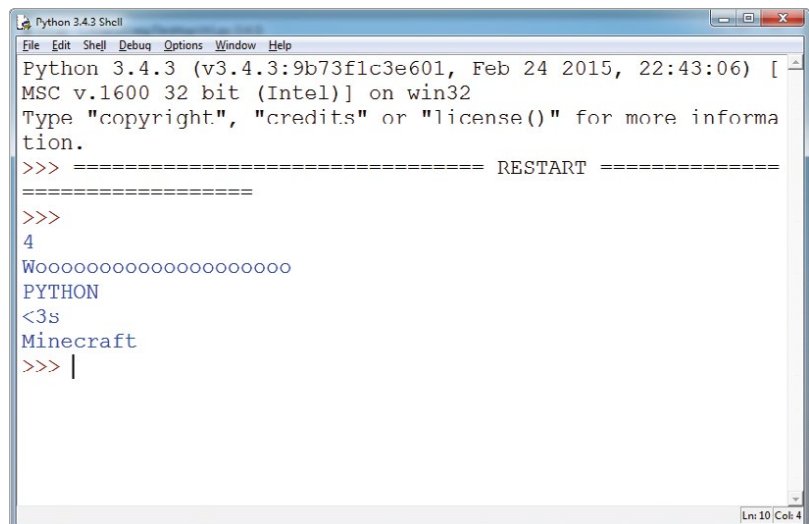


Bild 1.16: Ausgabe des Python-Programms

Anders als bei der Shell geben Befehle, die über den Texteditor ausgeführt werden, ihre Resultate nicht automatisch aus. Du musst dies vielmehr mit `print()` in deinem Code festlegen. Die Details kannst du an dieser Stelle ignorieren – du wirst alles Wichtige im Verlauf dieses Buchs lernen.

Wenn du Programme im IDLE-Texteditor startest, erfolgt die Programmausführung in der Shell, die sich dann öffnet. Du schreibst das Programm also in einem separaten Fenster, doch IDLE benutzt zur Ausführung deines Programms stets die Shell.

Shell oder Editor – wann verwendest du was?

Nachdem du nun den Unterschied zwischen der Python-Shell und dem Texteditor in IDLE kennst, fragst du dich vielleicht, wann das eine und wann das andere besser ist. Meine Faustregel lautet: Wenn ich nur ein paar Zeilen testen und diese nicht wiederverwenden möchte, nutze ich die Shell. Ich empfehle dir, die kurzen Beispiele, die im weiteren Verlauf dieses Buchs folgen werden, in der Python-Shell auszuprobieren.

Den Texteditor hingegen verwende ich für Programme mit vielen Codezeilen oder solchen, die ich wiederverwenden möchte. Alle Missionen in diesem Buch nutzen den Texteditor, damit du deine Fortschritte speichern kannst; möchtest du hingegen nur etwas ausprobieren, dann ist die Shell hierfür oft die bessere Variante.

Die in diesem Buch verwendeten Prompts

Wenn in diesem Buch ein Codeausschnitt in der Python-Shell geschrieben wurde, dann beginnt er mit dem bekannten Befehlsprompt `>>>`. Also so:

```
>>> print("Wooooo Minecraft")
```

Ich empfehle, die Lektüre an den betreffenden Stellen kurz zu unterbrechen und den Code nach IDLE zu kopieren, damit du dich damit vertraut machen kannst. Ausgaben der Shell stehen dann in der nachfolgenden Zeile:

```
>>> print("Wooooo Minecraft")
Wooooo Minecraft
```

Code, der im Texteditor geschrieben wird, beginnt *nicht* mit einem Prompt:

```
print("Adventures")
```

Die Codeausgabe wird auch nicht automatisch auf deinem Computer angezeigt. Um zu zeigen, wie die Ausgabe am Ende der Ausführung aussehen sollte, erläutere ich das oder zeige es in einem separaten Kästchen an. Der obige Code etwa sollte Folgendes ausgeben:

Adventures

Damit du den Erläuterungen des Codes in diesem Buch einfacher folgen kannst, verwende ich Markierungen, um die wesentlichen Aspekte hervorzuheben. Wenn du im Code eine solche Markierung siehst, dann gibt es dazu im Text auch eine passende Erklärung und umgekehrt. Die Markierungen sehen so aus:

❶ ❷ ❸ ❹ ❺ ❻

Minecraft-Python-Setup testen

Wir wollen nun überprüfen, ob du die gesamte Software ordnungsgemäß installiert hast. Zu diesem Zweck nutzen wir ein ganz einfaches Python-Programm, das mit Minecraft interagiert.

Der Reihe nach: Wenn du einen PC oder Mac verwendest, musst du drei Softwareprogramme starten. Führe dazu die folgenden Schritte aus:

1. Navigiere zum Ordner *Minecraft Tools*, und klicke auf *Start_Server*, um Spigot zu starten.
2. Öffne Minecraft, und stelle die Verbindung zum Spigot-Server her. Hierzu wählst du **Minecraft Python World** aus dem Multiplayer-Menü aus.
3. Drücke die Taste **Esc** auf deiner Tastatur, um den Mauszeiger aus dem Minecraft-Fenster zu lösen, und öffne eine Python-Shell in IDLE.

Wenn du Programme schreiben möchtest, die mit Minecraft interagieren, müssen diese drei Softwareanwendungen in jedem Fall geöffnet sein.

Bei Verwendung eines Raspberry Pi öffnest du IDLE und Minecraft.

Jetzt gibst du die folgende Zeile in deine Shell ein. Achte dabei darauf, die Groß-/Kleinschreibung exakt so zu verwenden wie hier gezeigt!

```
>>> from mcpi.minecraft import Minecraft
```

Drücke die **Eingabetaste**, um in die nächste Zeile der Shell zu wechseln. Nun gib folgende Zeile ein:

```
>>> mc = Minecraft.create()
```

Wenn jetzt eine Fehlermeldung erscheint, die der in Bild 1.17 ähnelt, dann hat irgendetwas nicht geklappt.

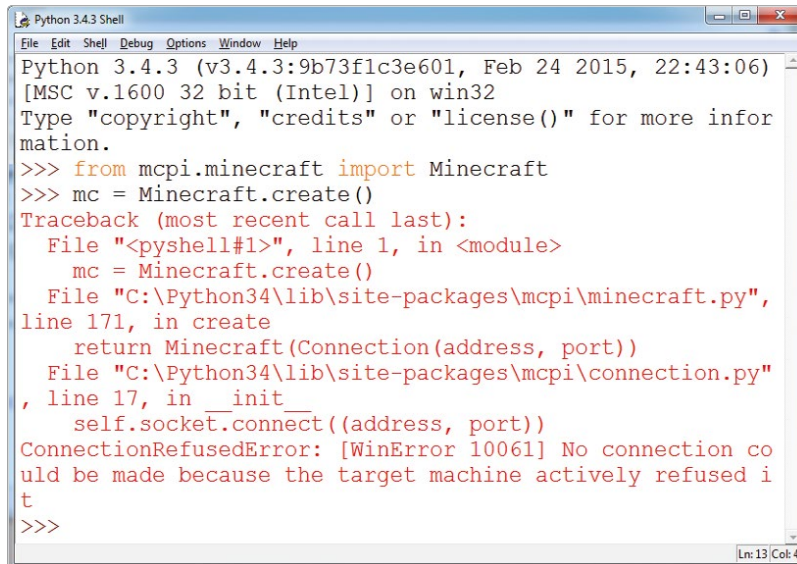


Bild 1.17: Diese Fehlermeldung besagt, dass Spigot nicht gestartet wurde.

Überprüfe Folgendes in der angegebenen Reihenfolge: Hast du Minecraft gestartet? Wird Spigot ausgeführt? Befindest du dich in der Multiplayer-Welt? Verwendest du die richtige Python-Version (nämlich Version 3, nicht 2)? Wenn der Fehler auftrat, nachdem du die erste Zeile eingegeben hast, weist dies darauf hin, dass du die API nicht richtig installiert hast. Führe die Schritte zur Installation der API erneut aus. Trat der Fehler nach Eingabe der zweiten Zeile auf, dann wurden Java oder Spigot unter Umständen nicht richtig installiert. Installiere diese beiden Komponenten nacheinander neu.

Sollte die Fehlermeldung `ImportError: No module named 'mcpi'` angezeigt werden, verwendest du offenbar eine veraltete Python-Version. Sorge dafür, dass stets die aktuelle Version installiert ist.

Wird keine Fehlermeldung angezeigt, dann füge folgende Zeile zum Programm in IDLE hinzu:

```
mc.player.setTilePos(0, 120, 0)
```

Dadurch fliegt der Spieler hoch in die Luft! Mit diesem Code wird er nämlich an eine neue Position teleportiert. Mehr dazu erfährst du in Kapitel 2. Bitte umblättern und weiterlesen!

Kapitel 2

Teleportation mit Variablen

Wir kommen nun zu dem, was wir uns eigentlich vorgenommen haben: Minecraft-Welten mit Python zu kontrollieren. Bist du bereit? In diesem Kapitel werden wir einen kleinen Rundgang durch die Grundlagen von Python machen. Danach wirst du deine neuen Fähigkeiten auf die Probe stellen und deine eigene Teleportationstour durch deine Minecraft-Welt erstellen.

Die in diesem Kapitel beschriebenen Konzepte sind nicht spezifisch für Minecraft-Python, können also in jedem Python-Programm verwendet werden, das du schreibst.

Was ist ein Programm?

Ein *Programm* ist eine Abfolge von Anweisungen, die dafür sorgen, dass dein Computer eine oder mehrere bestimmte Aufgaben erledigt. Stell dir mal eine Stoppuhr-App auf deinem Smartphone vor. Eine solche App ist nichts anderes als ein Programm mit Anweisungen, die dem Smartphone sagen, was zu tun ist, wenn du auf den START- oder STOP-Button tippst. Sie umfasst ferner Anweisungen zum Anzeigen der verstrichenen Zeit auf dem Bildschirm. Irgendjemand hat diese Stoppuhr-App so programmiert, dass sie funktioniert.

In der ganzen Welt werden Tag für Tag Millionen von Programmen eingesetzt. Die Messaging-App auf deinem Handy ist ein Programm, Verkehrsampeln werden mit Programmen gesteuert, und sogar Computerspiele wie Minecraft sind Programme.

In diesem Buch lernst du die Grundlagen der Programmierung kennen und erfährst, wie man eigene Ideen in Minecraft durch das Schreiben von Programmen umsetzt.

Daten in Variablen speichern

Beginnen wir mit der Speicherung von Daten in Variablen. *Variablen* gestatten dir das Speichern von Daten, die du an späterer Stelle im Programm noch einmal brauchst. *Daten* können dabei alles Mögliche sein, was gespeichert werden soll: Zahlenwerte, Namen, beliebige Texte, Objektlisten usw. Hier beispielsweise ist eine Variable namens `pickaxes`, die den Zahlenwert 12 speichert:

```
>>> pickaxes = 12
```

Variablen können Zahlen, Wörter und sogar ganze Sätze speichern, etwa »Hinfort mit dir, Creeper!«. Du kannst Variablenwerte auch ändern und damit in Minecraft einige nette Sachen machen. Und die Macht der Teleportation, wie wir sie gleich ausprobieren werden, basiert ebenfalls auf Variablen!

Zum Erstellen einer Variablen in Python brauchst du einen Variablennamen, ein Gleichheitszeichen (=) und einen Wert. Nehmen wir an, du bist im Begriff, dich auf eine lange Reise durch die unterschiedlichsten Minecraft-Biome zu machen, und möchtest dafür eine Menge Nahrungsmittel mitnehmen. Die Nahrungsmittel lassen sich als Variable darstellen. Beispielsweise ist in der folgenden Python-Shell `bread` der Variablenname und 145 der Wert:

```
>>> bread = 145
```

Der Variablenname steht immer links vom Gleichheitszeichen und der Wert, den du in der Variablen speicherst, steht rechts vom Gleichheitszeichen (siehe Bild 2.1). Diese Python-Codezeile *deklariert* die Variable `bread` (d.h. macht sie dem Computer bekannt) und nimmt eine *Zuweisung* des Werts 145 vor.

```
bread = 145
```

Variablenname Wert

Bild 2.1:
Teile einer Variablendeklaration. Für 145 Brotlaibe musst du schon echt hungrig sein.

Nach Deklaration und Wertzuweisung kannst du den Namen der Variablen in der Python-Shell eingeben, um ihren Wert abzufragen:

```
>>> bread  
145
```

Du kannst praktisch jeden Namen für eine Variable verwenden, doch bietet es sich an, eine aussagekräftige Bezeichnung zu vergeben, damit du nachvollziehen kannst, was in deinem Programm vor sich geht. Ebenfalls keine festgelegte Regel, aber trotzdem zu empfehlen ist eine Benennung mit einem Klein- statt einem Großbuchstaben am Anfang. An diese Vorgabe halten sich die meisten Python-Programmierer, und das solltest du auch tun, damit andere deinen Code leichter lesen können, falls du entscheiden solltest, ihn weiterzugeben.

Hinweis

Bitte beachte, dass der Wert einer Variablen *flüchtig* ist und nicht *gespeichert* wird. Der Variablenwert wird im Arbeitsspeicher des Computers abgelegt – wird das Programm beendet oder der Computer abgeschaltet, dann geht der Wert verloren. Schließe doch einmal IDLE, und starte es dann erneut. Was wird nun passieren, wenn du den Wert von *bread* abrufst?

Programmiersprachen und ihre Struktur

Als *Syntax* werden die Regeln bezeichnet, die die Grammatik und Zeichensetzung einer Programmiersprache beschreiben – ähnlich der Grammatik und Zeichensetzung in der menschlichen Sprache. Wenn du die Syntax von Python verstanden hast, wirst du in der Lage sein, Programme zu schreiben, die ein Computer abarbeiten kann. Verwendest du jedoch eine fehlerhafte Syntax, dann wird der Computer nicht verstehen, was du ihm sagen willst.

Stell dir eine einzelne Anweisung in deinem Code als Satz vor. Im Deutschen steht am Ende eines Satzes ein Punkt. Python verwendet anstelle des Punkts einen Zeilenwechsel, um das Ende einer Anweisung und den Anfang der nächsten zu signalisieren. Jede Anweisung in einer neuen Zeile wird auch als *Statement* bezeichnet.

Nehmen wir nun beispielshalber an, du möchtest festlegen, wie viele Spitzhacken, Eisenerzblöcke und Bruchsteinblöcke du hast. In der Python-Shell könntest du nun Folgendes schreiben:

```
>>> pickaxes = 12
>>> iron = 30
>>> cobblestone = 25
```

Bild 2.2 zeigt, wie das in der Python-Shell aussieht.

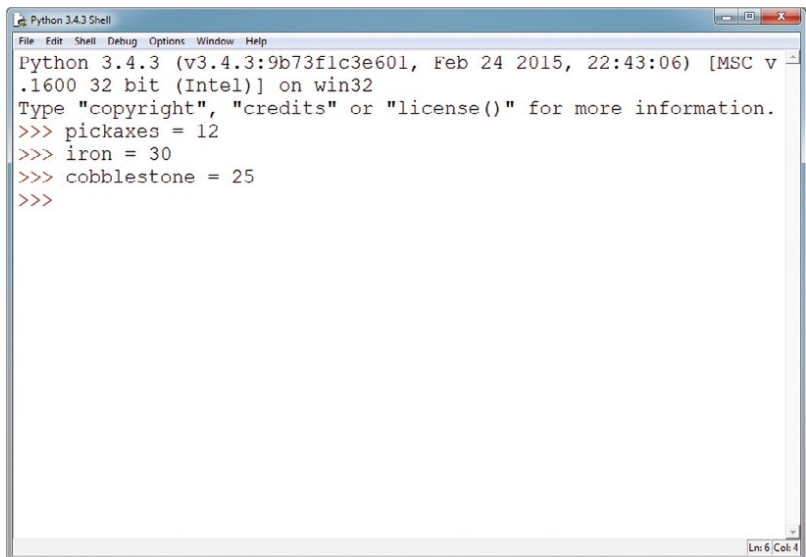


Bild 2.2: Codeeingabe in der Python-Shell

Beachte, dass jede Anweisung in einer eigenen Zeile steht. Durch die Zeilenwechsel weiß Python, dass du drei verschiedene Elemente mit Werten versiehst. Legst du hingegen nicht jede Anweisung in einer eigenen Zeile ab, dann verwirrt dies Python, und ein Syntaxfehler wird ausgegeben:

```
>>> pickaxes = 12 iron = 30 cobblestone = 25
SyntaxError: invalid syntax
```

Mit einem *Syntaxfehler* sagt Python dir, dass es etwas nicht verstanden hat. Python kann diese Anweisungen nicht verarbeiten, weil es nicht erkennen kann, wo eine Anweisung endet und wo die nächste beginnt.

Außerdem weiß Python auch nicht, was es machen soll, wenn du am Anfang einer Zeile ein Leerzeichen setzt:

```
>>> iron = 30
SyntaxError: unexpected indent
```

Wenn du genau hinsiehst, wirst du sehen, dass ich am Anfang der Zeile ein paar Leerzeichen gesetzt habe. Wenn du einen Syntaxfehler aufgrund eines unerwarteten Einzugs (engl. *unexpected indent*) erhältst, kannst du davon ausgehen, dass Leerzeichen am Anfang einer Zeile stehen, wo sie das nicht tun sollten.

Was das Schreiben von Code angeht, ist Python ausgesprochen pingelig. Wenn beim Eingeben der Beispiele aus diesem Buch ein Syntaxfehler auftritt, überprüfe deine Eingabe sorgfältig. Höchstwahrscheinlich hat sich irgendwo ein kleiner Fehler eingeschlichen.

Syntaxregeln für Variablen

Einige wenige Syntaxregeln zur Benennung von Variablen musst du kennen, um Missverständnisse aufseiten von Python zu vermeiden.

- Python 3 erlaubt sogenannte Sonderzeichen in den Variablennamen (dazu gehören z. B. ä, ö oder ü), doch rate ich von deren Verwendung ab. Beschränke dich am besten auf die Buchstaben a–z, die Ziffern 0–9 und den Unterstrich (_).
- Am Anfang eines Variablennamens darf keine Ziffer stehen. So etwas wie 9bread ist verboten. An allen anderen Stellen im Variablennamen dürfen Ziffern hingegen verwendet werden (z. B. bread9).
- Leerzeichen zu beiden Seiten des Gleichheitszeichens sind nicht erforderlich – dein Programm läuft auch ohne sie. Allerdings verbessern Leerzeichen die Lesbarkeit des Codes und sollten deswegen stets hinzugefügt werden.

Variablen sind etwas sehr Praktisches. Als Nächstes wirst du lernen, wie du den Wert einer Variablen ändern kannst. Dann bist du bereit für die Teleportation deines Spielers.

Variablenwerte ändern

Du kannst den Wert einer Variablen jederzeit verändern. Dafür benutzt du die gleiche Anweisung wie beim Deklarieren. Angenommen, du triffst in Minecraft fünf Katzen und möchtest diesen Wert als Variable speichern.

Zunächst deklarierst du also eine Variable `cats` und weist ihr den Wert 5 zu. In der Python-Shell sähe das dann so aus:

```
>>> cats = 5
>>> cats
5
```

Später triffst du fünf weitere Katzen und möchtest den Wert dieser Variablen deswegen aktualisieren. Was passiert jetzt, wenn du den Wert von `cats` auf 10 setzt?

```
>>> cats = 10
>>> cats
10
```

Wenn du den neuen Wert von `cats` in Python abfragst, beträgt er nicht mehr 5! Würdest du die Variable `cats` in einem Programm verwenden, dann hätte diese nun den neuen Wert 10.

In Variablen lassen sich Daten aller Art ablegen. Am *Datentyp* erkennt der Computer, wie er mit einem bestimmten Datenelement umgehen muss. Ich werde zunächst einen Datentyp beschreiben, mit dem du es sehr oft zu tun bekommen wirst: Integer (ganze Zahlen). Im weiteren Verlauf des Kapitels kommen dann die Floats (Fließkommazahlen) hinzu.

Integer

Integer sind positive oder negative ganze Zahlen. Werte wie 10, 32, -6, 194689 oder -5 sind Integer, Zahlen wie 3,14 und 6,025 jedoch nicht.

Wahrscheinlich verwendest du Integer-Zahlen Tag für Tag, ohne es überhaupt zu merken – sogar in Minecraft. Stell dir vor, du siehst zwölf Kühe auf einer Weide, während du unterwegs bist, um fünf Diamanten zu schürfen, und in deinem Inventar befinden sich zwei frische Äpfel. All diese Zahlen sind Integer.

Nehmen wir nun an, du besitzt in deiner Minecraft-Welt fünf Schweine und möchtest nun ein Programm schreiben, das die Anzahl der Schweine auf irgendeine Weise nutzt. In Python musst du eine Integer-Variable deklarieren, um die Anzahl der Schweine darzustellen (ich verwende hier englische Begriffe, du kannst natürlich deutsche verwenden):

```
>>> pigs = 5
```

Du kannst in Variablen auch negative Werte speichern. Um beispielsweise eine Temperatur von 5° unter null zu speichern, würdest du wie folgt deklarieren:

```
>>> temperature = -5
```

Um Python-Variablen und Integer mal mit Minecraft zu verwenden, musst du nun deine erste Mission absolvieren.

Mission 1: Den Spieler teleportieren

Bei dieser Mission untersuchst du die Wirkungsweise von Variablen. Hierzu teleportierst du deinen Spieler mithilfe von Integern an eine andere Position.

Wie Bild 2.3 zeigt, hat dein Spieler in der Minecraft-Welt eine *Position*, die durch die drei *Koordinaten* *x*, *y* und *z* dargestellt wird. Der Buchstabe *y* steht dabei für die Höhe; *x* und *z* stellen horizontale Positionen auf einer Ebene dar.

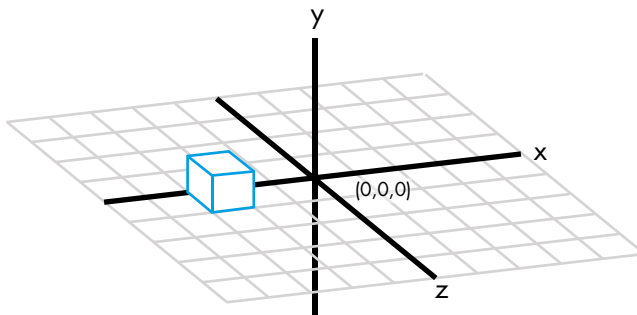


Bild 2.3: 3D-Koordinaten

Wenn du die Raspberry-Pi-Version des Spiels verwendest, wird die Position des Spielers durch drei Zahlen oben links in der Ecke des Spielfensters angegeben (Bild 2.4). In der Desktopversion kannst du die Koordinaten durch Drücken von **F3** anzeigen, indem du in der ersten Zeile des zweiten Textblocks links nach dem Eintrag XYZ suchst (Bild 2.5).

Bewege deinen Spieler im Spiel, und achte dabei darauf, wie sich die Positionszahlen verändern. Dies sollte beim Bewegen des Spielers in Echtzeit erfolgen. Eigentlich ganz cool, oder? Allerdings benötigt man hierbei für lange Wege viel Zeit. Wäre es nicht viel besser, wenn man – über Python –

direkt zu anderen Positionen springen könnte? Wir wollen uns einmal ansehen, wie das funktioniert.

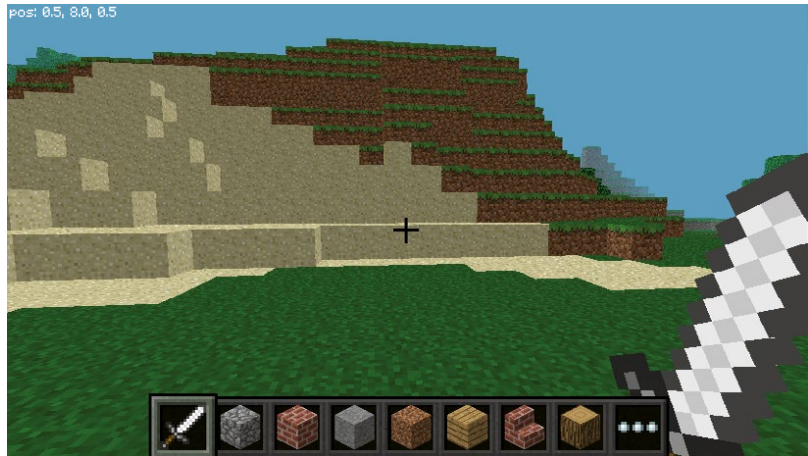


Bild 2.4: Spielerposition in der Minecraft: Pi Edition



Bild 2.5: Spielerposition in der Desktopversion von Minecraft

Schalte deinen Computer oder deinen Raspberry Pi ein, und führe die folgenden Schritte aus:

1. Öffne IDLE, und klicke auf **File ▶ New File** (auf einigen Computern musst du stattdessen **New Window** auswählen). Nun siehst du den

leeren Texteditor (Bild 2.6). Wenn du den Raspberry Pi verwendest oder wenn auf deinem Computer mehrere Python-Versionen installiert sind, dann verwende in jedem Fall Python 3, nicht Python 2.7.

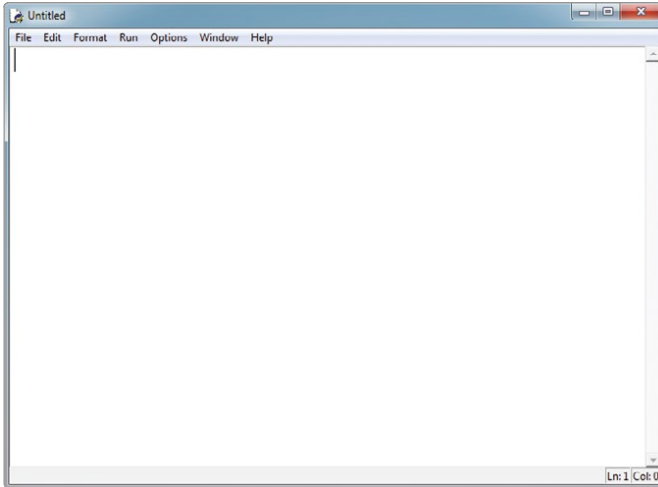


Bild 2.6: Neues Texteditorfenster in IDLE

2. Klicke nach dem Erscheinen des neuen Fensters auf **File ▶ Save As**.
3. Erstelle einen neuen Ordner namens *variables* in dem Ordner *Minecraft Python*, den du in Kapitel 1 angelegt hast.
4. Öffne den Ordner *variables*, gib deiner Datei den Namen *teleport.py*, und klicke auf **Save**.

Nun wechselst du in den IDLE-Texteditor und fügst dort die folgenden zwei Codezeilen am Anfang des Programms ein:

```
from mcpi.minecraft import Minecraft  
mc = Minecraft.create()
```

Über diese Zeilen verbindest du dein Programm mit Minecraft. Du musst sie in jedem Programm verwenden, das mit Minecraft interagieren soll. Als Nächstes erstellst du drei Integer-Variablen namens *x*, *y* und *z*:

```
x = 10  
y = 110  
z = 12
```

Diese Variablen stellen die Position dar, an die du deinen Spieler teleportieren möchtest. Lege sie zunächst wie hier gezeigt auf 10, 110 und 12 fest.

Danach gibst du die folgende Codezeile ein, die für den Transport des Spielers zuständig ist:

```
mc.player.setTilePos(x, y, z)
```

`setTilePos()` ist eine sogenannte *Funktion*, d. h. ein direkt einsatzbereites, wiederverwendbares Codeobjekt. Mit der Funktion `setTilePos(x, y, z)` wird Minecraft angewiesen, die Position des Spielers unter Verwendung der drei oben festgelegten Variablen zu ändern. Die Werte in den Klammern heißen *Argumente*. Du hast die von dir erstellten Variablen als Argumente an die Funktion *übergeben*, und deswegen kann sie die Werte von `x`, `y` und `z` bei der Ausführung verwenden.

Achtung

Auf dem Raspberry Pi darfst du für die Variablen `x` und `z` keine Werte angeben, die größer als 127 oder kleiner als -127 sind. Die Minecraft-Welt auf dem Pi ist klein, und Werte außerhalb dieses Bereichs führen zu einem Absturz.

Listing 2.1 enthält den gesamten Code zum Teleportieren des Spielers, der auch in Bild 2.7 gezeigt ist:

teleport.py

```
❶ # Verbindung mit Minecraft herstellen
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

# Variablen x, y und z zur Darstellung der Koordinaten festlegen
x = 10
y = 110
z = 12

# Position des Spielers ändern
mc.player.setTilePos(x, y, z)
```

Listing 2.1: Fertiger Teleportationscode

Damit du das Programm einfacher nachvollziehen kannst, habe ich einen *Kommentar* ❶ ergänzt. Kommentare sind nützliche Anmerkungen, die beschreiben, was an der betreffenden Stelle im Code geschieht. Python ignoriert diese Anmerkungen jedoch. Mit anderen Worten: Wenn du das Programm ausführst, übergibt Python kommentierte Zeilen, ohne irgendetwas zu tun. Ein einzeliger Kommentar beginnt mit einer Raute (#). Meine Kommentare beschreiben, was die einzelnen Teile von *teleport.py* tun. Es ist eine gängige und empfohlene Praxis, den eigenen Code mit Kommentaren zu versehen. So kannst du dich später, wenn du deinen Code noch einmal liest, besser daran erinnern, was die einzelnen Teile des Programms tun sollen.

Bild 2.7 zeigt das vollständige Programm im IDLE-Texteditor.

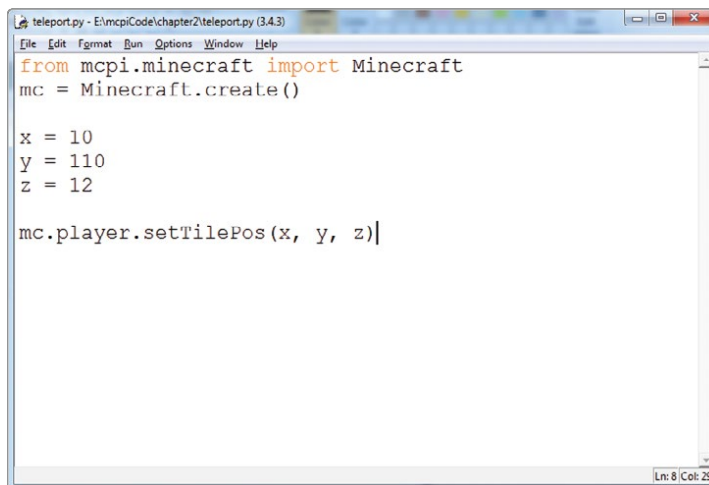


Bild 2.7: Vollständiges Programm im IDLE-Texteditor

Nun wollen wir das Programm starten, um zu sehen, ob alles funktioniert. Führe dazu die folgenden Schritte aus:

1. Öffne Minecraft, indem du auf das Desktopsymbol klickst.
2. Wenn Du einen Raspberry Pi verwendest, klicke auf **Start Game** und dann auf **Create a New World**. In der Desktopversion von Minecraft öffnest du die Spielwelt wie in »Spigot ausführen und ein Spiel erstellen« auf Seite 9 für Windows und auf Seite 21 für den Mac beschrieben.
3. Nach dem Erstellen der Welt drückst du auf die **Esc**-Taste (bzw. beim Raspberry Pi auf die **Tabulatortaste**), um den Mauszeiger aus dem Spiel zu holen. Du kannst die Maus nun außerhalb des

Minecraft-Fensters bewegen. Wenn du das Spiel fortsetzen möchtest, doppelklickst du in das Minecraft-Fenster. Bild 2.8 zeigt das IDLE- und das Minecraft-Fenster auf meinem Computer.



Bild 2.8: So sind das Minecraft-Fenster und der IDLE-Texteditor auf meinem Computer angeordnet.

4. Klicke auf das IDLE-Texteditorfenster mit deinem Programm *teleport.py*.
5. Klicke auf **Run ▶ Run Module** oder drücke **F5**. Falls du das Programm noch nicht gespeichert hast, wird IDLE dich immer fragen, ob du dies vor dem Ausführen tun möchtest. Klicke auf **OK**, um das Programm zu speichern. Wenn du auf **Cancel** klickst, wird das Programm nicht ausgeführt.

Hinweis

Wenn du Programme aus IDLE auf einem Raspberry Pi startest, wird ein Dialogfeld angezeigt, das dich zum Speichern des Programms auffordert. Dieses Feld ist jedoch möglicherweise hinter dem Minecraft-Fenster verborgen. Wenn IDLE scheinbar abgestürzt ist, kann das schlicht und einfach daran liegen, dass das Dialogfeld nicht zu sehen ist. In diesem Fall musst du das Minecraft-Fenster minimieren und dann im IDLE-Dialogfeld auf **OK** klicken. Danach musst du das Minecraft-Fenster wieder maximieren.

Gut gemacht! Nun sollte dein Programm gestartet werden, und nach einigen Sekunden wird dein Spieler zu den Koordinaten (10, 110, 12) teleportiert (Bild 2.9). Deine Welt sieht nicht genauso aus wie meine, weswegen sich die Ausführung auf deinem Computer etwas anders gestalten wird.

Bonus: Durch die Gegend springen

Glaubst du, du hast den Dreh mit der Teleportation raus? Ersetze x, y und z durch andere Integer, und finde heraus, wo dein Spieler landet. Du kannst auch negative Werte ausprobieren!



Bild 2.9: Ich habe den Spieler von meinem Haus zur Position (10, 110, 12) teleportiert, die sich über einem Moor befindet. Schau mal nach unten!

Float-Variablen

Nicht alle Zahlen sind ganze Zahlen. Zahlenwerte, die keine ganzen Zahlen (also Integer) sind, werden mit Nachkommastellen bzw. – im Englischen und damit auch in Python – mit Nachpunktstellen geschrieben. So kann man beispielsweise 0,5 Äpfel haben, was einem halben Apfel entspricht. Zahlen mit Nachkommastellen werden als *Fließkommazahlen* oder (auf Englisch) als *Floats* bezeichnet. Dies ist ein weiterer Datentyp, den Python verwendet. Fließkommazahlen werden anstelle von Integern benutzt, wenn eine größere Genauigkeit erzielt werden soll. Natürlich können Floats auch ganze Zahlen darstellen (z. B. 3.0), während Integer nicht in der Lage sind, Zahlen mit Nachkommastellen aufzunehmen.

Du hast vielleicht bemerkt, dass die Positionskoordinaten deines Spielers in den Bildern 2.4 und 2.5 Dezimalstellen aufweisen – es sind also Floats!

In Python kannst du eine Float-Variable auf gleiche Weise deklarieren wie einen Integer-Wert. Um also etwa die Variable `x` auf 1.34 zu setzen, schreibst du:

```
>>> x = 1.34
```

Wichtig: Du musst einen »Punkt« (.) statt eines Kommas verwenden, sonst erhältst du einen Syntaxfehler – also »1.34« statt »1,34«. Das liegt daran, dass im englischen Sprachraum der Punkt als Dezimaltrenner verwendet wird.

Willst du eine negative Float-Zahl erstellen, dann setzt du einfach das Minuszeichen (-) vor den Wert:

```
>>> x = -1.34
```

In der nächsten Mission erhältst du noch mehr Kontrolle über deine Teleportationskräfte, denn mithilfe von Floats werden wir unseren Spieler noch präziser in unserer Welt positionieren.

Mission 2: Genau dort landen, wo man hinwill

Du weißt schon, wie du die Spielerposition mit Integer-Werten festlegen kannst. Mit Floats ist das aber noch genauer möglich. Bei dieser Mission werden wir das Programm aus Mission 1 so überarbeiten, dass der Spieler mithilfe eines Float-Werts teleportiert wird:

1. Öffne in IDLE das Programm *teleport.py* (Seite 44), indem du auf **File ▶ Open** klickst und die Datei in deinem Ordner *variables* auswählst.
2. Speichere eine Kopie dieses Programms als *teleportPrecise.py* in deinem Ordner *variables*.
3. Lege in der Datei *teleportPrecise.py* die Variablen *x*, *y* und *z* so fest, dass Floats anstelle von Integern verwendet werden. Setze also die Werte von *x*, *y* und *z* von 10, 110 und 12 auf 10.0, 110.0 und 12.0.
4. Ändere die letzte Codezeile in `mc.player.setPos(x, y, z)`. Hierzu musst du lediglich das Wort `Tile` entfernen.
5. Speichere das Programm.
6. Öffne eine Minecraft-Welt, und führe den Code aus.

Das endgültige Ergebnis sollte wie folgt aussehen:

```
# Verbindung mit Minecraft herstellen
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
```

```
# Variablen x, y und z zur Darstellung der Koordinaten festlegen
x = 10,0
y = 110,0
z = 12,0
```

```
# Position des Spielers ändern
mc.player.setPos(x, y, z)
```

teleportPrecise
.py

Beachte den Unterschied zwischen dem hier verwendeten `mc.player.setPos(x, y, z)` und der Variante `mc.player.setTilePos(x, y, z)` aus Listing 2.1. Die Funktion `setTilePos()` verwendet Integer, um die Position des Blocks anzugeben, zu dem du dich teleportieren lassen möchtest. `setPos()` funktioniert ein bisschen anders: Mithilfe von Floats wird nicht nur die Position des Blocks angegeben, zu dem du dich teleportieren lassen möchtest, sondern auch die exakte Position *auf* diesem Block. Mit meinem Programm habe ich mich auf die Spitze meines Turms teleportiert (Bild 2.10).



Bild 2.10: Mit Floats bin ich präzise auf der Spitze meines Turms gelandet.

Bonus: Exaktes Teleportieren

Ändere die Werte der Variablen x , y und z . Verwende dabei einen Mix aus positiven und negativen Werten. Starte das Programm. Nun ändere die Dezimalwerte geringfügig ab. Was geschieht?

Teleportation mit dem time-Modul verlangsamen

Python führt deinen Code so schnell wie möglich aus. Du kannst Vorgänge allerdings auch verlangsamen, indem du deine Programme vor der Fortsetzung einige Sekunden warten lässt.

Um die Zeit in deinen Programmen zu beeinflussen, benötigst du das `time`-Modul, das eine Anzahl fertiger Funktionen für alles Mögliche umfasst, das mit dem Thema »Zeit« zu tun hat. Damit du das `time`-Modul verwenden kannst, musst du die folgende Codezeile ganz oben in deinen Programmen hinzufügen:

```
import time
```

Wenn du das `time`-Modul und die Funktion `sleep()` verwendest, die Bestandteil dieses Moduls ist, musst du in jedem Fall die richtige Reihenfolge beachten. Mit der `sleep()`-Funktion kannst du im Programm eine Pause mit einer bestimmten, in Sekunden angegebenen Länge festlegen. Du musst das `time`-Modul stets vor Verwendung der `sleep()`-Funktion importieren, sonst kann Python diese Funktion nicht finden und wird dadurch so sehr aus dem Tritt gebracht, dass es dein Programm nicht mehr weiter ausführt. Deswegen ist es immer am besten, alle im Code verwendeten Module stets zu Anfang zu importieren. Ich empfehle dir, alle `import`-Anweisungen am Beginn deines Programms zu gruppieren. Meine Struktur sieht beispielsweise so aus, dass ich zunächst die Codezeilen zum Herstellen der Verbindung mit Minecraft ganz oben einfüge und dass die `import time`-Anweisung dann in der dritten Zeile landet.

Hier siehst du ein Beispiel dafür, wie ich die `sleep()`-Funktion verwende:

```
time.sleep(5)
```

Mit dieser Codezeile unterbrichst du dein Programm für fünf Sekunden. Du kannst einen beliebigen Integer- oder Float-Wert angeben, wie auch das folgende Beispiel zeigt:

```
time.sleep(0.2)
```

Wenn ein Programm auf diese Codezeile stößt, wird es eine Pause von 0,2 Sekunden einlegen. Nachdem du nun gelernt hast, wie du zeitliche Abläufe steuern kannst, kommen wir zur nächsten Mission.

Mission 3: Teleportationstour

Das Schöne an der Teleportation in Minecraft ist, dass du deinen Spieler hinschicken kannst, wohin du möchtest. Mit den Fähigkeiten, die du schon erlernt hast, wirst du deinen Spieler nun auf eine vollautomatische Reise durch deine gesamte Minecraft-Welt senden.

Bei dieser Mission wirst du das Ändern von Variablenwerten üben. Hierzu modifizieren wir den Code aus Mission 1 (Seite 41) so, dass der Spieler an mehrere Positionen teleportiert wird, die auf der gesamten Karte verteilt sind. Der Spieler wird also zunächst an eine Position geschickt, dann wartet das Programm einige Sekunden und teleportiert ihn weiter.

1. Öffne in IDLE das Programm *teleport.py* (Seite 44), indem du auf **File ▶ Open** klickst und die Datei in deinem Ordner *variables* auswählst.
2. Speichere eine Kopie dieses Programms als *tour.py* in deinem Ordner *variables*.
3. Füge direkt hinter dem Code, der dein Programm mit Minecraft verbindet, `import time` ein.
4. Am Ende des Programms fügst du `time.sleep(10)` hinzu.
5. Nun kopierst du die Zeilen mit den Variablen `x`, `y` und `z` und der Funktion `setTilePos()` in die Zwischenablage und fügst sie am Ende ein, sodass sie zweimal im Programm vorhanden sind.
6. Ändere die Werte der beiden Variablensätze `x`, `y` und `z` beliebig ab. Du kannst die Koordinaten jeder Position in deinem Spiel ermitteln, indem du dorthin gehst und die Koordinaten des Spielers so notierst, wie ich weiter oben beschrieben habe.
7. Speichere das Programm.
8. Öffne eine Minecraft-Welt, und führe den Code aus.

Das endgültige Ergebnis einschließlich der neu eingefügten Koordinaten sollte wie folgt aussehen:

tour.py

```
# Verbindung mit Minecraft herstellen
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
import time

# Variablen x, y und z zur Darstellung der Koordinaten festlegen
x = Wert eintragen
y = Wert eintragen
z = Wert eintragen

# Position des Spielers ändern
mc.player.setTilePos(x, y, z)

# 10 Sekunden warten
time.sleep(10)

# Variablen x, y und z zur Darstellung der Koordinaten festlegen
x = Wert eintragen
y = Wert eintragen
z = Wert eintragen

# Position des Spielers ändern
mc.player.setTilePos(x, y, z)
```

Der Spieler sollte nun zur ersten Position teleportiert werden und nach einer zehnstündigen Pause zur zweiten Position gelangen (Bild 2.11).

Bonus: Noch mehr Teleportation

Kopiere den Code von `tour.py`, um den Spieler beliebig häufig neu zu platzieren. Ersetze die 10 in der Funktion `time.sleep(10)` durch einen anderen Wert. Du kannst auch jeder `sleep()`-Funktion einen anderen Wert zuweisen, sodass der Spieler jedes Mal unterschiedlich lange wartet.

Danach probiere mal, bei jeder Änderung nur einen der Variablenwerte `x`, `y` und `z` zu ändern. Du musst nämlich gar nicht jede Variable jedes Mal neu zuweisen! Benutze abschließend auch einmal Floats anstelle von Integern.



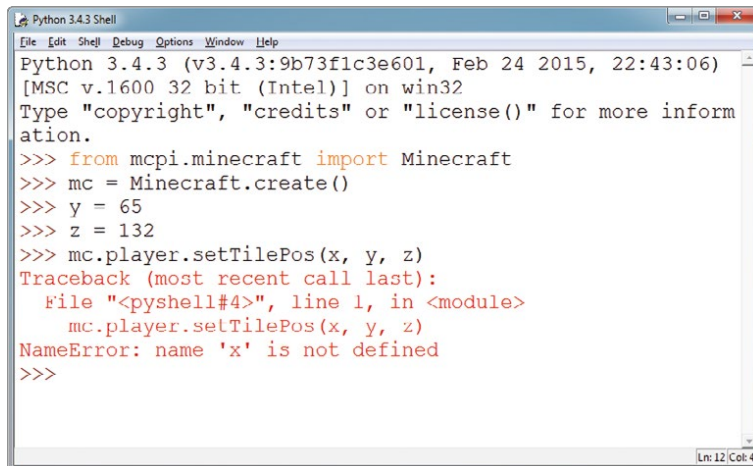
Bild 2.11: Ich habe die Koordinaten in meinem Programm so festgelegt, dass ich zunächst nach Hause und dann in die Wüste geschickt werde.

Debuggen

Jeder macht mal einen Fehler, und auch die besten Programmierer bekommen ihren Code nicht unbedingt auf Anhieb zum Laufen. Ein lauffähiges Programm zu schreiben ist nur eine der Fähigkeiten, die einen guten Programmierer ausmachen. Ein guter Programmierer weiß auch, wie er Programmierfehler findet und loswird. Diesen Vorgang nennt man *Debuggen*, und einzelne Probleme in einem fehlerhaften Programm werden als *Bugs* (Wanzen) bezeichnet. In diesem Abschnitt wirst du einige

Tipps und Tricks lernen, damit du in der Lage bist, alle deine Programme künftig von Fehlern zu befreien.

Bugs können dafür sorgen, dass ein Programm entweder nicht mehr ausgeführt wird oder aber sich nicht so verhält, wie du es erwartest. Läuft ein Programm nicht, dann zeigt Python eine Fehlermeldung an. Bild 2.12 zeigt ein Beispiel.

A screenshot of a Python 3.4.3 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area shows the following code and output:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06)
[MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more inform
ation.
>>> from mcpi.minecraft import Minecraft
>>> mc = Minecraft.create()
>>> y = 65
>>> z = 132
>>> mc.player.setTilePos(x, y, z)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    mc.player.setTilePos(x, y, z)
NameError: name 'x' is not defined
>>>
```

The status bar at the bottom right indicates 'Ln:12/Col:4'.

Bild 2.12: Python zeigt eine Fehlermeldung an, weil ich mich nicht an die vorgeschriebene Syntax gehalten habe.

In Bild 2.12 ist zu sehen, dass ich etwas Code in die Python-Shell eingegeben habe, woraufhin mir eine Fehlermeldung angezeigt wird. Die Fehlermeldung enthält eine Menge Informationen, doch reicht es aus, die letzte Zeile (NameError: name 'x' is not defined) zu betrachten, um zu wissen, dass mit meiner Variablen `x` etwas nicht stimmt. Ich habe nämlich schlicht vergessen, die Variable `x` zu definieren. Zur Korrektur füge ich eine zusätzliche Codezeile ein, die die Variable `x` definiert:

```
>>> x = 10
```

Hierdurch wird dieser Fehler behoben, was aber noch nicht bedeutet, dass das ganze Programm fehlerfrei sein muss.

Bugs, die die Lauffähigkeit des Programms nicht beeinträchtigen, aber zu einem unerwünschten Verhalten führen, zeigen zwar keine Fehlermeldungen an, aber du wirst trotzdem schnell merken, dass irgendetwas nicht stimmt. Wenn du etwa eine Codezeile in deinen Teleportationsprogrammen vergisst (z. B. `setTilePos()`), dann wird das Programm zwar

fehlerfrei ausgeführt, aber die Position des Spielers ändert sich nicht. Und das ist für ein Teleportationsprogramm sicher kein erwünschtes Verhalten.

Achtung

Zu den häufigsten Ursachen für Bugs gehören Tippfehler. Wenn du etwas so hinschreibst, dass der Computer nichts damit anfangen kann, führt das dazu, dass das Programm nicht läuft. Gehe sorgfältig vor, und vergewissere dich, dass die Schreibweise und die Groß-/Kleinschreibung korrekt sind!

Mission 4: Bug im Teleportationsprogramm beheben

In dieser Mission werden wir zwei Programme berichtigen. Das erste Programm (Listing 2.2) ähnelt *teleport.py* (Seite 44), doch enthält diese Version fünf Fehler. Öffne eine neue Datei im IDLE-Texteditor, kopiere Listing 2.2 dorthin, und speichere das Ergebnis als *teleportBug1.py*.

teleportBug1
.py

```
from mcpi.minecraft import Minecraft
# mc = Minecraft.create()

x = 10
y = 11
z = 12
```

Listing 2.2: *Schadhafte Version des Teleportationsprogramms*

Zum Debuggen dieses Programms führst du die folgenden Schritte aus:

1. Starte *teleportBug1.py*.
2. Sobald eine Fehlermeldung angezeigt wird, lies die letzte Zeile, um Informationen zum Fehler zu erhalten.
3. Behebe den Fehler, und führe den Code erneut aus.
4. Wiederhole die Fehlerkorrekturschritte so lange, bis das Programm den Spieler wie gewünscht an eine neue Position teleportiert.

Hinweis

Vergiss nicht, darauf zu achten, dass im Programm tatsächlich die Funktion `setTilePos()` aufgerufen wird!

Nun wollen wir ein anderes Programm debuggen. Die Version von *teleport.py* in Listing 2.3 wird zwar vollständig ausgeführt, doch erfolgt keine Teleportation des Spielers zur angegebenen Position. Kopiere Listing 2.3 in eine IDLE-Datei, und speichere sie als *teleportBug2.py*.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

x = 10
y = 110
z = -12

mc.player.setPos(x, z, y)
```

teleportBug2
.py

Listing 2.3: Teleportationsprogramm mit Bugs

Anders als bei *teleportBug1.py* wird hier bei der Programmausführung keine Fehlermeldung angezeigt. Um dieses Programm zu korrigieren, musst du also den Code so lange unter die Lupe nehmen, bis du den Fehler gefunden hast. Das Programm soll den Spieler zur Position (10, 110, -12) bringen. Führe das Programm aus, und überprüfe die Koordinaten, an die der Spieler tatsächlich teleportiert wird. Diese Information kann beim Debuggen des Programms und bei der Erkennung des Problems hilfreich sein.

Wenn du alle Bugs in diesen beiden Programmen erwischt hast, solltest du jeden davon mit einem Kommentar versehen, um das jeweilige Problem zu erläutern. Wenn du dir die Probleme aufschreibst, die du beim Debuggen gefunden hast, wirst du die entsprechenden Fehler bei zukünftigen Projekten leichter erkennen oder sogar ganz vermeiden.

Was du gelernt hast

Herzlichen Glückwunsch! Du hast nun deine ersten Python-Programme zur Steuerung eines Minecraft-Spielers mithilfe von Variablen und Funktionen geschrieben. Du hast außerdem zwei Datentypen (Integer und Floats) kennengelernt, und du weißt jetzt, wie man die Zeit steuern und Programme debuggen kann. Außerdem kennst du mit `setPos()` und `setTilePos()` bereits zwei nützliche Funktionen aus der Minecraft-Python-API.

In Kapitel 3 werden wir uns mit dem »Speed Building« befassen: also mit dem schnellen Platzieren von Blöcken mit mathematischen Operationen und Funktionen!

Kapitel 3

Schneller bauen und reisen – Mathe macht's möglich

In Kapitel 2 hast du gelernt, wie man eine Variable erstellt und ihren Wert ändert. In diesem Kapitel wirst du nun erfahren, wie du mit Mathematik in Python beliebige Blöcke erstellst und in deiner Minecraft-Welt schnell komplexe Strukturen baust. Und du bekommst Superkräfte, denn dein Spieler kann Riesenschritte machen!

Ausdrücke und Anweisungen

Wenn du dich mit jemandem unterhältst, möchtest du, dass dein Gegenüber versteht, was du ihm sagst. Du verwendest kurze Phrasen wie »drei Diamanten« oder »hinter einem Baum«, um der angesprochenen Person Informationen zu vermitteln. Allerdings sind diese Phrasen erst dann sinnvoll, wenn wir sie zu ganzen Sätzen kombinieren – z. B. »Ich habe hinter einem Baum drei Diamanten gefunden.«

Die Programmiersprache Python verfolgt ein ähnliches Konzept von Phrasen und Sätzen, die hier jedoch *Ausdrücke* und *Anweisungen* heißen.

Du kannst Werte, Variablen und Operatoren zu kleinen Codeabschnitten kombinieren, die als *Ausdrücke* bezeichnet werden. $2 + 2$ etwa ist ein Ausdruck. Ausdrücke wiederum lassen sich in *Anweisungen* verwenden, die du ja bereits in Kapitel 2 kennengelernt hast. Anweisungen sind einzelne Zeilen oder kurze Codeblöcke, die in einem Programm etwas tun, z. B. `zombies = 2 + 2`. In diesem Beispiel ist $2 + 2$ ein Ausdruck und Bestandteil der Anweisung `zombies = 2 + 2`.

Bei längeren Programmen, die einen Texteditor anstelle der Python-Shell verwenden, musst du vollständige Anweisungen schreiben. Der Ausdruck $2 + 2$ etwa wird von der Python-Shell einerseits und einem in einem Texteditor geschriebenen Programm andererseits vollkommen unterschiedlich behandelt. Wenn du die Python-Shell in IDLE verwendest, gibt Python 4 als Ergebnis für $2 + 2$ aus:

```
>>> 2 + 2
4
```

Verwendest du hingegen den Texteditor, dann macht Python mit dem Ausdruck überhaupt nichts, da er nicht Bestandteil einer vollständigen Anweisung ist. Um aus diesem Ausdruck eine vollständige Anweisung zu machen, könntest du seinen Wert einer Variablen zuweisen:

```
zombies = 2 + 2
```

Danach gibst du die Variable aus, um den Wert anzuzeigen:

```
print(zombies)
```

Wenn du dann diesen Code ausführst, wird 4 ausgegeben.

Noch einmal: Wenn du Programme im Texteditor schreibst, ist es sehr wichtig, statt einfacher Ausdrücke stets vollständige Anweisungen zu verwenden.

Operatoren

In der Mathematik ermöglichen *Operatoren* das Ändern und Kombinieren von Zahlen. Mit dem Additionsoperator beispielsweise kannst du zwei (oder mehr) Zahlen addieren. Der Subtraktionsoperator dagegen wird verwendet, um eine Zahl von einer anderen abzuziehen.

Python verwendet alle mathematischen Grundoperatoren, die du bereits kennst – Addition, Subtraktion, Multiplikation und Division –, sowie eine Reihe weiterer Operatoren wie beispielsweise Exponenten. Beginnen wollen wir jedoch mit der Addition.

Addition

In Python sieht die Addition so aus, wie du sie auch im richtigen Leben schreiben würdest, denn sie verwendet das Pluszeichen (+). Hast du etwa zwei Blumen und pflückst noch zwei dazu, dann kann man dies auch als Additionsanweisung beschreiben:

```
>>> flowers = 2 + 2
```

Python ermittelt das Ergebnis des Ausdrucks auf der rechten Seite des Gleichheitszeichens und weist dieses dann der Variablen auf der linken Seite zu. In diesem Fall beträgt das Ergebnis des Ausdrucks auf der rechten Seite 4. Solange dieser Codeausschnitt gültig ist, behält die Variable `flowers` den Wert 4.

Mithilfe der Addition kannst du in Minecraft im Handumdrehen Dinge bauen. Bereit für die nächste Mission? Steigen wir gleich ein.

Mission 5: Blöcke stapeln

Mit der Funktion `setBlock()` kannst du in Minecraft einen Block erstellen und platzieren. Ähnlich wie `setPos()` und `setTilePos()` nimmt `setBlock()` `x`-, `y`- und `z`-Koordinaten als Argumente entgegen, benötigt aber noch einen vierten Wert: den Blocktyp. Dieser Wert gibt die Art des Blocks an, der im Spiel platziert werden soll.

Dabei wird jeder Typ – Gras, Lava, Melone und jeder andere Block – durch einen speziellen Integer dargestellt. Gras etwa hat den Wert 2, Luft 0, Wasser 8 und Melone 103. Eine vollständige Liste der Blöcke und der zugehörigen Integer-Werte findest du online unter den auf Seite xv genannten URLs.

Zur Verwendung von `setBlock()` übergibst du – durch Kommas getrennt – Werte für die `x`-, `y`- und `z`-Koordinaten und den Integer, der den Blocktyp darstellt. Angenommen, wir wollen einen Melonenblock (Typ 103) an den Koordinaten (6, 5, 28) ablegen (oder nimm Koordinaten in Deiner Nähe):

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
mc.setBlock(6, 5, 28, 103)
```

Nach den ersten beiden Zeilen, die ja in allen Minecraft-Python-Programmen vorhanden sind, rufst du einfach `setBlock()` mit den von dir

gewünschten Parameterwerten auf. Den gleichen Effekt kannst du auch erzielen, indem du Variablen anstelle von Zahlen verwendest (Listing 3.1).

blockStack.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
x = 6
y = 5
z = 28
blockType = 103
mc.setBlock(x, y, z, blockType)
```

Listing 3.1: Wir basteln uns eine Melone.

Zunächst werden die Variablen angelegt, die die Blockkoordinaten (x, y und z) sowie den Typ (blockType) darstellen. Danach übergibst du alle Variablen an die Funktion `setBlock()`, und die Minecraft-Python-API kann ihr Zauberwerk beginnen. Du kannst diese Variablen später an beliebiger anderer Stelle im Programm verwenden und zudem ihre Werte bei Bedarf ändern (wobei diese Änderung nur einmal erfolgen muss).

Kombinierst du diesen Code jetzt mit mathematischen Operatoren, dann kannst du damit einige ziemlich abgefahrene Sachen anstellen. Wir wollen einmal ein paar Blöcke stapeln.

Erstelle einen neuen Ordner namens *math* in deinem Ordner *Minecraft Python*. Öffne IDLE, und erstelle ein leeres Programm mit dem IDLE-Texteditor. Speichere diese Datei dann unter dem Namen *blockStack.py* im Ordner *math*. Als Nächstes kopierst du den Code aus Listing 3.1 in deinen Editor und fügst die beiden Zeilen aus Listing 3.2 hinzu, um auf den gerade eben aufgestellten Melonenblock einen zweiten zu setzen.

blockStack.py

```
❶ y = y + 1
❷ mc.setBlock(x, y, z, blockType)
```

Listing 3.2: Mit diesem zusätzlichen Code stapeln wir eine zweite Melone auf die erste.

Du addierst 1 zum Wert von y ❶ hinzu und verwendest dann die Funktion `setBlock()`, um einen neuen Block zu erstellen ❷. Durch Erhöhen des Wertes von y um 1 wird der zweite Block auf der y-Achse ein bisschen höher angeordnet als der erste, d. h., er wird direkt auf dem ersten Block abgelegt.

Deine Aufgabe besteht nun darin, zwei weitere Blöcke auf den Stapel zu setzen. Ändere dein Programm *blockStack.py* doch einmal so ab, dass vier

anstelle von zwei Blöcken gestapelt werden. Wenn du dein Programm jetzt ausführst, sollte ein Stapel mit vier Melonenblöcken entstehen (Bild 3.1).



Bild 3.1: Jetzt habe ich einen Stapel aus Melonen gebastelt.

Hinweis

Um einen zweiten Block auf den ersten zu setzen, haben wir die Variable `y` um 1 erhöht und die Funktion `setBlock()` noch einmal verwendet. Was würde deiner Meinung nach geschehen, wenn wir die beiden Anweisungen am Ende deines Programms noch einmal verwenden würden? Oder dreimal? Wäre dies eine Möglichkeit, einen Stapel mit vier Blöcken zu erstellen?

Bonus: Einen Regenbogen erstellen

Du kannst zahlreiche Variationen des Programms `blockStack.py` entwerfen. Durch Ändern des Blocktyps kannst du beispielsweise einen Regenbogen oder einen Lavaturm bauen. Wähle verschiedene Blocktypen aus, und schaue, was passiert.

Mission 6: Der Supersprung

In Kapitel 2 hast du gelernt, wie die Position des Spielers geändert werden kann. Nun wollen wir noch einen Schritt weitergehen und schicken den Spieler hoch in die Luft. Mit der Addition geht das ganz einfach. Zuerst musst du herausfinden, wo sich der Spieler gerade befindet. Hierzu verwendest du die Funktion `getTilePos()` so wie in Listing 3.3.

superJump.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

position = mc.player.getTilePos()
x = position.x
y = position.y
z = position.z
```

Listing 3.3: Code zur Ermittlung der Spielerposition

Der Punkt zwischen den Variablen `position` und den Buchstaben `x`, `y` und `z` heißt *Punktnotation*. Die Punktnotation wird von diversen Variablen und Funktionen verwendet, so beispielsweise von allen Funktionen, die du in der Minecraft-Python-API verwendest (also etwa auch von `mc.setTilePos()`). Mehr zur Punktnotation erfährst du in den Kapiteln 11 und 12.

Kennst du die Spielerposition, dann kannst du die Variablen `x`, `y` und `z` auf die aktuellen Koordinaten des Spielers setzen, die durch `position.x`, `position.y` und `position.z` dargestellt werden. Du kannst dann den Spieler an jeden beliebigen Ort teleportieren, indem du relative Wertänderungen für die aktuellen Koordinaten eingibst (Listing 3.4).

superJump.py

```
x = x + 5
mc.player.setTilePos(x, y, z)
```

Listing 3.4: Mit diesem Code schiebst du die x-Position des Spielers um fünf Blöcke nach oben.

Hier habe ich den Spieler um fünf Blöcke entlang der `x`-Achse verschoben. Das ist allerdings nichts Besonderes, denn der Spieler kann ja in Minecraft in horizontaler Richtung jederzeit verschoben werden. Stattdessen wollen wir den Spieler einen Riesensprung machen lassen!

Deine Aufgabe besteht nun darin, den Spieler an der aktuellen Position zehn Blöcke hoch in die Luft springen zu lassen. Dies solltest du mit

dem Code in den Listings 3.3 und 3.4 bewerkstelligen können, wobei du jedoch geringfügige Anpassungen vornehmen musst. Kopiere den Code aus den Listings 3.3 und 3.4 nach IDLE, speichere ihn unter dem Namen *superJump.py*, und lege die *y*-Variable auf ähnliche Weise fest, wie ich es mit der *x*-Variable getan habe. Wenn du das Programm nun ausführst, sollte der Spieler hoch in die Luft springen (Bild 3.2).



Bild 3.2: Supersprung in Aktion!

Subtraktion

Python verfährt mit der Subtraktion ähnlich wie mit der Addition. Nehmen wir an, du möchtest eine Höhle erforschen, eine Spinne greift dich an und du verlierst ein paar Gesundheitspunkte:

```
health = 20  
health = health - 2
```

Der Wert von *health* in dieser Anweisung beträgt nun 18. Wie bei der Addition ermittelt Python das Ergebnis der Operation auf der rechten Seite des Gleichheitszeichens und legt die Variable dann auf diesen Wert fest.

Wir wollen uns dieses Prinzip einmal genauer ansehen.

Mission 7: Die Blöcke unter dir ändern

Hast du schon einmal jemandem in Minecraft eine Falle gestellt? Stell dir nur mal vor, der Boden unter dem Spieler verwandelt sich genau dann in Lava, wenn er es am wenigsten erwartet. Mit Python kannst du diesen Wunsch Wirklichkeit werden lassen. Nutze die Subtraktion, um Blöcke unter der aktuellen Position des Spielers zu platzieren. Du brauchst wirklich nur einige wenige Codezeilen, um einen Block beliebiger Art direkt unter dem Spieler anzuordnen!

Bei dieser Mission tauscht du den Block unter dem Spieler gegen einen Lavablock aus. Hierzu verwendest du `getTilePos()` und `setBlock()`. Sei aber vorsichtig, denn diese Mission ist gefährlich: Wenn du den Spieler nicht schnell genug in eine neue Position bringst, stürzt er in die kochende Lava!

Mit dem Programm in Listing 3.5 erstellst du einen Block an der aktuellen Position des Spielers. Kopiere diesen Code in eine neue Datei in IDLE, und speichere ihn als *blockBelow.py*. Danach nutzt du dein Wissen über den Subtraktionsoperator, um codegesteuert einen Lavablock direkt zu Füßen des Spielers anzuordnen (Bild 3.3).

blockBelow.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z
blockType = 10
mc.setBlock(x, y, z, blockType)
```

Listing 3.5: Mit diesem Code platzierst du einen Block an der aktuellen Position des Spielers.

Du wirst feststellen, dass ich die Variable, die die Position des Spielers speichert, `pos` genannt habe. Ich verwende diese Variable häufig. Der Name ist aussagekräftig, und er ist kürzer und damit schneller zu schreiben als `position`.

Die `y`-Koordinate bestimmt, wie hoch oder niedrig ein Block angeordnet ist. Deine Mission besteht darin, herauszufinden, wie die Variable `y` geändert werden kann, um einen Block unter dem Spieler zu platzieren.



Bild 3.3: Als der Block unter mir sich verwandelte, stürzte ich in die Lava.

Bonus: Blöcke über Blöcke

Du weißt jetzt, wie man Blöcke unter dem Spieler anordnet. Kannst du dir auch denken, wie man einen Block über dem Spieler platziert? Wenn du weißt, wie das geht, dann denke einmal darüber nach, wie man mehrere Blöcke gleichzeitig um den Spieler herum gruppiert. Hast du all das herausbekommen, dann solltest du in der Lage sein, Gebäude um den Spieler herum zu bauen!

Kombiniere dieses Programm mit dem aus Mission 6 (Seite 64). Kannst du dir denken, wie man den Spieler in die Luft springen lassen kann und dann direkt unter ihm einen Block platzieren kann, sodass er nicht in die Tiefe stürzt? Oder du bist richtig gemein und schreibst ein Programm, mit dem der Spieler aus großer Höhe in einen Lavasee fällt.

Mathematische Operatoren in Argumenten verwenden

Wenn du eine Funktion wie `setBlock()` oder `setTilePos()` verwendest, übergibst du dieser Funktion Argumente. Mit diesen Argumenten gibst du die Werte an, die die Funktion bei der Ausführung verwenden soll.

Bis jetzt habe ich den Additions- und den Subtraktionsoperator vorgestellt. Du kannst diese Operatoren in den Klammern eines Funktionsauf-

rufs angeben, um die Werte für die Argumente festzulegen. Wir wollen uns noch einmal die gestapelten Blöcke aus Mission 5 (Seite 61) ansehen. Wir können den Additionsoperator wie in Listing 3.6 gezeigt in den Klammern der `setBlock()`-Funktion verwenden, um die zwei Werte in den Klammern zu addieren – eine zusätzliche Anweisung ist hier nicht erforderlich.

blockStack1.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

x = 6
y = 5
z = 28
blockType = 103
mc.setBlock(x, y, z, blockType)
mc.setBlock(x, y + 1, z, blockType) ❶
```

Listing 3.6: Blockstapelprogramm mit einem Operator in den Argumenten

Listing 3.6 ist beinahe mit dem Blockstapelprogramm identisch, nur verwendet es den Additionsoperator nicht in einer separaten Anweisung, sondern in den Klammern der Funktion `setBlock()`. In der letzten Zeile wird `y + 1` als Argument für die Funktion verwendet ❶. Zwar ist der Wert dieses Arguments 6 ($5 + 1$), doch der Wert der Variablen `y` beträgt weiterhin 5. Über das Argument kannst du die Variable `y` mit einem anderen Wert addieren, ohne ihren aktuellen Wert zu ändern. Dies ist etwa dann nützlich, wenn du `y` an anderer Stelle im Code erneut verwenden möchtest.

Ebenso ist es möglich, zwei Variablen zu addieren und als einzelnes Argument zu verwenden. Listing 3.7 ist mit Listing 3.6 weitgehend identisch, nur bestimmt eine zusätzliche Variable namens `up` den Abstand des neuen Blocks von der `y`-Achse.

blockStack2.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

x = 6
y = 5
z = 28
blockType = 103
up = 1
mc.setBlock(x, y, z, blockType)
mc.setBlock(x, y + up, z, blockType) ❶
```

Listing 3.7: Diese Alternativversion des Stapelprogramms verwendet einen Additionsoperator in den Argumenten.

In der letzten Zeile werden die Variablen `y` und `up` addiert ❶. Wie in Listing 3.6 erhält das zweite Argument der Funktion `setBlock()` den Wert 6. Variablen sind nützlich, denn wenn du etwa den neuen Block zwei Blöcke über der `y`-Achse platzieren möchtest, musst du lediglich deinen Code an dieser Stelle ändern und `up` auf 2 setzen. Die Wirkung aller drei Programmversionen (Listings 3.1/3.2, 3.6 und 3.7) ist in Bild 3.4 zu sehen.



Bild 3.4: Die drei Programmversionen unterscheiden sich zwar, zeigen aber dieselbe Wirkung.

Mission 8: Speed Building

Normalerweise verbringst du deinen ersten Tag in Minecraft damit, einen Unterschlupf zu bauen. Mit dem, was du bis jetzt gelernt hast, kannst du dir aber schon ein einfaches Häuschen zusammenzimmern und deine erste Nacht standesgemäß verbringen. Das Programm in dieser Mission hilft dir, schnell die Wände, die Decke und den Boden eines Gebäudes zu bauen. Anstatt viel Zeit damit zu verbringen, jeden einzelnen Block separat zu platzieren, konstruierst du die Grundstruktur deines Gebäudes mit einigen wenigen Codezeilen.

Du hast `setBlock()` bereits zum Erstellen eines einzelnen Blocks verwendet. Erfreulicherweise hat `setBlock()` aber einen nahen Verwandten namens `setBlocks()`, mit dem sich mehrere Blöcke gleich-

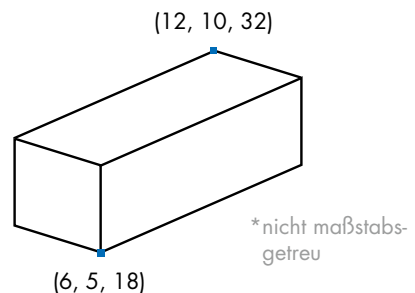


Bild 3.5:
Ein Quader und die Koordinaten zur Angabe seiner Abmessungen

zeitig erstellen lassen, die die Form eines Quaders bilden. Ein *Quader* ist quasi ein dreidimensionales Rechteck. Länge, Breite und Höhe des Quaders können dabei unterschiedlich sein.

Mit der Funktion `setBlocks()` kannst du viele Blöcke in einem großen Bereich erstellen. Um diese Funktion anzuwenden, übergibst du einfach zwei Koordinatensätze und den Blocktyp. Der erste Koordinatensatz gibt an, wo eine Ecke des Quaders platziert werden soll, der zweite beschreibt die Position der gegenüberliegenden Ecke. Bild 3.5 zeigt die Ecken des Quaders, beschriftet mit den jeweiligen Koordinaten.

Bauen wir also einfach mal den Quader aus Bild 3.5. Wie du Listing 3.8 entnehmen kannst, besteht mein Quader aus Bruchstein. Es steht dir aber frei, jeden anderen Blocktyp zu verwenden. Im Grunde genommen kannst du jedes Material mit Ausnahme von Lava, Wasser oder Luft benutzen. (Das könntest du natürlich auch, aber dann sähe dein Haus etwas seltsam aus!)

building.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
pos = mc.player.getPos() ❶
x = pos.x
y = pos.y
z = pos.z
width = 10
height = 5
length = 6
blockType = 4 ❷
air = 0 ❸
mc.setBlocks(x, y, z, x + width, y + height, z + length, blockType)
```

Listing 3.8: Code zum Erstellen eines Quaders

Wenn du den Code liest, wirst du feststellen, dass ich `getPos()` ❶ anstelle von `getTilePos()` verwendet habe. Die Funktion `getPos()` ist im Wesentlichen mit der Funktion `getTilePos()` identisch, gibt die Koordinaten aber in Form von drei Float-Werten (statt dreier Integer) zurück.

Die Breite, die Höhe und die Länge dieses Quaders betragen 10, 5 bzw. 6; und zur Generierung von Bruchsteinen habe ich die Block-ID 4 verwendet ❷. Das fertige Gebäude siehst du in Bild 3.6.

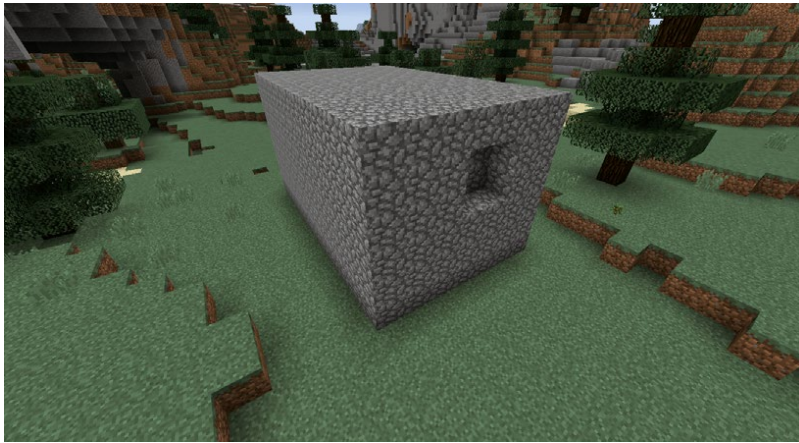


Bild 3.6: Mit dem Programm erstelltes Gebäude

Allerdings gibt es bei diesem Haus ein kleines Problem: Es ist vollkommen massiv! Nach der Ausführung des Programms habe ich ein Loch in die Seite des Hauses gemacht, nur um feststellen zu müssen, dass es auch innen vollständig gefüllt ist. Trotzdem ist ein solcher Quader ein guter Ausgangspunkt. Es ist nun an dir, ihn auszuhöhlen, damit der Spieler sich darin aufhalten kann.

Deine Mission besteht darin, das Programm so abzuändern, dass ein Gebäude mit Wänden, einer Decke und einem Boden an der Position des Spielers gebaut wird. Hierzu musst du innerhalb des massiven Quaders, den wir gerade gebaut haben, einen neuen Quader erstellen – aus Luft! Diese beiden Quader bilden gemeinsam einen leeren Behälter. Den Effekt des fertigen Programms siehst du in Bild 3.7. Auch hier habe ich ein Loch in die Seite des Hauses gemacht und einen Hohlraum im Innern vorgefunden.



Bild 3.7: Wenn dein Programm fertig ist, sollte sich damit ein hohler Quader erstellen lassen. Mit solchen Quadern kannst du ruckzuck ein Gebäude bauen!

Listing 3.8 enthält bereits eine Variable namens `air` ^③, mit der du die Luftblöcke innerhalb des Gebäudes platzieren kannst. Kopiere Listing 3.8 nach IDLE, speichere es als *building.py*, und ändere es so ab, dass ein zweiter Quader aus Luft erzeugt wird. Zum Bauen dieses Luftquaders benötigst du eine zusätzliche `setBlocks()`-Funktion in der letzten Zeile. Der Luftquader muss vollständig von einer Mauer umschlossen sein, die einen Block dick ist – das muss man wissen, um zu erkennen, wie man hier mit Addition und Subtraktion zum Ziel kommt. Nur Geduld: Wenn der erste Versuch misslingt, probiere etwas anderes aus!

Hinweis

Um den Luftquader innerhalb der Mauern anzulegen, kannst du die Additions- und Subtraktionsoperatoren verwenden. Erstelle den Luftquader mit `setBlocks()`, und addiere dann 1 zu den ursprünglichen Argumenten `x`, `y` und `z` hinzu. Dann ziehst du 1 von den Argumenten `x + width`, `y + height`, und `z + length` ab.

Bonus: Dies und das bauen

Du kannst den Code, den du geschrieben hast, jederzeit wiederverwenden, um das betreffende Gebäude zu bauen. Du möchtest aber lieber ein Gebäude einer anderen Größe errichten? Dann musst du herausfinden, wie du die Breite, Höhe und Tiefe des Gebäudes ändern kannst.

Und mit nur wenigen Änderungen kannst du dafür sorgen, dass dein Programm auch viele andere Zwecke erfüllt. Denk beispielsweise mal darüber nach, wie du dir einen Swimmingpool bauen kannst. Kleiner Tipp: Du musst als Blocktyp des inneren Quaders Wasser (Block-ID 8) verwenden und den Deckel des äußeren Quaders entfernen, damit der Spieler schwimmen gehen kann.

Multiplikation

Im Python-Code sieht eine Multiplikation etwas anders aus als das, was du gewohnt sein wirst. Statt einen Punkt zwischen die beiden Zahlen zu setzen, die wir multiplizieren wollen, verwenden wir hier das Sternchen (*). Ansonsten jedoch funktioniert die Multiplikation wie gehabt: Der Ausdruck `2 * 2` ergibt ebenso 4 wie das bekannte $2 \cdot 2$.

Stell dir vor, vor deinem Minecraft-Haus stehen vier Bäume. Und dann verdoppelt sich aus heiterem Himmel ihre Anzahl – einfach so! Diese Berechnung kannst du in Python wie folgt darstellen:

```
trees = 4  
trees = trees * 2
```

Der Wert von `trees` in diesem Beispiel beträgt 8, also 4 multipliziert mit 2.

Division

Die Division wird in Python durch einen Schrägstrich (/) statt durch den Doppelpunkt dargestellt.

Der Divisionsoperator teilt einen Wert durch einen anderen. Hierzu musst du lediglich die Zahl, die du teilen willst, links vom Schrägstrich angeben, während die Zahl, durch die geteilt werden soll, rechts davon steht.

Nehmen wir einmal an, vor deiner Minecraft-Festung stehen acht Skelette. Die Hälfte von ihnen geht dann weg. Um herauszukriegen, wie viele noch übrig sind, müsstest du dann 8 durch 2 teilen. Diese Rechnung ließe sich mithilfe eines Divisionsoperators in Python wie folgt darstellen:

```
skeletons = 8  
skeletons = skeletons / 2
```

Also befinden sich nur noch vier Skelette vor deiner Festung. Schwein gehabt! Wir wollen diese beiden Operatoren jetzt in Minecraft ausprobieren.

Mission 9: Spektakuläre Säulen

Eine Stärke von Variablen besteht darin, dass du den Wert einer einzelnen Variable ändern kannst, und schon ändert sich dieser Wert an jeder Stelle im Programm, an der die Variable verwendet wird. Du kannst dafür sorgen, dass dein Programm vollkommen unterschiedliche Dinge anstellt – einfach nur, indem du eine einzelne Variable mit mathematischen Operatoren wie Multiplikation oder Division änderst.

In dieser Mission wirst du erfahren, wie wir mithilfe von Division und Multiplikation einen sehr hohen und sehr schlanken Turm bauen – eine sogenannte *Säule*.

Bild 3.8 zeigt, wie diese Säule aussehen soll, wenn das Programm abgelaufen ist.



Bild 3.8: Säule aus Steinblöcken

Das Programm verwendet nur eine einzige Variable, mit der die Höhe der Säule festgelegt wird. Mithilfe von Multiplikation und Division kannst du verschiedene Säulenhöhen angeben.

In Listing 3.9 findest du den Anfang eines Programms, das die Säule erstellt. Allerdings verwende ich hier zunächst weder die Variable `height` noch mathematische Operatoren, um die Höhe der einzelnen Teile festzulegen.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z

height = 2
blockType = 1

# Säulenseite, muss der Höhe entsprechen
sideHeight = height
mc.setBlocks(x + 1, y, z + 1, x + 3, y + sideHeight - 1, z + 3,
blockType)

# Säulenspitze, muss der doppelten Höhe entsprechen
pointHeight = 4 ❶
mc.setBlocks(x + 2, y, z + 2, x + 2, y + pointHeight - 1, z + 2,
blockType)

# Säulenbasis, muss der halben Höhe entsprechen
baseHeight = 1 ❷
mc.setBlocks(x, y, z, x + 4, y + baseHeight - 1, z + 4, blockType)
```

spire.py

Listing 3.9: Programm zum Säulenbau

Kopiere Listing 3.9 in eine neue IDLE-Datei, und speichere sie als *spire.py* im Ordner *math*. Dieses Programm baut eine Säule. Allerdings beeinflusst eine Änderung der Variablen `height` und ein erneutes Ausführen des Programms nicht die Höhe aller Teile der Säule.

Um das Programm so zu verbessern, dass alle Teile der Säulenhöhe sich ändern, wenn du die Variable `height` modifizierst, müssen wir die Variablen `pointHeight` ❶ und `baseHeight` ❷ so abändern, dass sie Ausdrücke enthalten, die die Variable `height` sowie Multiplikations- oder Divisionsoperatoren verwenden. `pointHeight` soll der doppelten Größe von `height` entsprechen, `baseHeight` soll halb so groß wie `height` sein. Wenn

beispielsweise die Säulenspitze dreimal so groß wie die Höhe der Säulenseiten sein soll, würde ich den Code so abändern, dass `pointHeight = height * 3` und `baseHeight = height / 2` wäre.

Sobald du diese Änderungen vorgenommen hast, wird, wenn du der Variablen `height` einen neuen Wert zuweist, die Größe aller Bestandteile der Säule geändert.

Andere Teile des Programms brauchst du nicht zu ändern.

Du kannst das Programm testen, indem du den Ursprungswert von `height` änderst und es dann erneut ausführst. Wenn du die Variable `height` auf 3 setzt, wird deine Säule so aussehen wie in Bild 3.9.



Bild 3.9: Durch Modifizieren der Variablen `height` kannst du die Höhe der Säule ganz einfach ändern.

Da du die Werte von `pointHeight` und `baseHeight` mithilfe der Variablen `height` festlegst, lässt sich das Aussehen der Säule problemlos ändern. Du kannst ein bisschen mit diesem Code herumexperimentieren, indem du die Variable `height` auf unterschiedliche Werte setzt. Führe das Programm dann jedes Mal erneut aus, um zu sehen, was passiert.

Potenzrechnung

Mithilfe eines *Exponenten* stellst du dar, dass eine Zahl mit einer bestimmten Häufigkeit mit sich selbst multipliziert werden soll. Beispielsweise ist 3^4 (drei hoch vier) nichts anderes als eine verkürzte Schreibweise von $3 * 3 * 3 * 3$.

In Python wird `**` als Exponentialoperator verwendet. Dabei steht die zu multiplizierende Zahl (die *Basis*) links vom Operator und die Zahl, die die Häufigkeit der Multiplikationsvorgänge angibt (der *Exponent*), steht rechts davon.

Angenommen, du möchtest in Minecraft eine Farm bauen. Zunächst musst du dazu vier Landparzellen bestellen. Jede Parzelle soll 4×4 Blöcke groß sein, damit du viel Weizen anbauen kannst. Mathematisch lässt sich dies als $4 * 4 * 4$ bzw. als 4^3 ausdrücken. Mit dem folgenden Code findest du heraus, wie viel Weizen du anbauen kannst:

```
wheat = 4 ** 3
```

Die Antwort lautet: 64 einzelne Parzellen, denn $4 * 4$ ist 16, und $16 * 4$ ist 64.

Klammern und die Reihenfolge von Rechenoperationen

Wenn du im selben Ausdruck mehrere Rechenoperatoren verwendest, musst du bei ihrer Anordnung sehr vorsichtig vorgehen. Das liegt daran, dass die verschiedenen Operatoren jeweils eine unterschiedliche *Priorität* haben. Verwendest du mehrere Operatoren, dann werden Division und Multiplikation von links nach rechts gesehen zuerst ausgewertet. Erst dann folgen Addition und Subtraktion. Wir wollen uns einmal ansehen, wie der folgende Ausdruck ausgewertet wird:

```
mooshroom = 5 * 2 - 1 + 4 / 2
```

Da Multiplikation und Division jeweils vor Addition und Subtraktion stattfinden, beginnt Python links zunächst mit der Multiplikation von 5 und 2 (Ergebnis: 10) und teilt danach 4 durch 2 (Ergebnis: 2). Wir erhalten also $10 - 1 + 2$. Jetzt springt Python wieder an den Anfang der Rechnung, zieht zunächst 1 von 10 ab und addiert dann 2 hinzu. So erhält die Variable `mooshroom` den Wert 11.

Du kannst die Reihenfolge der Operationen aber auch mithilfe von Klammern steuern. Bei Ausdrücken mit Operatoren in Klammern werden diese zuerst ausgewertet, erst dann folgt alles andere. Wir wollen uns einmal ansehen, wie sich die Reihenfolge der Verarbeitung von Operationen

durch Klammern ändern lässt. Hier siehst du zunächst eine Anweisung, die keine Klammern verwendet:

```
zombiePigmen = 6 * 3 - 2
```

Bei dieser Notierung erhält `zombiePigmen` am Ende den Wert 16, denn 6 mal 3 ist 18, und 18 minus 2 ist 16. Durch die Verwendung von Klammern lässt sich das Ergebnis jedoch beeinflussen:

```
zombiePigmen = 6 * (3 - 2)
```

`zombiePigmen` hat nun den Wert 6! Statt den Ausdruck in der normalen Reihenfolge abzarbeiten, subtrahiert Python zunächst 2 von 3 (Ergebnis: 1) und multipliziert dann 6 mit 1. Und 6 mal 1 gibt bekanntlich 6.

Wenn du Berechnungen in einer bestimmten Reihenfolge ausführen möchtest, sage Python mit Klammern, was zuerst erledigt werden muss. So kannst du Python noch besser steuern.

Praktische Rechentricks

In den nächsten Abschnitten zeige ich dir, wie du deine Python-Programme mithilfe von zwei weiteren mathematischen Fähigkeiten noch mehr nach vorne bringst. Danach werden wir das bislang Gelernte in einer weiteren Mission kombinieren.

Kurzformen für Operatoren

Es kommt häufiger vor, dass eine Variable mit einem Rechenoperator bearbeitet und das Ergebnis dann in derselben Variable gespeichert werden soll. Nehmen wir etwa an, du möchtest eine Schafherde um fünf Tiere erweitern:

```
sheep = 6  
sheep = sheep + 5
```

Allerdings wird es irgendwann nervig, immer `sheep = sheep + 5` oder Ähnliches schreiben zu müssen. Es geht aber auch kürzer. Python bietet *verkürzte Operatoren*, mit denen du genau das tun kannst: einen Variab-

lenwert mit einem Rechenoperator bearbeiten und das Ergebnis dann in derselben Variable speichern. Es gibt vier verkürzte Operatoren:

- Addition (+=)
- Subtraktion (-=)
- Multiplikation (*=)
- Division (/=)

Unser Beispiel mit der Schafherde könnte man wie folgt verkürzt schreiben:

```
sheep = 6  
sheep += 5
```

Am Ende hat sheep den Wert 11 – genau wie oben.

Mehr Spaß mit Zufallszahlen

Mit Zufallszahlen kannst du deine Programme um Lustiges und Geheimnisvolles erweitern. Am Ende weißt du nämlich nie genau, wie es ausgeht! Viele echte Brettspiele basieren auf der Verwendung von Zufallszahlen. Denk nur mal an die vielen verschiedenen Spiele, bei denen du mit einem Würfel bestimmt hast, um wie viele Felder du vorrücken darfst. Das Würfeln ist ein klassisches Beispiel für die Verwendung von Zufallsfunktionen.

Python kann Zufallszahlen ohne Probleme für dich erzeugen. Wir wollen das einmal mit einem virtuellen sechsseitigen Würfel simulieren: Die Zahl, die wir erzeugen, also *generieren* wollen, soll zwischen 1 und 6 liegen:

```
import random ❶  
diceValue = random.randint(1, 6) ❷
```

Möchtest du Zufallszahlen erzeugen, dann musst du am Anfang eines Programms in jedem Fall `import random` ❶ einschließen. Die Funktion `randint()` ❷ generiert einen ganzzahligen Wert, den das Programm wie jeden anderen Zahlenwert verwenden kann. Die Zahlen, die du als Argumente in die Klammer setzt, weisen `randint()` an, Werte zwischen der ersten und der zweiten number zu generieren. In unserem Beispiel hat die erzeugte Zahl einen Wert von 1, 2, 3, 4, 5 oder 6.

Mit `randint()` kannst du dem Wert einer Variablen eine Zufallszahl hinzufügen und sogar negative Zahlen erzeugen. Wir wollen sehen, wie das geht.

```
import random
score = 0
score += random.randint(0, 99)
points = random.randint(-99, 99)
```

Mit dieser Anweisung kann Python Zahlen zwischen 0 und 99 zu `score` hinzufügen. `points` hingegen kann Zahlen zwischen -99 und +99 enthalten – beachte das negative Argument.

Mission 10: Supersprung 2.0

In der letzten Mission dieses Kapitels bringst du den Spieler dazu, einen Supersprung in unbekannte Gefilde durchzuführen. Hierzu speicherst du die aktuelle Position des Spielers und addierst dann zu allen drei Koordinaten (`x`, `y` und `z`) jeweils einen Zufallswert hinzu. Um die Werte von `x` und `z` zu ändern, solltest du Zufallszahlen zwischen -10 und 10 angeben, bei `y` verwendest du Werte zwischen 0 und 10.

Kopiere zunächst Listing 3.10 in eine neue Datei in IDLE, und speichere sie unter dem Namen *randomJump.py*.

randomJump.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
import random

pos = mc.player.getPos()
x = pos.x
y = pos.y
z = pos.z

x = x + random.randint(-10, 10) ❶
mc.player.setPos(x, y, z)
```

Listing 3.10: Das noch unvollständige Zufallssprungprogramm

In diesem Programm fehlt der Code zur Generierung des Zufallswerts für die Variablen `y` und `z`. Es ist deine Aufgabe, diesen zu ergänzen. Wenn du damit fertig bist, kann der Spieler irgendwohin springen – genau so, wie ich es in Bild 3.10 getan habe. Nutze den Zufall, um zu neuen und aufregenden Orten zu gelangen!

Zurzeit verwendet der Code noch keine verkürzten Operatoren, um die Variablenwerte zu ändern. Versuche einmal, den Additionsausdruck bei ❶ so zu verändern, dass er einen verkürzten Operator verwendet.



Bild 3.10: Irgendwohin gesprungen, oben auf einem Baum gelandet. Wo zum Teufel bin ich hier eigentlich?

Bonus: Blöcke kreuz und quer teleportieren

Du kannst den Grad der Zufälligkeit beim Programm *randomJump.py* sogar noch steigern! Nachdem der Spieler zu einer zufällig ausgewählten Position gesprungen ist, platzierst du einen ebenso zufällig ausgewählten Block unter ihm. Du könntest auch das Teleportationsprogramm aus Mission 1 (*teleport.py* auf Seite 44) so umbauen, dass der Spieler jedes Mal an einer anderen zufälligen Position landet. Solltest du dich versehentlich irgendwohin teleportieren, wo du nicht mehr rauskommst, dann kannst du jederzeit wieder *teleport.py* starten, um dich an eine Stelle zu teleportieren, die sicher ist.

Was du gelernt hast

In diesem Kapitel hast du gelernt, wie man Mathematik in Python einsetzt. Du wirst Addition, Subtraktion, Multiplikation und Division häufig in den Python-Programmen, die im weiteren Verlauf dieses Buchs vorgestellt werden, und auch zukünftig in deinen selbst geschriebenen Programmen einsetzen. Außerdem hast du erfahren, wie man Zufallszahlen erzeugt, und nebenbei einige sehr praktische Programme für Minecraft kennengelernt. Alle Hochachtung!

In Kapitel 4 stoßen wir auf Strings in Python. Dieser Datentyp kann Buchstaben, Symbole und Zahlen aufnehmen. Strings sind in Minecraft extrem nützlich, denn damit kannst du Meldungen an den Minecraft-Chat übergeben. Außerdem werden wir untersuchen, wie man diesen Chat mithilfe von Strings manipuliert und weitere coole Dinge anstellt.

Kapitel 4

Chatten mit Strings

In den Kapiteln 2 und 3 hast du mit Integern und Floats gearbeitet – zwei numerischen Datentypen. In diesem Kapitel lernst du einen weiteren Datentyp kennen: *Strings*. Strings setzt du zur Bearbeitung von Buchstaben und Symbolen, aber auch von Zahlen ein.

Mit Strings kannst du den Benutzern deiner Programme Daten anzeigen. Das ist ein wichtiger Aspekt der Programmierung. Python benutzt Strings, um Daten auf dem Bildschirm *auszugeben* und dem Benutzer so Informationen anzuzeigen und mit ihm zu kommunizieren.

In Minecraft kannst du Strings an verschiedenen Stellen einsetzen. Du kannst etwa Meldungen in den Chat einstellen, um so im Multiplayer-Modus mit anderen Spielern zu kommunizieren. Während das Posten von Meldungen in den normalen Minecraft-Versionen zum Standardfunktionsumfang gehört, ist es bei der Raspberry-Pi-Version ein verborgenes Feature. Dank der Wunder der Programmierung kannst du jedoch auch auf dem Raspberry Pi chatten! Du kannst geheimes Wissen mit deinen Freunden teilen oder damit angeben, was für tolle Schätze du schon gesammelt hast.

Neben den Strings werden wir uns in diesem Kapitel auch den Funktionen widmen. Als aufmerksamer Leser wirst du bemerkt haben, dass wir einige Funktionen bereits eingesetzt haben. `setPos()`, `setTilePos()`, `setBlock()`, `setBlocks()`, `getPos()` und `getTilePos()` sind *Funktionen*, d. h. wiederverwendbare Codeabschnitte, die das Erledigen von Aufgaben erheblich vereinfachen. Eigentlich ganz cool, oder?

In den Missionen dieses Kapitels wirst du auf dem Wissen aufbauen, das du bislang erworben hast. Du wirst mithilfe von Strings Meldungen im Minecraft-Chat posten und das Eingeben von Daten üben, um auf deren Basis Objekte in deiner Minecraft-Welt zu erstellen.

Was sind Strings?

Der Datentyp *String* kann beliebige Texte aufnehmen – von einem einzelnen Buchstaben oder Symbol (z. B. "a" oder "&") bis hin zu umfangreichen Textzeichenfolgen. Buchstaben, Ziffern oder Symbole in einem String werden *Zeichen* genannt. Wenn du Buchstaben, Symbole, Wörter, Sätze oder eine Kombination aus all diesen Elementen in deinem Programm verwenden möchtest, dann benutzt du Strings.

Du kannst in Strings Buchstaben, Zahlen und Symbole speichern. Alle Strings stehen in Anführungszeichen. Das hier wäre ein Beispiel für einen String:

```
"Achtung! Hinter dir steht ein Zombie!"
```

Auch das hier ist ein String:

```
'Willkommen in meinem geheimen Reich!'
```

Hast du den kleinen Unterschied bei der Notierung dieser Beispiele bemerkt? Beim Schreiben eines Strings kannst du sowohl einfache Anführungszeichen (') als auch doppelte (") verwenden. Achte jedoch darauf, die Anführungszeichen nicht gemischt einzusetzen! Wenn ein String mit einem einfachen Anführungszeichen beginnt, muss er auch mit einem einfachen Anführungszeichen enden. Beginnt er mit einem doppelten Anführungszeichen, dann muss er mit einem doppelten Anführungszeichen enden. Es gibt gute Gründe dafür, beide Optionen in Python zu implementieren. Wenn du etwa einen Apostroph in deinem String verwenden möchtest, kannst du dies problemlos tun, wenn du den gesamten String in doppelte Anführungszeichen setzt.

Die Funktion print()

Das Anzeigen von Text und anderen Informationen ist für die Benutzerinteraktion sehr wichtig. Der Benutzer weiß andernfalls nämlich nicht, was in deinen Programmen so vor sich geht. Die Informationen, die du dem Benutzer anzeigst, heißen *Ausgaben*. Um Daten auf dem Bildschirm des Benutzers auszugeben, verwendest du die Funktion `print()`.

Wenn du eine Meldung ausgeben möchtest, übergibst du der Funktion `print()` als Argument einen String:

```
>>> print("String")
```

So weist du Python an, dem Benutzer das Wort String anzuzeigen. Möchtest du also das Wort Schokolade an die Python-Shell ausgeben, dann schreibst du:

```
>>> print("Schokolade")
```

Und die Ausgabe lautet folglich:

```
Schokolade
```

Du kannst mit `print()` aber auch Variablenwerte ausgeben. Angenommen, du hast eine Variable namens `name`, in der ein Name als String gespeichert ist. Wenn du diesen nun auf dem Bildschirm ausgeben möchtest, machst du zunächst Folgendes:

```
>>> name = "Steve the Miner"
```

Nach dem Speichern des Strings "Steve the Miner" in `name` musst du nur noch `print(name)` eingeben, um die folgende Ausgabe anzuzeigen:

```
>>> print(name)
Steve the Miner
```

Nachdem wir uns nun mit den Grundlagen zu Strings vertraut gemacht haben, bearbeite bitte folgende Mission, um deine Minecraft-Welt zu begrüßen.

Mission 11: Hallo, Minecraft-Welt!

Wenn du mit anderen Spielern in Minecraft Pi chatten möchtest, bietet die Minecraft-Python-API die Möglichkeit, Nachrichten mithilfe der Funktion `postToChat()` zu versenden. Die Funktion `postToChat()` nimmt einen String als Argument entgegen und postet diesen im Minecraft-Chatfenster. Mit Listing 4.1 beispielsweise wird "Hello, Minecraft World" im Chat ausgegeben.

message.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
mc.postToChat("Hello, Minecraft World")
```

Listing 4.1: Mit Python eine Begrüßung im Minecraft-Chat versenden

Du weißt ja bereits, dass ein Argument eine Information ist, die du an eine Funktion übergeben musst, wenn sie aufgerufen wird. Die Funktion benötigt diese Information, um ihre Aufgabe zu erledigen. Im letzten Kapitel beispielsweise mussten wir Zahlen an unsere Funktionen übergeben, um zu definieren, was die jeweilige Funktion tun soll. Hier nun braucht `postToChat()` einen String wie etwa "Hello, Minecraft World".

`postToChat()` ähnelt der Funktion `print()`. Beide Funktionen können Strings auf dem Bildschirm anzeigen, und beide können eine Variable entgegennehmen, die einen String als Argument speichert. Der Unterschied besteht darin, dass `print()` Strings an die Python-Shell ausgibt, während `postToChat()` die Ausgabe im Minecraft-Chat anzeigt.

Kopiere den Code aus Listing 4.1, und speichere ihn als *message.py* in einem neuen Ordner namens *strings*. Wenn du das Programm ausführst, sollte der übergebene String wie in Bild 4.1 im Chat erscheinen.



Bild 4.1: Meine Nachricht steht im Chat.

Probiere nun einmal, einen anderen String an `postToChat()` zu übergeben, um so eine andere Nachricht im Chat anzuzeigen.

Bonus: Wo bist du?

Mit der Funktion `mc.postToChat()` kannst du alle möglichen Informationen im Chat posten. Versuche doch einmal, die aktuelle X-Position des Spielers oder den Typ des Blocks anzuzeigen, auf dem der Spieler steht. Zur Erinnerung: Die Funktion `mc.player.getTilePos()` ruft die aktuelle Position des Spielers ab; den Blocktyp an einer bestimmten Position ermittelst du mit der Funktion `mc.getBlock()`.

Die Funktion input()

Bislang haben wir alle unsere Variablen direkt im Programm festgelegt oder – wie man auch sagt – in das Programm *einkodiert*. Wenn du den Variablenwert ändern wolltest, musstest du das Programm selbst bearbeiten. Wäre es da nicht praktischer, wenn man diese Variablen bei laufendem Programm ändern oder womöglich sogar *Benutzereingaben* des Spielers annehmen und verarbeiten könnte?

Eine solche Form der Interaktivität lässt sich mit der Funktion `input()` tatsächlich realisieren. Die Funktion gibt zunächst einen String an die Konsole aus, um dem Benutzer mitzuteilen, was für eine Information er eingeben soll. Dann wartet sie, bis der Benutzer mit einer Eingabe reagiert hat. Gib einmal folgenden Code in die Python-Shell ein, und sieh dir an, was passiert:

```
>>> input("Wie heißt du? ")
```

Nun siehst du den String, den du an `input()` übergeben hast, und kannst die Frage mit einer Eingabe beantworten.

```
Wie heißt du?
```

Wenn du dann eine Antwort eingibst, sollte etwas in dieser Art angezeigt werden:

```
Wie heißt du? Craig  
'Craig'
```

Nett, oder? Möchtest du diese Eingabe allerdings in deinem Programm verwenden, dann musst du sie in einer Variablen speichern. Anders als die Python-Shell geben Programme, die im Texteditor erstellt werden, die Ergebnisse von Anweisungen nicht automatisch aus. Beispiel:

```
>>> name = input("Wie heißt du? ")
Wie heißt du? Craig
```

Wenn du diesmal deinen Namen eingibst und die **Eingabetaste** drückst, stellst du fest, dass das Programm deine Eingabe nicht automatisch anzeigt. Um die gespeicherte Eingabe zu sehen, übergibst du die Variable `name` als Argument an die Funktion `print()`:

```
>>> print(name)
Craig
```

Toll! Du hast es geschafft, deine Eingabe in einer Variablen zu speichern und deren Wert auszugeben. Das ist extrem praktisch, denn so kannst du Eingaben des Benutzers an beliebiger Stelle in deinem Programm weiterverwenden. Diese Technik möchten wir nun verwenden, um Chatnachrichten in das Minecraft-Chatfenster zu schreiben.

Mission 12: Chatnachrichten selbst gemacht

Unser Chat soll interaktiver werden! Mit der Python-Shell kannst du eine Nachricht in den Minecraft-Chat schreiben – das haben wir bereits in Mission 11 gesehen. In dieser Mission nun schreiben wir ein geringfügig abgeändertes Programm, das den String, den du im Chat posten möchtest, in einer Variablen namens `message` speichert.

Listing 4.2 stellt hierfür den Ausgangspunkt dar. Kopiere es in eine neue IDLE-Datei, und speichere sie als *messageInput.py* im Ordner *strings*.

messageInput
.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
message = "This is the default message." ❶
mc.postToChat(message) ❷
```

Listing 4.2: So werden Strings im Minecraft-Chat ausgegeben.

Das Programm speichert die Nachricht, die du an den Chat ausgeben möchtest, in der Variablen `message` ❶. In diesem Fall ist die Variable ein String mit dem Inhalt "This is the default message." Dann übergibt

das Programm die Variable `message` an die Funktion `postToChat()` ❷, die den String an den Minecraft-Chat ausgibt.

In diesem Programm ist der String fest einkodiert, d. h., er bleibt bei jeder Ausführung des Programms gleich. Du brauchst aber nur eine einzige Änderung vorzunehmen, und schon wird ausgegeben, was der Benutzer schreibt! Das bedeutet, dass du bei jeder Ausführung des Programms deine ganz eigenen persönlichen Worte formulieren kannst. So erstellst du ein vollkommen personalisiertes Chatprogramm.

Damit das Programm Eingaben entgegennimmt, ersetzt du den String "This is the default message." ❶ durch die Funktion `input()`. Vergiss nicht, der Funktion `input()` ein Argument zu übergeben, z. B. "Enter your message: " (dt. »Gib deine Nachricht ein«). Dieses Argument musst du wie üblich in die Klammern des Funktionsaufrufs von `input()` setzen. Hast du alle Änderungen vorgenommen, dann führe das Programm aus. Nun sollte ein Prompt mit der Aufforderung "Enter your message: " in der Python-Shell angezeigt werden. Gib deine Nachricht ein, und drücke die **Eingabetaste**. Die Nachricht wird in der Shell und im Minecraft-Chat angezeigt (Bild 4.2).

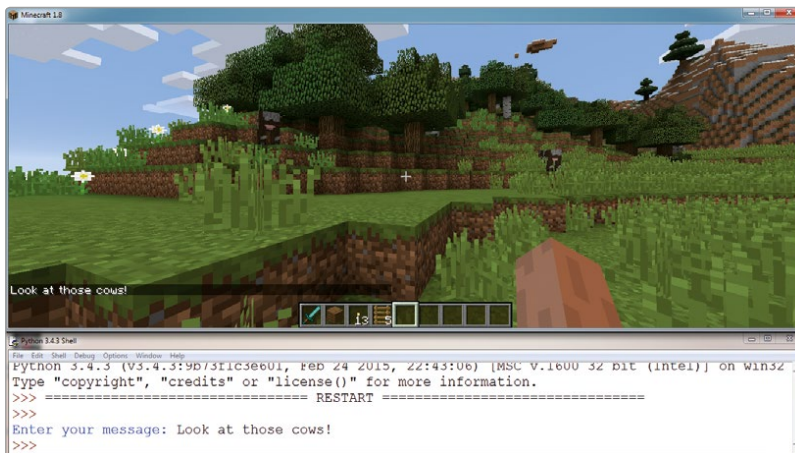


Bild 4.2: Wenn ich eine Meldung in die IDLE-Shell eingebe, wird sie im Minecraft-Chat gepostet.

Nun musst du die Meldung, die im Chat angezeigt werden soll, nicht mehr fest ins Programm einkodieren, sondern kannst sie vorher eingeben. Die Eingabefunktion macht das Chatten so viel einfacher, nicht wahr?

Bonus: Noch mehr Meldungen

Das Programm fragt in dieser Form erst einmal eine Meldung ab. Hast du eine Idee, wie man es dazu bringen könnte, zunächst eine Meldung zu erfragen, dann einige Sekunden zu warten (mit der `sleep()`-Funktion) und danach eine zweite Meldung anzufordern?

Strings verknüpfen

Häufig müssen Strings in kombinierter Form ausgegeben werden. Dies bezeichnet man als Verknüpfen oder *Verketteten* von Strings – und es ist in Python kein Problem!

In Kapitel 3 haben wir den Additionsoperator (+) zum Addieren von Zahlen verwendet; du kannst ihn aber auch zur Stringverkettung einsetzen. Beispiel:

```
firstName = "Charles"
lastName = "Christopher"
print(firstName + lastName)
```

`print()` gibt in diesem Fall "CharlesChristopher" aus. Wenn der Vor- und der Nachname durch ein Leerzeichen getrennt werden sollen, kannst du dieses Zeichen wie folgt mit dem Additionsoperator hinzufügen:

```
print(firstName + " " + lastName)
```

Python bietet häufig mehrere Wege zum selben Resultat. In diesem Fall könntest du zum Beispiel auch ein Komma setzen, um den gewünschten Effekt zu erzielen:

```
print(firstName, lastName)
```

Beide Anweisungen geben "Charles Christopher" aus. Du kannst auch fest einkodierte Strings mit Variablen verknüpfen, sofern es sich bei diesen auch um Strings handelt. Du schreibst den Wert einfach wie jeden anderen String auf:

```
print("Er heißt " + firstName + " " + lastName)
```

Hier wird "Er heißt Charles Christopher" ausgegeben.

Das Zusammenfügen von Textzeichenfolgen ist ganz nützlich, aber manchmal kann es auch sein, dass man Strings mit einem anderen Datentyp verknüpfen möchte – zum Beispiel mit einem Integer. Python unterstützt das direkte Zusammenfügen eines Strings mit einem Integer nicht. Deswegen musst du vorher dafür sorgen, dass der Integer in einen String konvertiert wird. Wir wollen das ausprobieren.

Zahlen in Strings umwandeln

Die Konvertierung eines Variablentyps in einen anderen ist erfreulich unkompliziert. Stell dir beispielsweise vor, du speicherst die Anzahl der goldenen Äpfel, die du besitzt, in einer Integer-Variablen namens `myGoldenApples`. Jetzt möchtest du vor deinen Freunden ein bisschen angeben, weil du doch so viele goldene Äpfel hast, denn sie sind selten, und außerdem gibst du ganz gern ein bisschen an. Jetzt könntest du eine Meldung wie "Mein gar nicht so geheimer Bestand an goldenen Äpfeln: ", gefolgt von dem Wert ausgeben, der in `myGoldenApples` gespeichert ist. Damit du aber den Wert von `myGoldenApples` in die ausgegebene Meldung einbinden kannst, musst du Python anweisen, den Integer `myGoldenApples` in einen String zu konvertieren.

Die Funktion `str()` konvertiert Datentypen, die keine Strings sind – also etwa Integers oder Floats –, in Strings. Um einen Wert in einen String zu konvertieren, setzt du diesen in die Klammern des Funktionsaufrufs `str()`.

Kehren wir noch einmal zu unserem Beispiel mit den goldenen Äpfeln zurück. Angenommen, du hast `myGoldenApples` auf 2 festgelegt und möchtest, dass Python diesen Wert als String statt als Integer behandelt. Deine Meldung würdest du wie folgt ausgeben:

```
print("Mein gar nicht so geheimer Bestand an goldenen Äpfeln: " +  
str(myGoldenApples))
```

Diese Anweisung gibt den String "Mein gar nicht so geheimer Bestand an goldenen Äpfeln: 2" aus.

Du kannst auch Floats in Strings umwandeln. Angenommen, du hast einen halben goldenen Apfel gegessen, und `myGoldenApples` beläuft sich jetzt nur noch auf 1,5 Äpfel. `str(myGoldenApples)` funktioniert für den Wert 1,5 genauso gut wie weiter oben bei 2: Der Wert wird in einen

String umgewandelt und kann dann in deine Meldung eingebunden werden.

Wenn du Zahlen in Strings konvertiert hast, kannst du sie beliebig miteinander verketten. Lass uns ein bisschen mit Zahlen herumspielen, indem wir daraus Strings machen und diese verketten.

Integer und Floats verketten

Wenn du zwei Datenelemente miteinander verketten möchtest, müssen beide Strings sein. Zudem wird das Pluszeichen sowohl für die Addition als auch für die Verkettung verwendet, d. h., wenn du Integer, Floats und andere Zahlen verkettest, wird Python stattdessen versuchen, sie zu addieren. Um sie mithilfe der Verkettung zusammenzufügen, musst du numerische Werte zuvor in Strings umwandeln.

Verwende die Methode `str()`, um zwei Zahlen zu verketten, anstatt sie zu addieren:

```
print(str(19) + str(84))
```

Da du Python angewiesen hast, die Zahlen 19 und 84 als Strings zu behandeln und sie zu verketten, gibt die Anweisung anstelle von 103 (der Summe aus 19 und 84) den Wert 1984 aus.

Du kannst die Verkettung beliebig oft in einer Anweisung verwenden. Beispiel:

```
print("The year is " + str(19) + str(84))
```

Diese Codezeile gibt `The year is 1984` aus.

Üben wir nun ein bisschen die Verkettung, indem wir deine neuen Fertigkeiten in der nächsten Mission auf die Probe stellen.

Mission 13: Benutzernamen zum Chat hinzufügen

Wenn du mit mehr als nur einem Mitspieler spielst, ist möglicherweise schwer zu erkennen, welcher der Mitspieler gerade eine Nachricht im Minecraft-Chat gepostet hat. Die naheliegende Lösung besteht darin, am Anfang der jeweiligen Nachricht den betreffenden Benutzernamen anzuzeigen. Bei dieser Mission änderst du das Programm aus Mission 12 so ab,

dass vor allen Nachrichten, die im Chat gepostet werden, der Benutzername eingefügt wird.

Öffne *messageInput.py* in IDLE, und speichere die Datei unter dem neuen Namen *userChat.py* im Ordner *strings*. Danach ergänzt du den Code, indem zunächst der Benutzername abgefragt wird, bevor die zugehörige Nachricht eingegeben werden kann. Die im Chat gepostete Nachricht sollte das folgende Format aufweisen: "Anna: Ich brauche TNT." Zur Bearbeitung dieser Mission musst du die Stringverkettung einsetzen.

Suche im Programm nach der folgenden Codezeile:

```
message = input("Enter your message: ")
```

In der obigen Zeile musst du eine weitere Variable namens *username* hinzufügen und ihren Wert auf `input("Please enter a username: ")` setzen. Nach dem Hinzufügen der Variablen *username* suchst du die folgende Zeile:

```
mc.postToChat(message)
```

Füge mithilfe der Verkettungsmethode die Strings *username* und *message* innerhalb der Funktion `postToChat()` zusammen. Setze `:` zwischen die beiden Strings, damit in der Ausgabe zwischen der Variablen *username* und der Variablen *message* ein Doppelpunkt und ein Leerzeichen angezeigt werden. Bild 4.3 zeigt, wie die Ausgabe des fertigen Programms aussehen sollte.

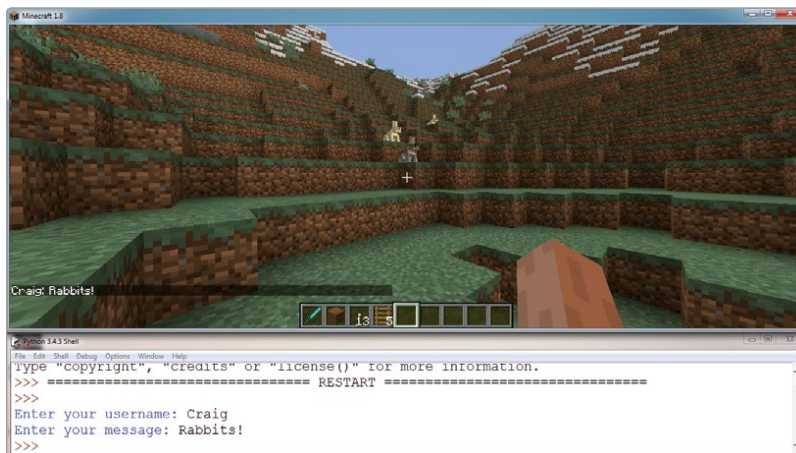


Bild 4.3: Anzeige des Benutzernamens beim Chatten

Speichere das Programm, und führe es aus. In der Python-Shell wirst du nun aufgefordert, deinen Benutzernamen einzugeben. Gib deinen Namen ein, und drücke auf die **Eingabetaste**. Als Nächstes gibst du deine Nachricht ein. Danach sollten Benutzername und Nachricht im Minecraft-Chat angezeigt werden.

Bonus: Benutzer »Namenlos«

Was passiert, wenn du den Benutzernamen leer lässt und die **Eingabetaste** drückst? Und warum, denkst du, ist das so?

Strings mit `int()` in Integer konvertieren

Ähnlich wie die Funktion `str()`, mit der Nicht-Strings in Strings konvertiert werden können, wandelt die Funktion `int()` Nicht-Integer-Daten in Integer um.

`int()` ist vor allem in Kombination mit der Funktion `input()` sehr nützlich. Die Funktion `input()` gibt die Benutzereingaben nämlich als String zurück, und das ist dann problematisch, wenn die Eingabe in Rechenoperationen verwendet werden soll. In solchen Fällen muss man sie also zunächst mit `int()` in einen Integer umwandeln.

Es funktioniert wie folgt: Angenommen, du hättest einer Variablen namens `cansOfTunaPerCat` bereits einen Integer-Wert zugewiesen und bräuchtest ein Programm, das dir sagt, wie viel Thunfisch abhängig von der Anzahl der Katzen im Besitz des Benutzers gegessen wird. Hier siehst du ein Beispiel für ein solches Programm:

```
cansOfTunaPerCat = 4
cats = input("Wieviele Katzen hast du? ")
cats = int(cats)
dailyTunaEaten = cats * cansOfTunaPerCat
```

Das Gleiche kannst du auch in einer einzigen Zeile erledigen, indem du die Funktionen miteinander verschachtelst:

```
cats = int(input("Wieviele Katzen hast du? "))
dailyTunaEaten = cats * cansOfTunaPerCat
```

Nachdem du nun weißt, wie man Eingaben in Integer umwandelt, solltest du auch in der Lage sein, Blocktypen in Minecraft-Programme einzugeben.

Mission 14: Block per Eingabe erstellen

In Minecraft gibt es unglaublich viele Blocktypen. Du kannst eine Menge davon im Kreativmodus auswählen, aber längst nicht alle. Über die Python-API für Minecraft jedoch erhältst du Zugriff auf sämtliche Blocktypen und kannst diese mithilfe der Funktion `setBlocks()` auswählen.

Wir haben die Funktion `setBlocks()` bereits kennengelernt, aber damals musstest du den Blocktyp noch in deinen Programmcode einkodieren, und er konnte während der Ausführung deines Programms nicht geändert werden. Aber jetzt kennen wir ja die Funktion `input()`! Wenn du ein Programm schreibst, das Eingaben entgegennimmt, kannst du bei jeder Programmausführung festlegen, welchen Blocktyp du erstellen willst. So kannst du bei der ersten Ausführung des Programms etwa einen Block aus Wolle erstellen und bei der zweiten einen aus Eisenerz.

Bei dieser Mission schreibst du ein Programm, das dem Benutzer die Entscheidung überlässt, was für einen Block er erstellen möchte. Kopiere Listing 4.3 in eine neue IDLE-Datei, und speichere sie als *blockInput.py* im Ordner *strings*.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
blockType = # Hier die input()-Funktion einfügen ❶
pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z
mc.setBlock(x, y, z, blockType)
```

blockInput.py

Listing 4.3: Mit diesem Code erstellst du einen Block an der Spielerposition.

Dieses Programm platziert einen Block an der aktuellen Position des Spielers. Versuche es so abzuändern, dass die Variable `blockType` mit der `input()`-Funktion festgelegt wird ❶. Ich empfehle dir, eine Frage oder einen aussagekräftigen Text einzufügen, damit der Benutzer weiß, dass er eine Block-ID und nicht irgendeinen anderen Text eingeben soll. Tust du dies nicht, dann zeigt IDLE nur eine leere Zeile mit einem blinkenden Cursor an und wartet, bis der Benutzer irgendetwas eingibt. Du solltest

dem Benutzer jedoch klarmachen, dass das Programm von ihm eine Zahleneingabe erwartet.

Zur Erinnerung: `input()` gibt deine Eingabe als String zurück; damit der Wert weiterverarbeitet werden kann, musst du ihn mithilfe der Funktion `int()` in einen Integer konvertieren. Der Ausdruck zur Erfassung der Blocktypeingabe sollte etwa so aussehen:

```
blockType = int(input("Bitte gib einen Blocktypen an: "))
```

Speichere dein geändertes Programm, führe es aus, und gib eine beliebige Blocknummer ein. Bild 4.4 zeigt das Resultat dieses Programms.

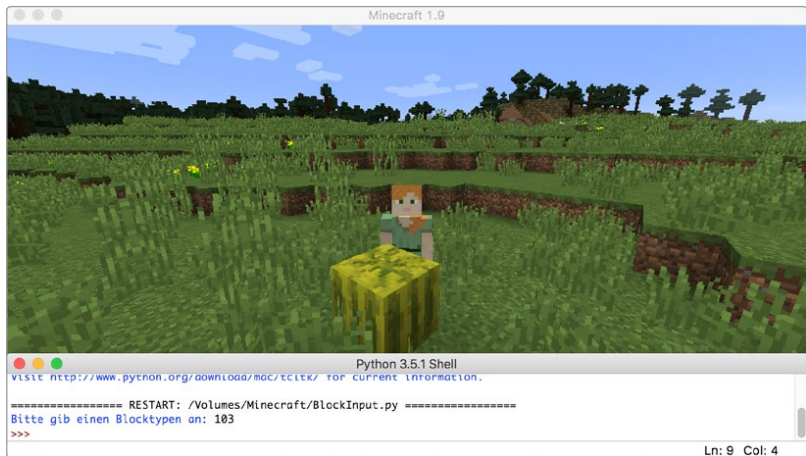


Bild 4.4: Blöcke bauen nach Belieben

Bonus: Noch mehr interaktive Games

Du kannst im Programm beliebig viele Eingaben verwenden. Zurzeit erstellt das Programm den Block an der aktuellen Position des Spielers. Versuche doch einmal herauszufinden, wie du die Variablen `x`, `y` und `z` über Eingaben festlegen kannst. Und wenn du richtig wagemutig bist, teleportierst du den Spieler einmal eingabegesteuert zu bestimmten Koordinaten.

Fehlermeldung? Lieber nicht!

Python verwendet das Prinzip der *Ausnahmebehandlung* (engl. »Exception Handling«), um sicherzustellen, dass dein Programm in der Lage ist, Fehler abzufangen und die Ausführung auch im Fehlerfall fortzusetzen. So stellt die Ausnahmebehandlung eine Möglichkeit dar, fehlerhafte Benutzereingaben zu verwalten.

Angenommen, dein Programm hat einen Integer angefordert, aber der Benutzer hat einen String eingegeben. Normalerweise würde das Programm eine Fehlermeldung anzeigen – man spricht auch vom *Auslösen einer Ausnahme* – und die Ausführung dann abbrechen.

Dank Ausnahmebehandlung kannst du den Fehler aber selbst abfangen: Das Programm läuft seelenruhig weiter, und dem Benutzer wird eine aussagekräftige Fehlermeldung angezeigt (z. B. "Bitte geben Sie einen Zahlenwert ein."). So kann der Benutzer seinen Fehler wieder gutmachen, ohne das Programm neu starten zu müssen.

Eine try-except-Anweisung ist ein Beispiel für ein Tool, das zum Abfangen von Fehlern verwendet werden kann. Sie eignet sich besonders gut, um sinnvolles Feedback zu geben, wenn die Eingabe des Benutzers fehlerhaft war, und sie verhindert einen Programmabbruch im Fehlerfall.

Die Anweisung besteht aus zwei Teilen: dem try-Teil und dem except-Teil. Der try-Teil ist der Codeabschnitt, der ausgeführt werden soll, wenn keine Fehler auftreten. Dieser Code könnte etwa eine Eingabe entgegennehmen oder einen String ausgeben. Der except-Teil der Anweisung wird nur ausgeführt, wenn im try-Teil ein Fehler auftritt.

Angenommen, wir haben einen Codeabschnitt, der abfragt, wie viele Sonnenbrillen du besitzt (ich beispielsweise besitze drei):

```
try:
    noOfSunglasses = int(input("Wieviele Sonnenbrillen besitzt du? ")) ❶
except:
    print("Ungültige Eingabe. Bitte gib eine Zahl ein.") ❷
```

Das Programm benötigt eine Zahl. Gibst du Buchstaben oder Symbole ein, dann wird "Ungültige Eingabe. Bitte gib eine Zahl ein." ausgegeben. Der Fehler tritt auf, weil die Funktion `int()` nur solche Strings umwandeln kann, die ausschließlich aus ganzen Zahlen bestehen ❶. Gibst du eine Zahl ein, dann funktioniert der Code einwandfrei; andern-

falls jedoch – z. B. bei Eingabe von viele Sonnenbrillen – würde ein Fehler in der `int()`-Funktion auftreten.

Ist dir übrigens an dem Code etwas aufgefallen? Hier verwenden wir erstmals eine Anweisung, bei der eine *Einrückung* erforderlich ist. Hierunter versteht man das Eingeben mehrerer Leerzeichen *vor* dem eigentlichen Text. Bei der Behandlung der `if`-Anweisungen in Kapitel 6 und der `for`-Schleifen in den Kapiteln 7 und 9 werde ich etwas mehr über Einrückungen sagen. An dieser Stelle möchte ich dich lediglich bitten, deinen Code exakt so einzugeben, wie er im Buch abgebildet ist.

Normalerweise zeigt Python bei Auftreten eines Fehlers eine kryptische Meldung an, aus der nicht hervorgeht, wie das Problem zu beheben ist. Mit `try-except` kannst du jedoch verhindern, dass die Python-Fehlermeldungen dem Benutzer bei der Eingabe von Daten des falschen Typs angezeigt werden; stattdessen gibst du selbst dem Benutzer einfache und nützliche Anweisungen, damit er weiß, was zu tun ist. Manchmal drückt der Benutzer auch einfach nur die **Eingabetaste**, statt tatsächlich eine Eingabe vorzunehmen. Im Normalfall tritt hierbei ein Fehler auf, aber dank des Codes in der `try-except`-Anweisung ❶ wird stattdessen der String "Ungültige Eingabe. Bitte gib eine Zahl ein." ausgegeben ❷.

Du kannst praktisch beliebigen Code in eine `try-except`-Anweisung setzen und `try-except`-Anweisungen sogar verschachteln. Probiere das doch in der nächsten Mission einmal aus!

Mission 15: Zahlen, nichts als Zahlen

Erinnerst du dich an das Programm aus Mission 14? Dort hast du einen Integer-Wert eingegeben, und das Programm hat erwartungsgemäß einen Block erstellt. Hättest du jedoch einen String eingegeben, dann hätte das Programm die Ausführung mit einer Fehlermeldung abgebrochen (Bild 4.5).



Bild 4.5: cake ist doch keine Zahl!

Wenn du mit Python gut vertraut bist, ist eine solche Fehlermeldung durchaus aussagekräftig. Was aber, wenn jemand, der überhaupt keine Ahnung von Python hat, hier nun einen String statt einer ganzen Zahl eingibt? Ihm würde eine vollkommen unverständliche Fehlermeldung angezeigt. Deine Mission besteht darin, eine leicht verständliche Meldung zu schreiben und mithilfe der Fehlerbehandlung anzuzeigen.

Öffne das Programm *blockInput.py*, das du in Mission 14 erstellt hast. Speichere es dann als *blockInputFix.py* im Ordner *strings*.

Du wirst dieses Programm jetzt so ändern, dass es beim Abfragen einer Block-ID eine try-except-Anweisung verwendet. Suche hierzu nach der letzten Codezeile im Programm. Diese sollte wie folgt aussehen:

```
mc.setBlock(x, y, z, blockType)
```

Füge direkt über dieser Zeile eine try-Anweisung hinzu, und setze vier Leerzeichen vor die Funktion `mc.setBlock()`. Als Nächstes fügst du in der Zeile über `setBlock()` folgenden Code hinzu, um eine Benutzereingabe abzurufen: `blockType = int(input("Bitte gib einen Blocktypen an: "))`.

Dann schreibst du in der Zeile unter der `setBlock()`-Funktion eine except-Anweisung. In dieser except-Anweisung fügst du eine Codezeile ein, die eine Meldung im Minecraft-Chat postet. Diese Meldung besagt, dass der Benutzer eine Zahl eingeben muss (Beispiel: "Du hast keine

Zahl eingegeben! Bitte gib beim nächsten Mal eine Zahl ein.". Der geänderte Code sollte nun wie folgt aussehen. Beachte den Einzug mit den vier Leerzeichen zu Beginn der Zeilen ❶ und ❷:

```
try:
    blockType = int(input("Bitte gib einen Blocktypen an: ")) ❶
    mc.setBlock(x, y, z, blockType)
except:
    mc.postToChat("Du hast keine Zahl eingegeben. Bitte gib beim ↵
    nächsten Mal eine Zahl ein.") ❷
```

Die `int()`-Funktion geht davon aus, dass sie den vom Benutzer eingegebenen Wert in einen Integer umwandeln soll ❶. Weil wir die `try-except`-Anweisung zum Programm hinzugefügt haben, wird, wenn die Benutzereingabe kein Zahlenwert ist (sondern Buchstaben oder Symbole enthält), ein Fehler auftreten. Anstatt nur die normale Python-Fehlermeldung anzuzeigen, gibt das Programm einen String im Chat aus, mit dem der Benutzer aufgefordert wird, beim nächsten Mal eine Zahl einzugeben ❷. Du solltest die Chatmeldung vielleicht ein bisschen entschärfen!

Wenn du eine etwas höflichere Meldung eingegeben hast, speichere die Datei `blockInputFix.py` und führe sie aus, um deine Arbeit zu begutachten. Das Ergebnis sollte etwa so aussehen wie in Bild 4.6.

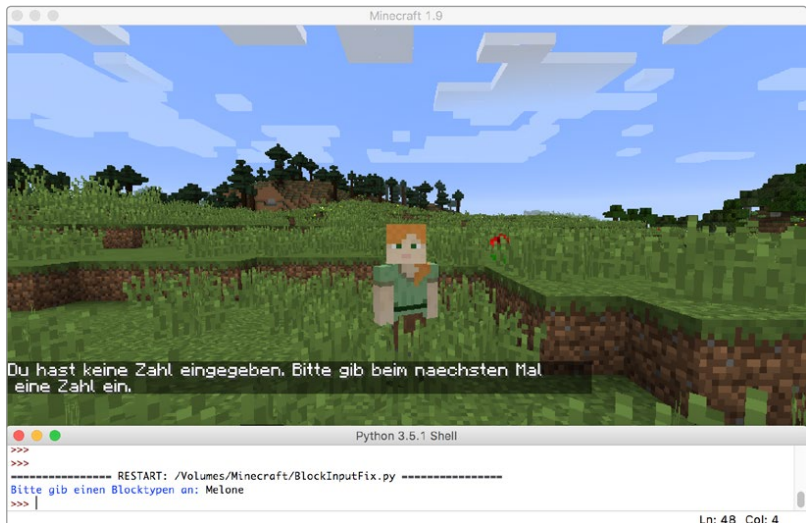


Bild 4.6: Aussagekräftige Fehlermeldung im Chat

Mission 16: 100 Meter in 10 Sekunden

In der letzten Mission dieses Kapitels ist alles zusammengefasst, was du bislang gelernt hast: die Variablen aus Kapitel 2, die Rechenoperatoren aus Kapitel 3 und das Posten von Meldungen im Chat. Du sollst dir nun einen Chronisten bauen: Das Programm, das du gleich erstellen wirst, ermittelt, welche Strecke der Spieler innerhalb von zehn Sekunden zurücklegt, und zeigt das Ergebnis im Chat an.

Denke daran, dass du dein Programm mit dem folgenden Code dazu bringen kannst, eine bestimmte Zeitspanne zu warten. Diese Zeit wird in Sekunden angegeben.

```
import time      # Diese Zeile recht weit oben im Programm einfügen
time.sleep(30)   # Hierbei wartet das Programm 30 Sekunden
```

Verwende diese `sleep()`-Funktion, und gib den folgenden Code ein, um mit dem neuen Programm zu beginnen:

```
import time
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

pos1 = mc.player.getTilePos()  ❶
x1 = pos1.x
y1 = pos1.y
z1 = pos1.z

time.sleep(10)

pos2 = mc.player.getTilePos()  ❷
x2 = pos2.x
y2 = pos2.y
z2 = pos2.z

# Differenz zwischen der Start- und der Endposition vergleichen
xDistance = x2 - x1  ❸
yDistance =
zDistance =

# Ergebnisse im Chat ausgeben
mc.postToChat("")  ❹
```

sprint.py

Wir wollen diesen Code einmal analysieren. Das Programm ruft die Startposition des Spielers ab ❶, wartet dann 10 Sekunden und ruft nachfolgend bei ❷ die Endposition des Spielers ab. Zum Abschluss des Pro-

gramms musst du die Differenz zwischen der Start- und der Endposition ermitteln. Hierzu legst du die Werte der Variablen `yDistance` und `zDistance` fest, die bei ③ beginnen. Um dir ein bisschen zu helfen, habe ich den Wert der Variablen `xDistance` eingefügt, der $x_2 - x_1$ betragen muss. Die Werte von `yDistance` und `zDistance` lassen sich auf ähnliche Weise bestimmen, nur verwendest du andere Variablen als `x1` und `x2`.

In der letzten Zeile gibst du dann die Ergebnisse in den Minecraft-Chat aus ④. Diese Ergebnisse sollten im folgenden Format angezeigt werden: "The player has moved x: 10, y: 6, and z: -3". Hierzu verwendest du Strings, das Verkettungsprinzip und die Werte der Variablen `xDistance`, `yDistance` und `zDistance`.

Speichere das Programm im Ordner *strings* unter dem Namen *sprint.py*, und führe es aus. Bild 4.7 stellt das Ergebnis dar.



Bild 4.7: Die zurückgelegte Distanz wird nach Abschluss des Programms angezeigt.

Wenn du das Programm zum Laufen gebracht hast, aber es schwierig findest, schnell genug zwischen der Befehlszeile und Minecraft umzuschalten, füge einen dreisekündigen Countdown vor Schritt 2 ein. Auch diesen Countdown kannst du im Chat anzeigen.

Bonus: Auf direktem Wege

An diesem Punkt zeigt das Programm die Distanz für jede Dimension (x, y und z) separat an. Wie würdest du die zurückgelegte Entfernung als einzelnen Integer-Wert berechnen, der die direkt zurückgelegte Strecke angibt? Hinweis: Hierzu benötigst du Kenntnisse aus der Trigonometrie und insbesondere den Satz des Pythagoras. Keine Sorge, falls du nicht genau weißt, wie du das anstellen sollst: In Mission 40 (Seite 185) wirst du ein ähnliches Programm kennenlernen, mit dem sich die direkt zurückgelegte Strecke als Integer angeben lässt.

Was du gelernt hast

Herzlichen Glückwunsch! In diesem Kapitel hast du wirklich viel gelernt. Du hast Strings erstellt, sie mit `print()`-Anweisungen angezeigt und sie miteinander verkettet. Du hast Programme geschrieben, die Benutzereingaben entgegennehmen, den Datentyp von Werten geändert und dich in die Ausnahmebehandlung eingearbeitet. Dabei hast du deine Python-Kenntnisse auch schon in der Praxis angewendet, um deinen Minecraft-Chat etwas aufzupeppen.

In Kapitel 5 wirst du erfahren, wie du den Programmablauf steuerst und deinen Programmen beibringst, Entscheidungen zu treffen.

Kapitel 5

Stimmt's oder nicht? Boolesche Variablen

Ja-Nein-Fragen stellt man sich eigentlich die ganze Zeit: Regnet's? Müsste ich mal wieder zum Friseur? Sobald man weiß, ob die Antwort ja oder nein lautet, kann man entscheiden, was als Nächstes zu tun ist: Regenschirm rauskramen, zum Hörer greifen und einen Termin beim Hairstylisten des Vertrauens machen usw.

In all diesen Situationen hängt das, was man tut, von der Beantwortung der gestellten Frage ab. Und auch in der Programmierung stellen von Antworten abhängige Entscheidungen ein wichtiges Konzept dar. In diesem Kapitel erfährst du, wie man in Python Fragen stellt.

In der Programmierung geht es bei solchen Fragen meist um das Vergleichen von Werten. Sind zwei Werte gleich? Oder ist einer der Werte größer oder kleiner als der andere? Diese Ja-Nein-Fragen heißen *Bedingungen*, und die Antwort heißt nicht *ja* oder *nein*, sondern `True` (wahr) oder `False` (falsch). Nehmen wir an, du würdest gerne wissen, ob du mehr Goldblöcke hast als dein Kumpel. Oder anders formuliert: »Ist mein Goldvorrat größer als seiner?« Um aus dieser Frage eine Bedingung zu machen, die Python verstehen kann, müssen wir sie als Aussage formulieren, die wahr oder falsch sein kann: »Mein Goldvorrat ist größer als der meines Kumpels.«

Überprüfen zu können, ob eine Bedingung wahr oder falsch ist, ist in Python derart nützlich, dass es hierfür einen speziellen Datentyp gibt, der lediglich die Werte `True` und `False` speichern kann. Wir haben ja bereits einige Datentypen kennengelernt: Integer, Floats und Strings. Der Datentyp, der `True` und `False` speichert, ist der *boolesche* Datentyp (eng. »Boolean«). Boolesche Variablen können nur die Werte `True` und `False` annehmen. Wenn du in Python Fragen stellst, ist das Ergebnis folglich entweder `True` oder `False`. Wenn eine Bedingung wahr oder falsch ist, sprechen Programmierer davon, dass sie als `True` oder `False` *ausgewertet* wird.

Du wirst in diesem Kapitel boolesche Variablen, Vergleichsoperatoren und logische Operatoren verwenden, um verschiedene Bedingungen anhand von Werten zu prüfen. Danach bist du bereit, Kapitel 6 anzugehen, wo wir auf der Grundlage von Antworten auf Fragen Entscheidungen über den weiteren Ablauf eines Programms treffen werden.

Grundlagen zu booleschen Variablen

Eine boolesche Variable ähnelt ein wenig einem Lichtschalter, denn sie ist entweder `True` (an) oder `False` (aus). In Python kannst du eine boolesche Variable wie folgt deklarieren, um darzustellen, dass das Licht an ist:

```
light = True
```

Hiermit weist du der Variablen `light` den Wert `True` zu. Um das Licht nun auszuschalten, kannst du der Variablen stattdessen den Wert `False` zuweisen:

```
light = False
```

Du musst dabei den ersten Buchstaben von `True` oder `False` jeweils großschreiben, sonst erkennt Python das Wort nicht als booleschen Wert und löst eine Ausnahme aus, anstatt deine Berechnung auszuwerten.

In der nächsten Mission wirst du mithilfe boolescher Variablen verhindern, dass der Spieler Blöcke in der Minecraft-Welt zerschlägt.

Mission 17: Schluss jetzt mit der Blockzerstörung!

In Minecraft kannst du Blöcke problemlos zerschlagen. Das ist gut, wenn du Ressourcen schürfen möchtest, aber schlecht, wenn du Ewigkeiten am Bau einer wirklich coolen Struktur gesessen hast und sie dann versehentlich zerschlägst und so ruinierst. In dieser Mission machst du deine Minecraft-Welt haltbar.

Mit `setting("world_immutable", True)` kannst du Blöcke *unzerstörbar* machen, d. h., sie können nicht verändert werden. Die `setting()`-Zeile ist eine Funktion – du kennst bereits Funktionen wie `setTilePos()` und `setPos()`. Listing 5.1 zeigt, wie man eine Welt unzerstörbar macht.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

mc.setting("world_immutable", True)
```

immutableOn
.py

Listing 5.1: Code gegen ungewollte Zerstörung

Die Funktion `setting()` bietet verschiedene Optionen, die du zur Aktivierung auf `True` setzen kannst. Eine dieser Optionen heißt `"world_immutable"`. Um eine `setting()`-Option zu aktivieren, schreibst du in die Klammern der Funktion `True` hinter den Namen der betreffenden Einstellung.

Kopiere Listing 5.1 nach IDLE, und speichere den Code unter dem Namen *immutableOn.py* in einem neuen Ordner namens *booleans*. Wenn du ihn ausgeführt und keinen Fehler gemacht hast, solltest du die meisten Blöcke nicht mehr zerschlagen können (Bild 5.1). Aber was ist, wenn du jetzt *doch* wieder Blöcke kaputt machen willst? In diesem Fall kopierst du das Programm in eine neue Datei und änderst diese so ab, dass das Bearbeiten von Blöcken wieder möglich ist. (Kleiner Tipp: Verwende eine boolesche Variable.) Speichere die neue Datei dann als *immutableOff.py* im Ordner *booleans*.



Bild 5.1: Ich kann machen, was ich will – der Block geht nicht kaputt!

Boolesche Werte verketteten

Wie Integer und Floats müssen boolesche Variablen vor der Verkettung in Strings konvertiert werden. Um etwa boolesche Variablen mit der Funktion `print()` auszugeben, verkettest du sie mit Strings. Hierzu verwendest du die bereits bekannte Funktion `str()`:

```
>>> agree = True
>>> print("I agree: " + str(agree))
I agree: True
```

Die Variable `agree` speichert einen booleschen Wert. Sie wird in der zweiten Zeile mit `str(agree)` in einen String umgewandelt, dann mit dem String `"I agree: "` verkettet und schließlich ausgegeben.

Vergleichsoperatoren

Im Vergleichen von Werten bist du sicher echt gut. Du weißt, dass 5 größer als 2 ist, dass 8 und 8 die gleiche Zahl darstellen und dass dies bei 6 und 12 nicht der Fall ist. Auch ein Computer ist gut im Vergleichen von Werten – man muss ihm nur genau sagen, was für einen Vergleich man braucht. Hierzu musst du ein Symbol eingeben, das als *Vergleichsoperator* bezeichnet wird. Soll beispielsweise überprüft werden, ob ein Wert größer als der andere ist oder ob er kleiner ist?

Vergleichsoperatoren in Python erlauben dir das Vergleichen von Daten. Python verwendet dabei sechs Operatoren:

- ist gleich (==)
- ist ungleich (!=)
- ist kleiner als (<)
- ist kleiner als oder gleich (<=)
- ist größer als (>)
- ist größer als oder gleich (>=)

Jeder Vergleichsoperator gibt einen booleschen Wert (True oder False) zurück, der angibt, ob die Bedingung erfüllt ist. Wir wollen uns diese Vergleichsoperatoren und die Möglichkeiten ihres Einsatzes einmal genauer ansehen.

Ist gleich

Wenn du herausfinden möchtest, ob zwei Werte gleich sind, kannst du den Gleichheitsoperator (==) verwenden. Sind die Werte gleich, dann gibt der Vergleich den booleschen Wert True zurück. Andernfalls – wenn also die Werte unterschiedlich sind – gibt der Vergleich False zurück.

Wir wollen beispielshalber zwei Variablen deklarieren und diese dann mit dem Gleichheitsoperator vergleichen.

```
>>> length = 2
>>> width = 2
>>> length == width
True
```

Das Ergebnis ist True, weil die Werte der Variablen length und width identisch sind.

Wären sie hingegen unterschiedlich, dann lautete das Ergebnis False:

```
>>> length = 4
>>> width = 1
>>> length == width
False
```

Du kannst den Gleichheitsoperator für alle Variablentypen verwenden: Strings, Integer, Floats und boolesche Variablen.

Oben erkennst du, dass ich für den Vergleich von `length` und `width` die Zeichen `==` anstelle von `=` verwendet habe, denn damit würde eine Variable festgelegt. Python verwendet den Operator `==`, um zwischen einem Vergleich (der Frage, ob zwei Werte identisch sind) und der Festlegung einer Variablen zu unterscheiden. Diesen Unterschied musst du auf dem Schirm haben, um Bugs in deinem Code zu vermeiden. Aber keine Sorge, sogar ich mache ab und zu noch den Fehler, `=` statt `==` zu verwenden!

Mission 18: Schwimme ich?

Als Nächstes wirst du mithilfe der Vergleichsoperatoren ein Programm schreiben, mit dem sich feststellen lässt, ob du im Wasser stehst oder nicht. Die Ergebnisse werden dann im Minecraft-Chat gepostet.

Um den Blocktyp an einer bestimmten Position zu ermitteln, verwendest du die Funktion `getBlock()`. Diese Funktion nimmt die Koordinaten in Form von drei Argumenten entgegen und gibt den Blocktyp als Integer zurück. Beispiel:

```
blockType = mc.getBlock(10, 18, 13)
```

Hier habe ich das Ergebnis von `mc.getBlock(10, 18, 13)` in einer Variablen namens `blockType` gespeichert. Wenn sich an den Koordinaten (10, 18, 13) eine Melone befindet (Blockwert 103), enthält die Variable `blockType` den Wert 103.

Nun wollen wir die Funktion `getBlock()` einmal sinnvoll einsetzen. Mit Listing 5.2 wird überprüft, ob der Spieler auf trockenem Boden steht oder nicht.

swimming.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

pos = mc.player.getPos()
x = pos.x
y = pos.y
z = pos.z

blockType = mc.getBlock(x, y, z)
mc.postToChat(blockType == 0)
```

Listing 5.2: Mit diesem Code wird der Typ des Blocks abgefragt, auf dem sich die Füße des Spielers befinden.

Hier rufe ich die drei Koordinaten der Spielerposition ab und übergebe diese als Argumente an `getBlock()`. Das Ergebnis von `mc.getBlock(x, y, z)` speichere ich in `blockType`. Mit dem Ausdruck `blockType == 0` wird festgestellt, ob der Block Luft ist. Falls ja, weißt du, dass du an irgendeiner Stelle in deiner Minecraft-Welt stehst – der Ausdruck ist also wahr, und folglich wird `True` an den Chat gepostet. Besteht der Block nicht aus Luft, dann wird `False` in den Chat geschrieben. In diesem Fall befindest du dich wahlweise im Wasser oder im Treibsand – und drohst zu ertrinken bzw. zu ersticken.

Kopiere Listing 5.2, und speichere es als *swimming.py* im Verzeichnis *booleans*. Ändere dann den Code so ab, dass überprüft wird, ob der Spieler im Wasser steht (Blocktyp 9), und führe ihn aus.

Stell dich einmal versuchsweise ins Wasser, und starte das Programm. Überprüfe dann, ob im Chat die Meldung `True` angezeigt wird, wenn der Spieler sich tatsächlich im Wasser befindet. Andernfalls sollte im Chat `False` erscheinen.

Die Programmausgabe sollte so aussehen wie in Bild 5.2.

Hinweis

Zu diesem Zeitpunkt kannst du das Programm nicht fortlaufend ausführen, sondern musst es jedes Mal neu starten, wenn du den Block unter dem Spieler überprüfen möchtest. Dies gilt auch für alle anderen Missionen in diesem Kapitel.



Bild 5.2: Natürlich sehe ich, dass ich im Wasser stehe. Aber Python bestätigt mir das freundlicherweise auch.

Bonus: Ich kann fliegen!

Mit nur wenigen Änderungen in deinem Code kannst du überprüfen, ob der Block *unter* dir Luft ist. Das würde bedeuten, dass du entweder fliegst oder springst. Wie könnte man das bewerkstelligen?

Ist ungleich

Der Ungleichheitsoperator ist das Gegenteil des oben beschriebenen Gleichheitsoperators. Statt zu überprüfen, ob zwei Werte gleich sind, kannst du mit ihm feststellen, ob sie sich unterscheiden. Wenn die beiden Werte verschieden sind, wird der Vergleich als `True` ausgewertet; sind sie gleich, dann lautet das Ergebnis `False`.

Angenommen, du möchtest überprüfen, ob ein Objekt rechteckig, aber nicht quadratisch ist. Da bei einem Rechteck, das kein Quadrat ist, die Länge und die Breite unterschiedlich sind, könntest du einen Vergleich schreiben, der überprüft, ob Länge und Breite nicht gleich sind:

```
>>> width = 3
>>> length = 2
>>> width != length
True
```

Der Ausdruck `width != length` fragt, ob die Werte für `width` und `length` unterschiedlich sind.

Das Ergebnis dieses Vergleichs ist `True`, weil die Variablen `width` und `length` unterschiedliche Werte haben.

Sind die Werte hingegen gleich, dann gibt der Vergleich `False` zurück:

```
>>> width = 3
>>> length = 3
>>> width != length
False
```

Wie der Gleichheitsoperator funktioniert auch der Ungleichheitsoperator bei Strings, Integern, Floats und booleschen Variablen.

Mission 19: Und worin stehe ich eigentlich?

Angenommen, du möchtest überprüfen, ob du in etwas anderem als Luft stehst – also Wasser, Lava, Erde, Kies oder irgendeinem anderen Blocktyp. In Mission 18 hast du getestet, ob der Block an deiner aktuellen Position aus Luft war, und du hast ferner gelernt, wie du feststellst, ob du im Wasser stehst. Du könntest das Programm nun x-mal kopieren und jeweils anpassen, um auf Lava, Erde, Kies usw. zu prüfen. Aber das wäre ja langweilig. Stattdessen verwenden wir den Ungleichheitsoperator, um zu prüfen, ob du unter der Erde, im Sand oder am Grunde des Ozeans stehst oder gerade in Lava gebraten wirst.

Öffne das Programm aus Mission 18 (*swimming.py*), und speichere es als *notAir.py* im Ordner *booleans*. Lösche die letzte Programmzeile, und ersetze sie durch Listing 5.3.

```
notAir = blockType == 0 ❶
mc.postToChat("The player is not standing in air: " + str(notAir))
```

notAir.py

Listing 5.3: Änderungen am Schwimmprogramm

Die letzte Codezeile zeigt dir, ob du nicht in der Luft stehst. Das Vergleichsergebnis wird in der Variablen `notAir` ❶ gespeichert. Wird der Vergleich als `True` ausgewertet, dann hat `notAir` den Wert `True`; wird der Vergleich hingegen als `False` ausgewertet, hat auch die Variable `notAir` den Wert `False`.

Allerdings stimmt der Vergleich in der ersten Zeile ❶ nicht ganz. Gegenwärtig wird mit dem Gleichheitsoperator (==) geprüft, ob `blockType` Luft ist. Wir wollen aber ermitteln, ob `blockType` gerade nicht Luft ist, und verwenden dazu den Ungleichheitsoperator (`!=`). Ändere die erste Zeile so ab, dass anstelle des Gleichheits- der Ungleichheitsoperator verwendet wird. So wird festgestellt, ob der Block an der aktuellen Position des Spielers nicht Luft ist.

Wenn du das Programm ausführst, überprüfe die Resultate unabhängig vom Blocktyp, d. h. für Situationen, in denen du dich unter Wasser, in Lava, im Kies oder im Sand befindest oder unter die Erde teleportiert wurdest. Bild 5.3 zeigt die Meldung, die gepostet wird, wenn die Bedingung wahr (True) ist.



Bild 5.3: Schwimme doch ganz gemütlich eine Runde im Wasser. Ist ja nicht Luft.

Größer als/kleiner als

Wenn Du herausfinden möchtest, ob ein Wert größer als ein anderer ist, verwendest du den Größer-als-Vergleichsoperator. Dieser Vergleichsoperator gibt `True` zurück, wenn der Wert zu seiner Linken größer ist als der auf der rechten Seite. Ist der Wert links hingegen kleiner als der Wert auf der rechten Seite oder mit diesem identisch, wird `False` zurückgegeben.

Angenommen, wir haben eine Lore, in der maximal 99 Obsidianblöcke Platz finden. Solange diese Höchstkapazität unserer Lore größer ist als

die Anzahl der darin vorhandenen Blöcke, können weitere Blöcke aufgeladen werden:

```
>>> limit = 100
>>> obsidian = 99
>>> limit > obsidian
True
```

Brillant! Unsere Lore kann jede beliebige Anzahl von Obsidianblöcken transportieren, die kleiner als 100 ist. 99 ist kleiner als 100, d. h., `limit > obsidian` wird als `True` ausgewertet. Aber was geschieht, wenn jemand einen weiteren Obsidianblock auf den Haufen legt?

```
>>> limit = 100
>>> obsidian = 100
>>> canLift = limit > obsidian
False
```

Oh nein! Wir haben das Limit erreicht. Jetzt lautet das Resultat `False`, denn 100 ist nicht größer als 100, sondern dasselbe. Deswegen kann kein Obsidianblock mehr dazukommen.

Der Kleiner-als-Vergleichsoperator funktioniert auf gleiche Weise.

Der Fahrer eines Lasters (engl. »van«), der unter einer Brücke durchfahren soll, muss wissen, ob der Laster niedrig genug ist, um darunter durchzupassen:

```
>>> vanHeight = 388
>>> bridgeHeight = 400
>>> vanHeight < bridgeHeight
True
```

In diesem Fall passt der Laster, da er niedriger ist als die Brücke, denn 388 ist kleiner als 400. Im weiteren Verlauf der Fahrt kommt der Laster zu einer anderen Brücke, die jedoch zu niedrig ist:

```
>>> vanHeight = 388
>>> bridgeHeight = 350
>>> vanHeight < bridgeHeight
False
```

Weil 388 nicht kleiner als 350 ist, lautet das Ergebnis `False`.

Größer oder gleich und kleiner oder gleich

Wie der Größer-als-Operator bestimmt auch der Größer-oder-gleich-Operator, ob ein Wert größer ist als ein anderer. Im Gegensatz zum Größer-als-Operator erhält dieser Vergleichsoperator jedoch den Wert `True`, wenn die Werte gleich sind.

Nehmen wir an, ich verschenke Aufkleber an alle Besucher einer mitreisenden Präsentation, die ich gehalten habe. Dazu muss ich erst einmal nachsehen, ob genügend Aufkleber für jeden vorhanden sind:

```
>>> stickers = 30
>>> people = 30
>>> stickers >= people
True
```

Ich habe also genügend Aufkleber: 30 ist dasselbe wie 30, d. h., `stickers >= people` wird als `True` ausgewertet. Jetzt hat aber einer meiner Freunde die Aufkleber gesehen und möchte selbst einen haben. Also bräuchte ich nun Aufkleber für 31 Leute:

```
>>> stickers = 30
>>> people = 31
>>> stickers >= people
False
```

30 ist nicht größer als oder gleich 31 – ich habe nicht mehr genug Aufkleber. Mein Kumpel bekommt also keinen.

Wir sind jetzt an einer Stelle angekommen, an der wir praktisch jeden Vergleich durchführen können. Probiere doch jetzt auch einmal den Kleiner-oder-gleich-Operator (`<=`) in IDLE aus, um zu sehen, wie er funktioniert.

Hinweis

Die Größer-als-, Größer-oder-gleich-, Kleiner-als- und Kleiner-oder-gleich-Operatoren funktionieren bei Integern, Floats und booleschen Werten, nicht aber bei Strings.

Mission 20: Stehe ich eigentlich auf dem Boden?

Die y-Koordinate zeigt, wie hoch sich ein Spieler in Minecraft befindet. Auch Blöcke werden mit Koordinaten gespeichert – erst dadurch wird es möglich, Blocktypen an bestimmten Positionen mit `getBlock()` abzufragen und Blöcke mit `setBlocks()` zu erstellen.

Um den höchsten Block in Minecraft zu ermitteln, kannst du die Funktion `getHeight()` verwenden. Diese Funktion nimmt eine x- und eine z-Koordinate entgegen und gibt die y-Koordinate des höchsten Blocks an der betreffenden Position zurück. Listing 5.4 zeigt, wie es geht.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z
highestBlockY = mc.getHeight(x, z)
mc.postToChat(highestBlockY)
```

*aboveGround
.py*

Listing 5.4: Mit diesem Code ermittelst du die y-Koordinate des höchsten Blocks an der aktuellen Position des Spielers.

Das Programm ermittelt die aktuelle Position des Spielers, ruft die y-Koordinate für den höchsten Block an dieser Position ab und schreibt den Wert in den Minecraft-Chat.

Durch Kombination dieses Programms mit einem Größer-oder-gleich-Operator kannst du überprüfen, ob sich der Spieler auf der Erdoberfläche befindet. Und genau das werden wir jetzt tun.

Kopiere das Programm aus Listing 5.4, und speichere es als *aboveGround.py*. Ändere das Programm so ab, dass überprüft wird, ob die y-Koordinate des Spielers größer ist als die Variable `highestBlockY`. Danach fügst du ein wenig Code hinzu, um das Ergebnis im folgenden Format in Chat zu posten: "The player is above the ground: True/False".

Tipp

Du weißt schon, dass du das Ergebnis eines Vergleichs in einer Variablen speichern kannst. Wenn ich beispielsweise überprüfen möchte, ob `y` größer oder gleich 10 ist, und das Ergebnis in einer Variablen namens `highEnough` speichern möchte, verwende ich folgende Anweisung: `highEnough = y >= 10`.

Führe das Programm aus, nachdem du alle erforderlichen Änderungen vorgenommen hast. Die Ausgabe für das Ergebnis `False` siehst du in Bild 5.4.

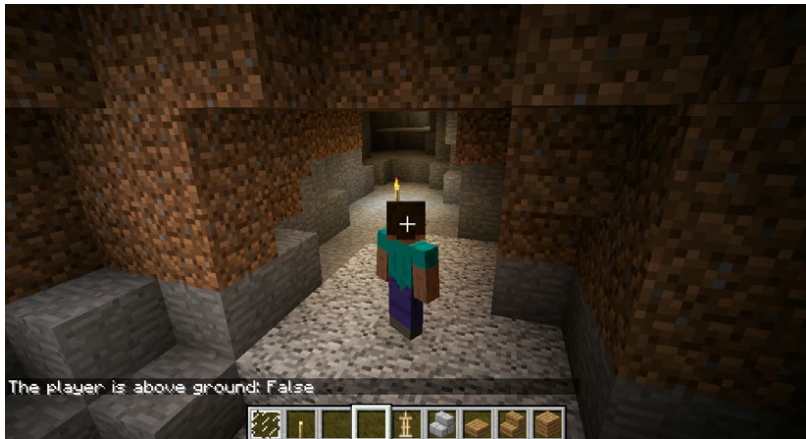


Bild 5.4: Python hat recht: Ich befinde mich unter der Erde.

Mission 21: Trautes Heim, Glück allein

Möglicherweise verirrst du dich bei deinen Wanderungen durch die Minecraft-Welt und kannst deine Basis nicht mehr finden. Du könntest stundenlang durch die Gegend spazieren, nur um dann festzustellen, dass du, als du durch verirrt hast, eigentlich ganz nahe bei deinem Zuhause warst.

Mit nur einer Codezeile kannst du überprüfen, wie weit du von einer beliebigen Position innerhalb des Spiels entfernt bist. So könntest du beispielsweise die Koordinaten deines Hauses und deine aktuelle Position angeben, um die Entfernung zwischen diesen beiden Punkten zu

berechnen. Und durch Hinzufügen eines Vergleichsoperators könntest du zudem feststellen, ob du dich in einer bestimmten (in Blöcken angegebenen) Entfernung von deinem Haus befindest. Sagen wir einmal, du befindest dich dann in der Nähe deines Hauses, wenn es höchstens 40 Blöcke weit weg ist.

Wir wollen ein Python-Programm schreiben, mit dem sich das überprüfen lässt. Der Code dieser Mission sollte feststellen, wie weit du dich von deinem Haus entfernt hast. Listing 5.5 zeigt ihn.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
import math
homeX = 10 ❶
homeZ = 10
pos = mc.player.getTilePos()
x = pos.x
z = pos.z
distance = math.sqrt((homeX - x) ** 2 + (homeZ - z) ** 2) ❷
mc.postToChat(distance) ❸
```

farFromHome.py

Listing 5.5: Dieser Code gibt die Entfernung zu deinem Haus an.

Dieser Code setzt voraus, dass sich dein Haus an den Koordinaten $x = 10$ und $z = 10$ befindet, die mit den Variablen `homeX` und `homeZ` ❶ festgelegt werden. In diesem Fall ist die y -Koordinate uninteressant. Mit der Funktion `getTilePos()` rufe ich die aktuelle Position des Spielers ab und lege die Werte für x und z fest.

Zur Berechnung der Variablen `distance` verwenden wir den *Satz des Pythagoras*. Mit dieser Formel lässt sich die Länge einer Seite eines rechtwinkligen Dreiecks berechnen – und in Minecraft die Entfernung zwischen zwei Punkten (in Bild 5.5 die Strecke von der rechts unten zur oberen Spitze des Dreiecks). Du kennst diese Formel wahrscheinlich aus dem Mathematikunterricht, wo sie wie folgt notiert wird: $a^2 + b^2 = c^2$. Hierbei sind a und b die beiden Katheten (also die am rechten Winkel anliegenden Seiten) eines rechtwinkligen Dreiecks, und c ist die Hypotenuse. Bei ❷ lösen wir nach c auf, was durch die Variable `distance` dargestellt wird.

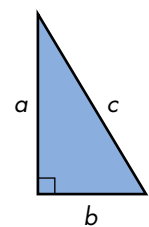


Bild 5.5: Rechtwinkliges Dreieck

Speichere Listing 5.5 als *farFromHome.py* im Ordner *booleans*.

Zur Abrundung des Programms überprüfst du mit einem Kleiner-oder-gleich-Operator, ob der Wert der Variablen `distance` kleiner oder gleich 40 ist, und `postest` das Ergebnis im Format "Your house is nearby: True/False" ❸ im Chat. Die Kombination des Strings mit dem

Vergleichsergebnis erfolgt mit einer Verkettung. Passe das Argument der Funktion `postToChat()` ③ wie erforderlich an, um den String auszugeben.

Nun teste das Programm. Befindest du dich in einer Entfernung von maximal 40 Blöcken von deinem Haus, dann sollte die Meldung `True` angezeigt werden; falls nicht, erscheint dagegen `False`. Bild 5.6 zeigt das Programm während der Ausführung.



Bild 5.6: Natürlich bin ich in der Nähe meines Hauses. Das ist doch die Eingangstür!

Logische Operatoren

Die Kombination von zwei oder mehr Vergleichsoperatoren wird in Programmen häufig eingesetzt. Oft möchte man ja feststellen, ob zwei Bedingungen gleichzeitig `True` sind – ob beispielsweise ein Auto rot ist *und* weniger als 10.000 Euro kostet.

Zur Kombination von zwei oder mehr Vergleichsoperatoren verwendest du *logische Operatoren*. Ähnlich wie Vergleichsoperatoren kannst du logische Operatoren überall dort verwenden, wo das gewünschte Ergebnis ein boolescher Wert ist. Logische Operatoren heißen auch *boolesche Operatoren*. Du wirst gleich drei Arten von Logikoperatoren kennenlernen: `and`, `or` und `not`.

and

Mit dem and-Operator kannst du überprüfen, ob zwei Vergleiche True sind. Damit ein Ausdruck mit einem and-Operator True ist, müssen die beiden enthaltenen Vergleiche ebenfalls True sein. Ist einer der Vergleiche False, dann gibt die Anweisung False zurück.

Angenommen, ich möchte herausfinden, ob eine Person älter als 18 Jahre ist und ein Auto besitzt. Hierzu könnte ich das folgende Programm schreiben:

```
>>> age = 21
>>> ownsCar = True
>>> age > 18❶ and ownsCar == True❷
True
```

Ich kombiniere hier die zwei Vergleichsoperatoren bei ❶ und ❷ mit and. Da das Alter der Person größer als 18 ist (age > 18 wird als True ausgewertet) und sie ein Auto besitzt (ownsCar == True), wird der gesamte Ausdruck age > 18 and ownsCar == True als True ausgewertet.

Wäre einer der Vergleiche False, dann würde False zurückgegeben. Nehmen wir nun an, die Person ist älter als 18, hat aber gar kein Auto:

```
>>> age = 25
>>> ownsCar = False
>>> age > 18 and ownsCar == True
False
```

Hier wird age > 18 als True ausgewertet, ownsCar == True hingegen als False, weswegen der gesamte Ausdruck False ist.

Tabelle 5.1 fasst alle möglichen booleschen Kombinationen und ihre Ergebnisse bei Verwendung des and-Operators zusammen.

Vergleich A	Vergleich B	A and B
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Tabelle 5.1: Kombinationen aus True und False mit dem and-Operator

Mission 22: Der Taucher

In Mission 18 (Seite 110) hast du überprüft, ob der Spieler gerade schwimmt. Das Programm hat je nachdem, ob der Block an der aktuellen Position des Spielers aus Wasser bestand oder nicht, `True` oder `False` zurückgegeben. Du hast damit herausgefunden, ob der Spieler mit den Füßen im Wasser steht. Allerdings geht aus der Antwort nicht hervor, ob sich auch der Kopf des Spielers unter Wasser befindet oder nicht. Wie also könntest du feststellen, ob sowohl die Füße als auch der Kopf des Spielers unter Wasser sind?

Mit einigen einfachen Änderungen, mit denen du einen `and`-Operator einfügst, kannst du mit dem Programm *swimming.py* überprüfen, ob sich der Spieler vollständig unter Wasser befindet. Öffne *swimming.py*, und speichere die Datei als *underwater.py*.

Nimm die folgenden Änderungen vor, damit das Programm überprüft, ob der Spieler vollständig von Wasser umgeben ist:

1. Füge eine zweite Variable hinzu, die den Typ des Blocks an der Position y-Position des Spielers + 1 ermittelt. Diese Variable speichert den Blocktyp am Kopf des Spielers. Wir wollen diese Variable `blockType2` nennen.
2. Überprüfe, ob sowohl `blockType` als auch `blockType2` Wasser sind.
3. Schreibe das Vergleichsergebnis mit der Meldung "The player is underwater: `True/False`" in den Chat.

Tipp

Um festzustellen, ob sowohl `blockType` als auch `blockType2` Wasser sind, kannst du den `and`-Operator verwenden. Zunächst überprüfst du mit dem Ausdruck `blockType == 9`, ob `blockType` Wasser ist. Danach ermittelst du mit `blockType2 == 9`, ob `blockType2` Wasser ist. Um die beiden zu kombinieren, setzt du einen `and`-Operator dazwischen: `blockType == 9 and blockType2 == 9`.

Wenn du das Programm ausführst, vergewissere dich, dass es in allen drei möglichen Fällen funktioniert: wenn der Spieler über dem Wasser ist, wenn nur seine Füße im Wasser stehen und wenn er vollständig von Wasser umschlossen ist. Bild 5.7 zeigt ein Beispiel für den Programmablauf.



Bild 5.7: Der Spieler läuft über den Meeresboden.

Bonus: Bin ich in einem Tunnel?

Überprüfe, ob sich der Spieler in einem Erd- oder Bruchsteintunnel befindet. Hierzu musst du die Blocktypen ober- und unterhalb des Spielers ermitteln.

or

Der Operator `or` arbeitet anders als `and`: Er gibt `True` zurück, wenn einer der beiden Vergleiche oder aber beide `True` sind. Solange also mindestens ein Vergleich `True` ist, ist auch der Gesamtausdruck `True`. Ist hingegen kein Vergleich `True`, dann wird der Ausdruck als `False` ausgewertet.

Nehmen wir an, ich möchte eine Katze aufnehmen. Diese muss aber entweder ein schwarzes oder ein orangefarbenes Fell haben. Mit dem folgenden Code könnte ich eine Benutzereingabe abrufen und dann überprüfen, ob der Wert des Strings entweder `"black"` oder `"orange"` ist:

```
catColor = input("What color is the cat?")
myCatNow = catColor == "black" or catColor == "orange"
print("Adopt this cat: " + str(myCatNow))
```

Wenn `catColor` einen der Werte "black" oder "orange" hat, findet die Katze bei mir ein neues Zuhause. Hat sie hingegen eine andere Farbe (z. B. "gray"), dann hat `myCatNow` den Wert `False`, und die Katze muss woanders hin.

Tabelle 5.2 enthält alle möglichen Kombinationen und ihre Ergebnisse bei Verwendung des `or`-Operators mit booleschen Variablen.

Vergleich A	Vergleich B	A or B
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Tabelle 5.2: Kombinationen aus `True` und `False` mit dem `or`-Operator

Mission 23: Sitze ich auf einem Baum?

Die Programme, die du in diesem Kapitel bislang erstellt hast, haben je nachdem, ob der Spieler auf oder in einem Block bestimmten Typs stand, `True` oder `False` angezeigt. Jetzt aber möchte ich überprüfen, ob der Spieler auf einem Baum sitzt. Wie könnte man das bewerkstelligen? Bäume bestehen ja aus Holz und Laub, d. h., du musst herausbekommen, ob der Spieler auf Holz oder Laub steht.

Schreiben wir dazu ein Programm. Öffne *swimming.py* erneut, und speichere es als neues Programm namens *inTree.py*.

Ändere das Programm so ab, dass der Typ des Blocks abgefragt wird, der sich einen Block unter dem des Spielers befindet. Du solltest hierzu den `or`-Operator verwenden, um zu ermitteln, ob der Block unter dem Spieler aus Laub (Blocktyp 18) oder Holz (Blocktyp 11) ist, und das Resultat dann im Chat posten.

Zur Erinnerung: Den Block unter dem Spieler kannst du mit `y = y - 1` abfragen.

Hinweis

Auch wenn Bäume und Laub unterschiedliche Farben aufweisen, haben alle Bäume dieselbe Block-ID, und auch Laubblöcke weisen jeweils dieselbe ID auf. (Die einzigen Ausnahmen sind Akazien- und Schwarzeichenholz und das zugehörige Laub, die jeweils einen anderen Blocktyp haben. Deswegen sollten wir Akazie und Schwarzeiche an dieser Stelle außen vor lassen.) Die Farbe wird mit einem zweiten Wert festgelegt, den wir in einem späteren Kapitel behandeln werden.

Wenn du das Programm ausführst, sollte die Ausgabe aus Bild 5.8 zu sehen sein.

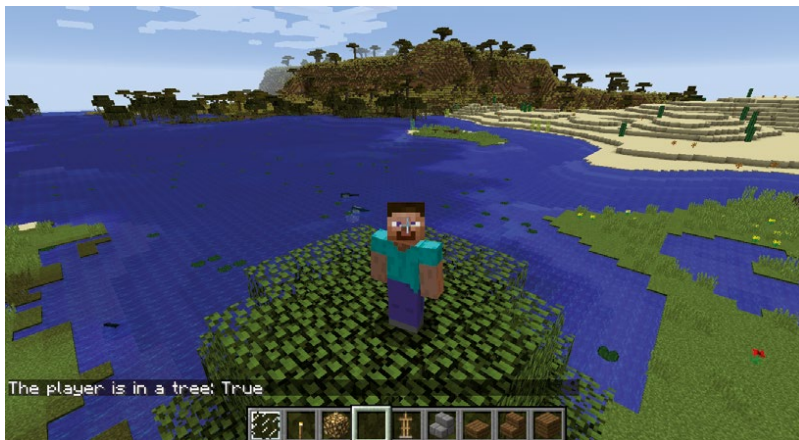


Bild 5.8: Ich sitze auf meinem Baum.

not

Der not-Operator unterscheidet sich erheblich von den and- und or-Operatoren. Er wird für einen einzelnen booleschen Wert oder einen Vergleich verwendet und verkehrt den übergebenen Wert schlicht ins Gegenteil.

Man könnte auch einfacher sagen: not macht aus True False und aus False True:

```
>>> not True
False
>>> not False
True
```

Das mag jetzt wenig spannend klingen, aber in Verbindung mit anderen logischen Operatoren kann der `not`-Operator tatsächlich sehr praktisch sein. Wir wollen den Wert `timeForBed` zuweisen, wenn du nicht hungrig, wohl aber müde bist:

```
>>> hungry = False
>>> sleepy = True
>>> timeForBed = not hungry and sleepy
>>> print(timeForBed)
True
```

Der `not`-Operator wirkt sich nur auf den booleschen Wert aus, dem er vorangestellt ist. Hier kehrt er den Wert der Variablen `hungry` um, lässt jedoch `sleepy` unbearbeitet. Da wir weiter oben `hungry` auf `False` gesetzt haben, ändert `not hungry` den Wert jetzt in `True`. Der Wert von `sleepy` ist ohnehin `True`. Beide Werte sind `True`, mithin ist `timeForBed` ebenfalls `True`.

Mission 24: Ist dieser Block keine Melone?

Du hast Hunger und möchtest wissen, ob etwas zu essen im Hause ist. Am liebsten isst du Melone, und deswegen lagerst du diese immer im selben Bereich deines Hauses. Allerdings weist du gerade nicht, ob noch eine Melone da ist, und deswegen möchtest du gern herausbekommen, ob du auf dem Heimweg noch etwas zu essen besorgen musst.

Aber zum Glück haben wir ja Python! Mit ein bisschen Hirnschmalz kannst du ein Python-Programm schreiben, mit dem du feststellen kannst, ob noch eine Melone da ist.

In dieser Mission schreibst du ein Programm, das dir sagt, ob du nach Nahrung suchen musst, bevor du zu deinem Minecraft-Haus zurückkehrst. Das Programm überprüft, ob sich an bestimmten Koordinaten ein Melonenblock befindet. Welche Koordinaten hierbei abgefragt werden, ist deine Angelegenheit: Diese können sich in deinem Haus, auf deiner Farm oder an irgendeinem anderen Ort befinden, wo du Melonen lagerst. Auch das Platzen einer Melone an den betreffenden Koordinaten liegt ganz bei dir.

Kopiere Listing 5.6, und speichere es als *notAMelon.py*.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

x = 10
y = 11
z = 12
melon = 103 ❶
block = mc.getBlock(x, y, z) ❷

noMelon = # Überprüfen, ob der Block keine Melone ist ❸

mc.postToChat("You need to get some food: " + str(noMelon)) ❹
```

notAMelon.py

Listing 5.6: Am Anfang des Codes wird geprüft, ob sich an einer bestimmten Position eine Melone befindet.

Genauer gesagt soll mit diesem Code festgestellt werden, ob der Block an einer bestimmten Position ein Melonenblock ist. Ich habe eine Variable namens `melon` eingefügt, in der die Block-ID einer Melone (103) gespeichert ist ❶, die Methode `getBlock()` aufgerufen und das Ergebnis in einer Variablen namens `block` ❷ gespeichert. Um das Programm zu vervollständigen, musst du die Zeile bei ❸ ergänzen, mit der überprüft wird, ob der Wert der Variablen `melon` ungleich dem Wert der Variablen `block` ist. Das Ergebnis soll in der Variablen `noMelon` gespeichert werden, um es in der letzten Zeile ❹ an den Minecraft-Chat auszugeben.

Du kannst die Überprüfung ❸, ob die Variablen `melon` und `block` nicht gleich sind, auf zweierlei Weise notieren: entweder mit dem Ungleichheitsoperator oder mit dem logischen Operator `not`. Das Ganze funktioniert zwar unabhängig von der gewählten Vorgehensweise, aber wir wollen in diesem Programm einmal den logischen Operator `not` verwenden.

Führe das Programm aus, nachdem du alle erforderlichen Änderungen vorgenommen hast. Das Ergebnis sollte etwa so aussehen wie in Bild 5.9.



Bild 5.9: Auf meiner Farm ist eine Melone! Was brauche ich mehr?

Bonus: Eine gut gefüllte Speisekammer

Ändere den Blocktyp, auf den das Programm prüft. So kannst du beispielsweise feststellen, ob auf deiner Farm Mais wächst oder irgendetwas deine Haustür entwendet hat.

Bearbeitungsreihenfolge bei logischen Operatoren

Du kannst beliebig viele logische Operatoren in einer einzelnen Anweisung kombinieren. Hier etwa haben wir eine etwas abgedrehte Kombination aus and, or und not:

```
>>> True and not False or False
True
```

Dieser Code wird als True ausgewertet. Überrascht? Also der Reihe nach: Der Teil not False der Anweisung wird als True ausgewertet. Dies ist äquivalent mit Folgendem:

```
>>> True and True or False
True
```

Das `and` wird als Nächstes ausgewertet: `True and True` ist `True`, das heißt:

```
>>> True or False
True
```

Schließlich wird das `or` ausgewertet: `True or False` wird `True`.

Python wertet logische Operatoren in einer bestimmten Reihenfolge aus. Wenn du diese Reihenfolge nicht richtig beachtest, dann erhältst du womöglich ein anderes Ergebnis als erwartet. Die folgenden Operatoren wertet Python als Erstes, Zweites und Drittes aus:

1. `not`
2. `and`
3. `or`

Übe ein wenig das Erstellen von Anweisungen mit logischen Operatoren in IDLE, und überprüfe, ob du die Resultate stets nachvollziehen kannst.

Liegt meine Zahl zwischen zwei anderen?

Häufig musst du feststellen, ob ein Wert kleiner als ein anderer Wert und größer als ein dritter ist. Stellen wir uns vor, du möchtest dafür Sorge tragen, dass du zwischen zehn und zwanzig Wölfe hast. Du magst Wölfe nämlich gern und möchtest deswegen mindestens zehn davon halten, aber mehr als zwanzig sollten es nicht sein, da du ansonsten nicht genug Futter für alle hast. Diese Bedingung könntest du mit einem `and`-Operator überprüfen:

```
wolves = input("Enter the number of wolves: ")
enoughWolves = wolves > 10 and wolves < 20
print("Enough wolves: " + str(enoughWolves))
```

Man kann es aber auch anders machen: Statt den `and`-Operator zu verwenden, schreibst du die Variable einmal zwischen zwei Vergleiche:

```
wolves = input("Enter the number of wolves: ")
enoughWolves = 10 < wolves < 20
print("Enough wolves: " + str(enoughWolves))
```

Führst du nun eines dieser Programme aus und gibst einen Wert zwischen 11 und 19 ein, dann wird `enoughWolves` als `True` ausgewertet. Das

Gleiche kannst du auch mit dem Größer-oder-gleich-Operator ($\geq$) bzw. dem Kleiner-oder-gleich-Operator ($\leq$) machen:

```
wolves = input("Enter the number of wolves: ")
enoughWolves = 10 <= wolves <= 20
print("Enough wolves: " + str(enoughWolves))
```

In diesem Fall würde die Eingabe von 10 oder 20 `enoughWolves` ebenfalls `True` werden lassen.

Mission 25: Bin ich zu Hause?

Mit Python-Code kannst du, wenn du in einem bestimmten Bereich der Karte unterwegs bist, einige coole Sachen machen. Du könntest beispielsweise eine Geheimtür öffnen, wenn der Spieler einen bestimmten Block betritt, oder eine Fallgrube basteln, in die der Spieler stürzt, wenn er darüberläuft. In dieser Mission zeige ich dir, wie du feststellen kannst, ob sich jemand in deinem Minecraft-Haus befindet.

Wir haben in Mission 8 (Seite 69) bereits ein Programm erstellt, mit dem sich die Wände, die Decke und der Boden eines Gebäudes automatisch bauen lassen. Dieses Programm haben wir unter dem Namen *building.py* im Ordner *math* gespeichert. Öffne es.

Lies den Code in *building.py*, und achte dabei vor allem auf die Werte der Variablen *width*, *height* und *length* (standardmäßig waren dies 10, 5 und 6). Notiere dir ferner die Koordinaten deiner aktuellen Position. Führe das Programm aus, um ein Haus zu bauen.

Nach dem erfolgreichen Hausbau können wir nun ein Programm wie in Listing 5.7 schreiben, mit dem du feststellen kannst, ob sich der Spieler im Haus befindet.

insideHouse.py

```
from mcpi.minecraft import Minecraft
mc = minecraft.create()

❶ buildX =
buildY =
buildZ =
❷ width = 10
height = 5
length = 6

pos = mc.player.getTilePos()
x = pos.x
```

```
y = pos.y  
z = pos.z
```

```
③ inside = buildX < x < buildX + width and
```

Listing 5.7: Anfang des Programms, mit dem du feststellen kannst, ob der Spieler sich im Haus befindet

Listing 5.7 soll überprüfen, ob die x-Koordinate des Spielers sich innerhalb des Gebäudes befindet, das mit *building.py* erstellt wurde. Allerdings ist das Programm noch nicht fertig. Deine Aufgabe besteht nun darin, dafür zu sorgen, dass das Programm auch die y- und z-Koordinaten mit den Koordinaten des Hauses vergleicht, das du mit dem Programm *building.py* gebaut hast.

Kopiere Listing 5.7 in eine neue Datei, und speichere diese als *inside-House.py*. Vervollständige das Programm dann so, dass es feststellt, ob sich der Spieler innerhalb des Gebäudes befindet.

Hierzu gehst du wie folgt vor:

1. Füge die Koordinaten des Gebäudes hinzu. Diese entsprechen den Koordinaten deiner Position bei der Ausführung des Programms *building.py* ①.
2. Falls erforderlich, ändere die Variablen *width*, *height* und *length* ab, sofern sie sich von denen unterscheiden, die du in deinem Programm *building.py* verwendet hast ②.
3. Vervollständige den Vergleich für die Variable *inside*, sodass ermittelt wird, ob sich der Spieler innerhalb des Gebäudes befindet. Der erste Teil – die Überprüfung, ob sich die x-Position innerhalb des Hauses befindet – ist bereits erledigt ③. Du musst allerdings noch die Vergleiche für die y- und z-Positionen ergänzen. Die Ausdrücke ähneln jenem, den ich für die x-Position verwendet habe (*buildX < x < buildX + width*).
4. Poste den Wert der Variablen *inside* im Chat.
5. Hast du alle notwendigen Änderungen vorgenommen, speichere das Programm und führe es aus. Die Ausgabe sollte so ähnlich aussehen wie in Bild 5.10.



Bild 5.10: Ich in meinem Schlafzimmer – und damit natürlich zu Hause.

Was du gelernt hast

Du hast in diesem Kapitel mit booleschen Variablen, Vergleichsoperatoren und logischen Operatoren Fragen in deinen Programmen beantwortet. In Kapitel 6 wirst du Programme schreiben, die auf der Grundlage solcher Antworten Entscheidungen treffen. Du wirst überprüfen, ob eine Bedingung wahr ist oder nicht, und dafür sorgen, dass je nachdem, ob die Bedingung wahr ist oder nicht, in deinem Programm unterschiedlicher Code ausgeführt wird. In Kapitel 7 werden deine Programme dann einen Codeabschnitt so lange ausführen, bis eine Bedingung, die zuvor wahr war, dies nicht mehr ist. Dort werden wir die wahre Leistungsfähigkeit von booleschen Variablen und Vergleichsoperatoren wirklich kennenlernen. Du kannst damit nämlich steuern, welcher Code zu welchem Zeitpunkt in deinem Programm ausgeführt wird.

Kapitel 6

Minispiele mit if-Anweisungen basteln

In Kapitel 5 hast du gelernt, wie man in Python Fragen stellt. Du hast Vergleichsoperatoren (wie `==`, `!=`, `>`, `<` usw.) und logische Operatoren (`and`, `or`, `not`) genutzt, um herauszufinden, ob eine Bedingung oder eine Gruppe von Bedingungen als `True` oder `False` ausgewertet wird. In diesem Kapitel wirst du auf Basis der Antworten zu diesen Fragen – also der Ergebnisse der geprüften Bedingungen – entscheiden, welcher Code ausgeführt werden soll.

Entscheidungen auf der Grundlage von Bedingungen treffen wir ja nun täglich. Ist es Nacht? Wenn ja, solltest du deine Diamantrüstung anlegt haben und ein Schwert bei dir tragen, um Monster zu bekämpfen. Andernfalls kannst du das ganze Zeug in deiner Geheimbasis lassen. Bist du hungrig? Dann solltest du etwas Brot oder einen Apfel essen. Oder du bist nicht hungrig. Dann kannst du natürlich auf große Fahrt gehen, um dir etwas Appetit zu verschaffen. Und so, wie du tagtäglich Entscheidungen im echten Leben triffst, müssen deine Programme abhängig von den Bedingungen jeweils unterschiedliche Aufgaben ausführen.

Damit Programme Entscheidungen treffen können, verwenden wir einen speziellen Python-Code. `if`-Anweisungen sagen dem Programm, ob ein bestimmter Codeabschnitt ausgeführt werden soll oder nicht. Eine `if`-Anweisung bedeutet: »Wenn diese Bedingung wahr ist, führe den folgenden Code aus«. Du könntest beispielsweise überprüfen, ob der Spieler sich in einem verbotenen Raum befindet und – wenn dies der Fall ist – den Boden in Lava verwandeln. Oder du findest heraus, ob er einen bestimmten Block an einer bestimmten Position platziert hat, und öffnest dann ggf. eine Geheimtür. Mit Bedingungen und `if`-Anweisungen kannst du eigene Minigames in Minecraft basteln.

if-Anweisungen verwenden

Die Ausführung eines Programms beeinflussen zu können gibt dir nicht nur eine Menge Macht, sondern ist für das Coden unabdingbar! Programmierer bezeichnen dieses Konzept manchmal als *Ablaufsteuerung*. Die einfachste Möglichkeit, um eine Ablaufsteuerung zu implementieren, ist die Verwendung einer einfachen if-Anweisung, die einen Codeabschnitt nur dann ausführt, wenn eine Bedingung den Wert True hat.

Eine if-Anweisung besteht aus drei Teilen:

- dem if-Operator
- einer zu testenden Bedingung
- einem Codeabschnitt, der ausgeführt wird, wenn die Bedingung den Wert True hat

Sehen wir uns eine if-Anweisung in Aktion an. Der folgende Code gibt die Meldung "That's a lot of zombies" nur dann aus, wenn mehr als 20 Zombies da sind. Ist das nicht der Fall, dann passiert überhaupt nichts.

```
zombies = int(input("Enter the number of zombies: "))
if zombies > 20:
    print("That's a lot of zombies.")
```

Hier ist `zombies > 20` die Bedingung, auf die getestet wird, und `print("That's a lot of zombies.")` ist der Code der if-Anweisung – also der Code, der ausgeführt wird, wenn `zombies > 20` als True ausgewertet wird. Der Doppelpunkt (:) am Ende der if-Zeile sagt Python, dass jetzt der if-Anweisungskörper beginnt. An dem Einzug erkennt Python, welche Codezeilen zu diesem Anweisungskörper gehören. *Einzug* bezeichnet hier zusätzliche Leerzeichen am Anfang einer Textzeile. In Python werden Zeilen um vier Leerzeichen eingezogen. Wenn wir innerhalb der if-Anweisung zusätzliche Zeilen hinzufügen wollten, dann müssten diese alle um dieselbe Anzahl Leerzeichen eingezogen werden wie die Zeile `print("That's a lot of zombies.")`.

Führe den Code einige Male aus, teste dabei alle denkbaren Bedingungen, und stelle fest, was passiert. Gib eine Zahl unter 20, die Zahl 20 selbst und eine Zahl ein, die größer ist als 20. Was beispielsweise bei der Eingabe von 22 geschieht, siehst du hier:

```
Enter the number of zombies: 22
That's a lot of zombies.
```

Gut, das ist ein sinnvolles Ergebnis. Führen wir den Code nun noch einmal aus, und überprüfen wir, was passiert, wenn die Bedingung nicht erfüllt ist.

```
Enter the number of zombies: 5
```

Wie du siehst, siehst du nichts: Das passiert nämlich, wenn die Bedingung als `False` ausgewertet wird. Der Anweisungskörper der `if`-Anweisung wird vollständig ignoriert. Eine `if`-Anweisung führt den Code im Anweisungskörper nur dann aus, wenn die abgefragte Bedingung erfüllt ist. Ist die `if`-Anweisung verarbeitet, wird das Programm mit der Zeile fortgesetzt, die auf den `if`-Anweisungskörper folgt.

Sehen wir uns noch ein Beispiel an, um besser nachvollziehen zu können, wie das funktioniert. Der folgende Code überprüft mithilfe einer `if`-Anweisung, ob ein eingegebenes Passwort richtig ist:

```
password = "cats"
attempt = input("Please enter the password: ")
if attempt == password:
    print("Password is correct")
print("Program finished")
```

Der Ausdruck, der direkt auf die `if`-Anweisung folgt, ist die Bedingung: `attempt == password`. Die eingezogene Zeile, die auf `if attempt == password` folgt, ist der Anweisungskörper der `if`-Anweisung: `print("Password is correct")`.

Dieser Code gibt `"Password is correct"` nur dann aus, wenn der in der Variablen `attempt` gespeicherte Wert mit dem Wert der Variablen `password` identisch ist. Andernfalls wird überhaupt nichts ausgegeben. Die letzte Zeile wird dann in jedem Fall ausgeführt, d. h., `"Program finished"` wird unabhängig davon angezeigt, ob der `if`-Anweisungskörper ausgeführt wurde oder nicht.

Damit kommen wir nun zu einer etwas explosiveren Angelegenheit.

Mission 26: Wir basteln uns einen Krater

Du weißt bereits, wie man den Spieler teleportiert und hochspringen lässt. Jetzt werden wir dafür sorgen, dass die Blöcke um den Spieler herum verschwinden.

Wenn dieses Programm ausgeführt wird, verwandeln sich die Blöcke über, unter und neben dem Spieler in Luft. Seine Kraft ist hochzerstörerisch – sei vorsichtig, wenn du es einsetzt. Zur Sicherheit fragt das Programm den Benutzer, ob er die Blöcke wirklich zerstören möchte, und tut dies nur, wenn diese Frage bejaht wird.

Listing 6.1 schafft einen Krater um den Spieler herum, indem alle Blöcke über, unter und neben ihm gelöscht werden. Danach wird "Boom!" in den Chat gepostet. Speichere dieses Programm als *crater.py* in einem neuen Ordner namens *ifStatements*.

crater.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

answer = input("Create a crater? Y/N ")

# Hier eine if-Anweisung hinzufügen ❶

pos = mc.player.getPos()
mc.setBlocks(pos.x + 1, pos.y + 1, pos.z + 1, pos.x - 1, pos.y - 1, ↵
pos.z - 1, 0) ❷
mc.postToChat("Boom!")
```

Listing 6.1: Dieser Code erstellt in jedem Fall einen Krater.

Die Variable *answer* fragt den Benutzer mithilfe der Funktion *input()*, ob er einen Krater erschaffen möchte. Im obigen Code wird der Krater jedoch unabhängig davon erstellt, ob der Benutzer Y, N, irgendetwas anderes oder überhaupt nichts eingegeben hat.

Zur Vervollständigung des Programms brauchst du eine *if*-Anweisung. Mit dieser überprüfst du, ob der Benutzer als Antwort Y eingegeben hat. Diese Logik fügst du bei ❶ hinzu. Vergiss nicht, dass die Antwort des Benutzers in der Variablen *answer* gespeichert wird. Das heißt, deine *if*-Anweisung sollte diese Variable überprüfen. Nach dem Hinzufügen der *if*-Anweisung sollte das Programm die letzten drei Zeilen nur dann ausführen, wenn der Benutzer Y eingegeben hat. Hierzu musst du diese Zeilen um vier Leerzeichen einziehen. Achte auf Groß- bzw. Kleinschreibung und nimm den gleichen Buchstaben, den du in der *if*-Anweisung verwendet hast, damit die Abfrage richtig funktioniert.

Achte darauf, dass das allerletzte Argument der Funktion `setBlocks()` den Blocktyp angibt, den du festlegen möchtest. Hier lautet das letzte Argument 0, wodurch Luft als Blockmaterial angegeben wird. Mit anderen Worten: Der Krater wird mit der Funktion `setBlocks()` erstellt, die wir benutzen, um das Blockmaterial in Luft umzuwandeln ②. Hierdurch wird der Eindruck erweckt, dass alle Blöcke um den Spieler herum zerstört wurden. Durch Addieren oder Subtrahieren von 1 zu bzw. von den Werten von `pos.x`, `pos.y` und `pos.z` platziert der Code Luftblöcke in einem Würfel mit der Kantenlänge 3 um den Spieler herum. Dies ist der Krater.

Hast du alle Änderungen vorgenommen, dann speichere das Programm und führe es aus. Die Frage `Create a Crater? Y/N` wird in der Python-Shell angezeigt. Gib entweder Y oder N ein. Verwende dabei nur Großbuchstaben, da das Programm andernfalls nicht richtig funktionieren wird.

Wenn der Benutzer Y eingibt, erscheint ein Krater (Bild 6.1).



Bild 6.1: Rumms! Auf einmal ist da ein Krater.

Bonus: Hausbau leicht gemacht

Was könntest du noch mit diesem Programm machen? Versuche doch einmal, es so abzuändern, dass anstelle eines Kraters ein Haus um den Spieler herum gebaut wird.

else-Anweisungen

Als Nächstes wollen wir uns eine etwas komplexere Anweisung ansehen. Diese verwenden wir, um einen *anderen* Code auszuführen, wenn die `if`-Bedingung `False` zurückgibt. In diesem Fall kommt die `else`-Anweisung ins Spiel.

Eine `else`-Anweisung funktioniert nur in Kombination mit einer `if`-Anweisung. Die Vorgehensweise sieht so aus, dass du zunächst eine `if`-Anweisung schreibst, die bestimmten Code ausführt, wenn sie `True` ist. Nach der `if`-Anweisung musst du dann eine `else`-Anweisung verfassen, um anderen Code für den Fall auszuführen, dass die obige Bedingung `False` ist. Einfacher ausgedrückt: Wenn die Bedingung erfüllt ist, muss dies getan werden, andernfalls jenes.

Das folgende Programm gibt "Ahhhh! Zombies!" aus, wenn sich mehr als 20 Zombies im Raum befinden. Sollte dies hingegen nicht zutreffen, dann wird "You know, you zombies aren't so bad" ausgegeben.

```
zombies = int(input("Enter the number of zombies: "))
if zombies > 20:
    print("Ahhhh! Zombies!")
else:
    print("You know, you zombies aren't so bad.")
```

Wie die `if`-Anweisung verwendet auch `else` einen Doppelpunkt und den Einzug, um zu signalisieren, welcher Code in den Anweisungskörper gehört. Allerdings kann `else` nicht für sich verwendet werden: Es ist doch logisch, dass zuerst eine `if`-Anweisung im Code stehen muss. Die `else`-Anweisung verfügt auch über keine eigene Bedingung, sondern der Anweisungskörper wird in jedem Fall ausgeführt, wenn die Bedingung der `if`-Anweisung (hier `zombies > 20`) nicht `True` ist.

In unserem Passwortbeispiel von weiter oben könnten wir in einer `else`-Anweisung eine Meldung ausgeben lassen, wenn das Passwort falsch ist:

```
password = "cats"
attempt = input("Please enter the password: ")
if attempt == password:
    print("Password is correct.")
else:
    print("Password is incorrect.")
```

Wenn der Wert von `attempt` mit dem Wert von `password` übereinstimmt, wird die Bedingung `True`. Das Programm führt dann den Code aus, der `"Password is correct"` ausgibt.

Ist `attempt` hingegen nicht mit `password` identisch, dann wird die Bedingung `False`. In diesem Fall führt das Programm den Code aus, der `"Password is incorrect"` ausgibt.

Was wäre nun, wenn wir eine `else`-Anweisung ohne vorangehendes `if` ausführten? Wenn also beispielsweise ein Programm nur die folgenden beiden Zeilen umfasste:

```
else:
    print("Nothing happened.")
```

Hier würde Python nicht verstehen, was du von ihm willst, und folgerichtig eine Fehlermeldung ausgeben.

Mission 27: Blöcke nicht zerschlagen – oder doch?

In Mission 17 (Seite 141) hast du ein Programm geschrieben, mit dem verhindert wurde, dass der Spieler Blöcke zerschlägt. Hierzu hast du deine Welt mit `mc.setting("world_immutable", True)` unveränderlich gemacht. Auf diese Weise konntest du deine wertvollen Kreationen vor Unglück und Vandalismus schützen. Das war praktisch, aber nicht besonders flexibel. Die Deaktivierung dieser Schutzfunktion erforderte nämlich ein zweites Programm, was recht unpraktisch ist.

Mit einer `if`-Anweisung, einer `else`-Anweisung und einer Konsoleneingabe kannst du ein Programm entwickeln, mit dem sich die Unveränderlichkeit ein- und ausschalten lässt. Dein Programm fragt dann nach, ob die Blöcke unveränderlich werden sollen, und weist »immutable« je nach deiner Antwort `True` oder `False` zu.

Öffne IDLE, und erstelle eine neue Datei. Speichere diese dann als *immutableChoice.py* im Ordner *ifStatements*. Dann führst du die folgenden Schritte aus, um das Programm fertigzustellen:

1. Das Programm soll den Benutzer fragen, ob er die Blöcke unveränderlich machen möchte:

```
"Do you want blocks to be immutable? Y/N"
```

Füge diesen String als Argument in `input()` ein, und speichere die Eingabe in einer Variablen namens `answer`.

2. Das Programm überprüft nun, ob der in `answer` gespeicherte Wert "Y" ist. Falls ja, wird folgender Code ausgeführt:

```
mc.setting("world_immutable", True)
mc.postToChat("World is immutable")
```

Kopiere diesen Code, und füge ihn in eine `if`-Anweisung ein, damit er nur ausgeführt wird, wenn der Wert der Variablen `answer` "Y" ist. Einzug nicht vergessen!

3. Das Programm führt den folgenden Code aus, wenn die Variable `answer` nicht den Wert "Y" hat:

```
mc.setting("world_immutable", False)
mc.postToChat("World is mutable")
```

Kopiere diesen Code, und lege ihn in der vorgesehenen `else`-Anweisung ab.

Speichere das Programm, und führe es aus. Wenn du gefragt wirst, ob du die Blöcke unveränderlich machen möchtest, gib Y oder N ein, und drücke die Eingabetaste. Nun teste das Programm. Wenn du entscheidest, die Blöcke unveränderlich zu machen, dann sollten sie nicht kaputtgehen; andernfalls sollten sie sich zerstören lassen.

Bild 6.2 zeigt die ausgegebene Meldung und die Frage in der Shell.

Wenn du "N" oder irgendeinen Blödsinn wie "banana" eingibst, erhältst du jeweils dasselbe Ergebnis. Warum, glaubst du, ist das so?

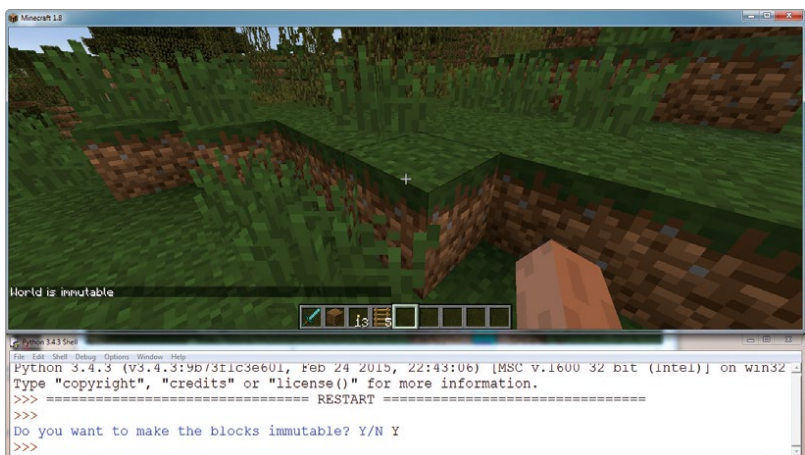


Bild 6.2: Ich kann meine Welt unverwundbar machen. Jetzt lassen sich keine Blöcke mehr zerstören.

Bonus: Verbesserte Benutzerfreundlichkeit

Wir könnten die Benutzung dieses Programms noch einfacher gestalten, indem wir mithilfe boolescher Operatoren Varianten von »Yes« und »No« akzeptierten, also etwa »yes«, »YES« oder auch eine Antwort, die nur einen Buchstaben umfasst (»y«). Probier's mal aus!

elif-Anweisungen

Mit einer `if`- und einer `else`-Anweisung führt dein Programm je nachdem, ob eine Bedingung `True` oder `False` ist, unterschiedlichen Code aus. Aber könnte man nicht auch mehr als zwei Codeblöcke zur alternativen Ausführung festlegen?

Das geht – mit einer `else-if`-Anweisung, die in Python `elif` heißt. Du schreibst zunächst eine `if`-, dann eine `elif`- und schließlich eine `else`-Anweisung. Wenn du diese Anweisungen gemeinsam verwendest, sagst du prinzipiell: »Wenn eine Bedingung `True` ist, führe diesen Code aus. Andernfalls wird, wenn eine zweite, andere Bedingung `True` ist, ein anderer Codeabschnitt gestartet. Schließlich soll, wenn keine dieser beiden Bedingungen `True` ist, noch ein anderer Codeabschnitt ausgeführt werden.«

Wir wollen uns das einmal genauer ansehen. Nehmen wir an, du denkst gerade darüber nach, welche Eissorten du auf der Waffel haben möchtest, die du gleich in einer Eisdiele kaufen wirst. Wenn noch Schokolade da ist, dann möchtest du davon eine Kugel haben. Ist Schokolade aus, aber Erdbeere noch zu haben, dann nimmst du Erdbeereis. Schließlich würdest du dich, wenn Schokolade und Erdbeere alle sind, für Vanille entscheiden.

In einem Programm sieht dieser Entscheidungsprozess wie folgt aus:

```
hasChocolate = False
hasStrawberry = True
if hasChocolate:
    print("Hooray! I'm getting chocolate.")
elif hasStrawberry:
    print("I'm getting the second best flavor, strawberry.")
else:
    print("Vanilla is OK too, I guess.")
```

Die ersten beiden Zeilen haben lediglich die Bühne für dieses Szenario bereitet: Wir gehen davon aus, dass Schokolade ausverkauft ist, Erdbeere jedoch nicht. Deswegen haben wir `hasChocolate` auf `False` und `hasStrawberry` auf `True` gesetzt.

Danach folgt die Entscheidungslogik: Eine `if`-Anweisung gibt "Hooray! I'm getting chocolate." aus, wenn `hasChocolate` den Wert `True` hat. In diesem Beispiel jedoch ist der Wert `False`, d. h., die Meldung wird nicht ausgegeben. Stattdessen wendet sich das Programm nun der `elif`-Anweisung zu und überprüft, ob `hasStrawberry` den Wert `True` hat. Dies trifft zu, und deswegen wird der Code im `elif`-Anweisungskörper ausgeführt. Folglich wird die Meldung "I'm getting the second best flavor, strawberry" angezeigt.

Wie du siehst, hat diese `elif`-Anweisung eine eigene Bedingung und einen eigenen Anweisungskörper. Die `elif`-Anweisung wird nur ausgeführt, wenn die Bedingung der `if`-Anweisung `False` und die Bedingung der `elif`-Anweisung `True` ist.

Abschließend wird die auf die `elif`-Anweisung folgende `else`-Anweisung ausgeführt, sofern die Bedingung der `if`-Anweisung `False` und die Bedingung der `elif`-Anweisung ebenfalls `False` ist. In unserem Beispiel würde der Code der `else`-Anweisung ausgeführt, sofern sowohl `hasChocolate` als auch `hasStrawberry` `False` wären; dann würde "Vanilla is OK too, I guess" ausgegeben.

Noch ein Beispiel. Wir haben weiter oben ein Programm gesehen, das "Ahhhh! Zombies!" ausgibt, wenn mehr als 20 Zombies in einem Raum sind. Mit einer `elif`-Anweisung können wir eine weitere Bedingung testen, wenn die Bedingung der `if`-Anweisung `False` ist:

```
zombies = int(input("Enter the number of zombies: "))
if zombies > 20:
    print("Ahhhh! Zombies!")
elif zombies == 0:
    print("No zombies here! Phew!")
else:
    print("You know, you zombies aren't so bad.")
```

Wir fügen eine `elif`-Anweisung hinzu, um `zombies` mit 0 zu vergleichen. Wenn `zombies == 0` den Wert `True` hat, gibt der Code im `elif`-Anweisungskörper "No zombies here! Phew!" aus. Hat die Bedingung der `elif`-Anweisung jedoch den Wert `False`, dann geht der Code weiter zur `else`-Anweisung und gibt "You know, you zombies aren't so bad" aus.

Mission 28: Geschenke, Geschenke!

Lass uns nun einmal ein Programm erstellen, das überprüft, ob auf einem bestimmten Block ein Geschenk steht, und das abhängig von der Art des Geschenks jeweils eine andere Reaktion zeigt.

Mit diesem Programm kannst du eines von zwei verschiedenen Geschenken auf dem Block ablegen. Eines der Geschenke ist ein Diamantblock, und weil dies ein rar gesätes Item ist, bleibt als zweites Geschenk nur ein Setzling übrig.

In Listing 6.2 wird überprüft, ob ein Block an der Position 10, 11, 12 ein Diamantblock oder ein Baumsetzling ist oder ob gar kein Geschenk vorhanden ist. Allerdings ist das Programm noch nicht vollständig.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
x = 10
y = 11
z = 12
gift = mc.getBlock(x, y, z)

# Falls das Geschenk ein Diamantblock ist
if ❶

# sonst falls das Geschenk ein Setzling ist
elif ❷

else:
    mc.postToChat("Bring a gift to " + str(x) + ", " + str(y) + ", " + str(z))
```

gift.py

Listing 6.2: Der Anfang des Codes, der feststellt, ob du ein Geschenk mitgebracht hast

Erstelle eine neue Datei in IDLE, und speichere sie als *gift.py* im Ordner *ifStatements*. Kopiere Listing 6.2 in diese Datei. Der Code, der den Blocktyp abrufen ist bereits fertig. Der Blocktyp wird in der Variablen *gift* gespeichert. Die *else*-Anweisung wird ausgeführt, sofern weder ein Diamantblock noch ein Baumsetzling vorhanden ist. Der Spieler wird dann über den Chat aufgefordert, ein Geschenk zu besorgen und zu diesen Koordinaten zu bringen. Du kannst die Koordinaten in den Variablen *x*, *y* und *z* auf beliebige Werte setzen.

Zum Vervollständigen des Programms führst du die folgenden Schritte aus:

1. Erweitere die if-Anweisung bei ❶, damit sie prüft, ob die Variable `gift` den Wert für einen Diamantblock (57) enthält. Falls ja, soll das Programm die Meldung "Thanks for the diamond" im Chat ausgeben.
2. Füge unter dem zweiten Kommentar bei ❷ eine `elif`-Anweisung hinzu, die prüft, ob die Variable `gift` den Wert für einen Baumsetzling (6) enthält. Tut sie dies, dann soll folgende Meldung an den Chat übermittelt werden: "I guess tree saplings are as good as diamonds..."

Wenn du alle Änderungen vorgenommen hast, speicherst du das Programm und führst es aus. Positioniere einen Diamantblock an den Koordinaten, und stelle fest, was geschieht. Mache dasselbe dann mit einem Setzling, und probiere anschließend auch einmal, keinen Block an die Koordinaten zu setzen. Vergiss dabei nicht, dass der Setzling in einem Gras- oder Erdblock angepflanzt werden muss. Erhältst du jedes Mal die richtige Antwort? Du musst das Programm für jede Überprüfung erneut ausführen. Bild 6.3 zeigt mein funktionsfähiges Programm.

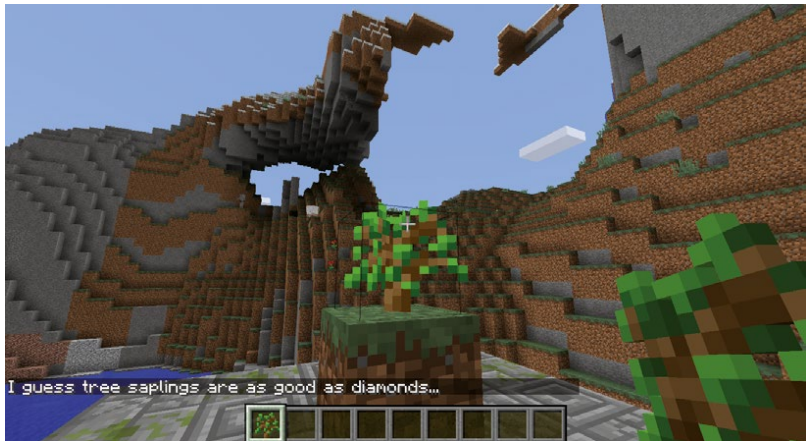


Bild 6.3: Ein Setzling. Das ist ein tolles Geschenk.

Bonus: Lobet den Gott der Melone

Du kannst in dieser Mission viele verschiedene Blöcke verwenden. Ändere den Code doch einmal so ab, dass er überprüft, ob du einen Goldblock oder eine Melone als Geschenk darbietest. Du könntest den Code auch so erweitern, dass dein Geschenk nach dem Platzieren zerstört wird.

elif-Anweisungen verketteten

Die Anzahl der `elif`-Anweisungen, die sich in eine `if`-Anweisung einfügen lassen, ist unbegrenzt. Man kann mit nur einer `elif`-Anweisung arbeiten oder mit hundert: Python wird sie stur eine nach der anderen auswerten.

Hier ist ein Beispiel mit dem Programm, das die Anzahl anwesender Zombies auswertet:

```
zombies = int(input("Enter the number of zombies: "))
if zombies > 20:
    print("Ahhhh! Zombies!")
elif zombies > 10: ❶
    print("There's just half a Minecraft zombie apocalypse.")
elif zombies == 0:
    print("No zombies here! Phew!")
else:
    print("You know, you zombies aren't so bad.")
```

Wir haben hier eine neue `elif`-Anweisung bei ❶ – unmittelbar nach der `if`-Anweisung – hinzugefügt, mit der wir überprüfen, ob mehr als zehn Zombies im Raum sind. Ist dies der Fall, dann gibt das Programm "There's just half a Minecraft zombie apocalypse" aus, ansonsten schreitet es fort zum nächsten `elif`.

Wichtig ist dabei die Reihenfolge der `if`- und `elif`-Anweisungen. Wenn du hierbei einen Fehler machst, dann kann es sein, dass ein Teil deines Codes nie ausgeführt wird und das Programm nicht wie erwartet funktioniert.

Tauschen wir beispielsweise die Bedingung der `if`-Anweisung und die der ersten `elif`-Anweisung gegeneinander aus, dann bekommen wir ein Problem:

```
zombies = int(input("Enter the number of zombies: "))
if zombies > 10:
    print("There's just half a Minecraft zombie apocalypse.")
elif zombies > 20:
    print("Ahhhh! Zombies!")
elif zombies == 0:
    print("No zombies here! Phew!")
else:
    print("You know, you zombies aren't so bad.")
```

Was läuft hier falsch? Spielen wir doch einmal durch, was passiert, wenn `zombies` zum Beispiel den Wert 22 hat. Da 22 größer als 10 ist, ist die Bedingung der ersten `if`-Anweisung (`zombies > 10`) `True`. Folglich wird der Code der `if`-Anweisung ausgeführt. Danach wird keine weitere `elif`- oder `else`-Anweisung mehr ausgeführt. Das Programm erreicht also niemals `elif zombies > 20`, denn der `if`-Anweisungskörper wurde ja bereits ausgeführt. Das ist ein Bug.

Wenn du bei der Ausführung von `if`-Anweisungen unerwartete Ergebnisse erhältst, musst du immer überprüfen, ob die Reihenfolge deiner `if`- und `elif`-Anweisungen richtig ist.

Mission 29: Teleportation zum gewünschten Ort

Wenn deine `if`- und `elif`-Anweisungen in der falschen Reihenfolge aufgeführt sind, funktioniert Code nicht, der eigentlich funktionieren müsste. Schlimmer noch: Code, der nicht funktionieren dürfte, funktioniert auf einmal! Auf diese Weise gelangen schwierig zu analysierende Bugs in deine Programme. Zur Behebung solcher Probleme musst du also alle Bedingungen in die richtige Reihenfolge setzen. Versuchen wir es einmal.

Listing 6.3 funktioniert nicht. Eigentlich soll der Spieler damit je nachdem, wie viele Punkte er eingibt, an verschiedene Positionen teleportiert werden. Die Punkte führen zwar zu den gewünschten Positionen, doch hat es den Anschein, dass die Reihenfolge der Bedingungen nicht richtig ist.

Je mehr Punkte ein Spieler hat, desto vorteilhafter ist die Zielposition. Hier ist der Code dazu. Die Bedingungen werden für jeden Transportvorgang mit `setPos()` festgelegt.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

points = int(input("Enter your points: "))
if points > 2:
    mc.player.setPos(112, 10, 112)
elif points <= 2:
    mc.player.setPos(0, 12, 20)
elif points > 4:
    mc.player.setPos(60, 20, 32)
elif points > 6:
    mc.player.setPos(32, 18, -38)
else:
    mc.postToChat("I don't know what to do with that information.")
```

teleportScore.py

Listing 6.3: Je nach Anzahl deiner Punkte wirst du an einen anderen Standort teleportiert.

Es gibt unterschiedliche Bedingungen für Werte größer 6, größer 4, größer 2 und 2 oder weniger Punkte.

Die letzte Zeile in der `else`-Anweisung wird nur dann ausgeführt, wenn der Benutzer etwas Abwegiges – beispielsweise einen Text anstelle einer Punktzahl – oder aber gar nichts eingibt.

Erstelle eine neue Datei in IDLE, und speichere sie als *teleportScore.py* im Ordner *ifStatements*. Passe das Programm so an, dass die Bedingungen in der richtigen Reihenfolge überprüft werden und alle Positionen erreicht werden können. Teste das Programm mit unterschiedlichen Eingaben für die Punkte, um sicherzustellen, dass der Code für jedes Teleportationsziel ausgeführt werden kann. Bild 6.4 zeigt eine Programmversion, die nicht funktioniert.



Bild 6.4: Hier wollte ich jetzt eigentlich nicht hin ...

Weil das Programm nicht wie gewünscht arbeitet, werde ich bei der Eingabe von 5 an die Position für einen Punktwert > 2 gebracht, obwohl ich doch logischerweise zur Position für den Wert > 4 müsste.

Bonus: Beam mich rauf, Scotty

Erstelle ein Programm, mit dem du den Standort, an den du teleportiert werden möchtest, als String angibst (z. B. "castle"). Mit if- und elif-Anweisungen kannst du dann festlegen, wohin du teleportiert werden möchtest. So bringt dich "sea fortress" an einen bestimmten Ort und "tree house" an einen anderen.

Verschachtelte if-Anweisungen

Angenommen, du hast eine if-Anweisung, deren Bedingung True ist, und möchtest eine weitere Bedingung testen (und Code ausführen, sofern auch diese zweite Bedingung True ist). Vielleicht möchtest du den Zugang zum Geheimversteck in deiner Basis ermöglichen und schreibst zu diesem Zweck einen Code, der überprüft, ob du auf einem Schalter stehst. Ist dies der Fall, dann überprüft eine weitere Codezeile, ob du das geheime Item in der Hand hast, mit dem das Schloss geöffnet werden kann. Wie könnte man das bewerkstelligen?

Nun, du kannst eine if-Anweisung in den Anweisungskörper einer anderen if-Anweisung ablegen. Dies bezeichnet man als *verschachtelte* if-Anweisung.

Listing 6.4 zeigt ein Beispiel für eine verschachtelte if-Anweisung. Ein Geldautomat überprüft zunächst, ob du genug Geld auf dem Konto hast, und fordert dich, falls dies der Fall ist, dazu auf, die Abhebung zu bestätigen. Tust du dies, wird die Abhebung durchgeführt.

```
withdraw = int(input("Wie viel möchtest du von deinem Konto abheben?
"))
balance = 1000

if balance >= withdraw: ❶
    confirm = input("Bist du sicher? ")
    if confirm == Ja: ❷
        print("Hier ist dein Geld.")
else:
    print("Schade, du hast nicht genug Geld auf dem Konto.")
```

Listing 6.4: Ein Geldautomat in Python

Du wirst bemerken, dass die zweite if-Anweisung innerhalb der ersten if-Anweisung eingerückt ist. Wenn die Bedingung ❶ der äußeren if-Anweisung True ist, dann hast du genug Geld auf dem Konto, und die Zeile `confirm = input("Are you sure? ")` wird ausgeführt. Dann wird, falls die Bedingung ❷ der inneren if-Anweisung True ist, der Code "Here is your money" ausgegeben.

Mission 30: Eine Geheimtür öffnen

In dieser Mission wollen wir das vorherige Beispiel ein wenig ausbauen. Du erstellst hier ein Gebäude mit einem Geheimgang, der sich nur dann öffnet, wenn ein Diamantblock auf einem Sockel steht. Wird hingegen irgendein anderer Block auf dem Sockel abgestellt, dann verwandelt sich der ganze Boden in glühende Lava!

Als Erstes brauchen wir ein Gebäude. Dafür verwenden wir das Programm *building.py* (Seite 70) im Ordner *math*. Füge keine Tür zum Gebäude hinzu. Stattdessen platzierst du dort, wo der Eingang sein soll, einen einzelnen Block. Dieser ist unser Sockel. Positionierst du einen Diamantblock auf diesem Sockel, dann öffnet der Code eine Geheimtür, durch die du ins Gebäudeinnere gelangst. Listing 6.5 zeigt ein Codegerüst, das du als Ausgangspunkt verwenden kannst.

secretDoor.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

x = 10
y = 11
z = 12
gift = mc.getBlock(x, y, z)
if gift != 0:
    # Hier eigenen Code einfügen
else:
    mc.postToChat("Bitte lege ein Geschenk auf den Sockel.")
```

Listing 6.5: Der Anfang des Codes zum Öffnen einer Geheimtür, wenn du ein Geschenk auf einem Sockel ablegst

Erstelle eine neue Datei in IDLE, und speichere sie als *secretDoor.py* im Ordner *ifStatements*. Ändere die Koordinaten in diesem Programm auf die Position ab, an der in deiner Minecraft-Welt der Diamantblock platziert werden muss, um die Tür zu öffnen.

Kopiere Listing 6.5, und füge Code für folgende Aufgaben hinzu:

- Wenn auf dem Sockel ein Diamantblock (ID 57) liegt, öffne eine Geheimtür in den verborgenen Raum. (Kleiner Tipp: Um eine Öffnung im Gebäude zu erstellen, musst du die entsprechenden Blöcke in Luft umwandeln.)
- Wenn ein Block, der kein Diamant ist, auf dem Sockel abgestellt wird, verwandle den Boden unter dem Spieler in Lava (ID 10).

Um diese Anforderungen umzusetzen, benötigst du eine verschachtelte *if*-Anweisung.

Da dieses Programm bereits recht komplex ist, solltest du es in Phasen erstellen und testen. Wenn du eine Funktion hinzugefügt hast, starte das Programm und vergewissere dich, dass der betreffende Programmteil funktioniert, bevor du fortfährst. Das Debuggen kürzerer Codeabschnitte ist einfacher als bei umfangreichen Codes. Bild 6.5 zeigt den geheimen Zugang.



Bild 6.5: Die Geheimtür zum Tempel ist nun geöffnet.

Bonus: Fahrstuhl

Aus dem Programm *secretDoor.py* lässt sich einiges machen. Wie wäre es etwa mit einer automatischen Tür, die erkennt, wenn der Spieler davor steht? Oder mit einem Aufzug, der einen Spieler automatisch in ein höher gelegenes Stockwerk bringt?

Mit if-Anweisungen einen Wertebereich testen

Wie wir in Kapitel 5 erfahren haben, lässt sich in Python feststellen, ob ein Wert zwischen zwei anderen Werten liegt. Da eine Bereichsprüfung als `True` oder `False` ausgewertet wird, können wir eine solche Prüfung ebenso als Bedingung für eine `if`-Anweisung verwenden, wie wir es von Größer-als- oder Kleiner-als-Vergleichen oder von Vergleichen auf Gleichheit/Ungleichheit her kennen. Jeder Ausdruck, der als `True` oder `False` ausgewertet werden kann, kann als Bedingung für eine `if`-Anweisung verwendet werden.

Angenommen, du möchtest diese ganz vorzüglichen Minecraft-Kuchen backen und hast den ganzen Tag mit dem Sammeln von Zutaten verbracht. Dabei hast du genügend Zutaten für 30 Kuchen gefunden. Diese Kuchen möchtest du nun verkaufen. Wer auch immer dir Kuchen abkauft,

muss mindestens einen, aber weniger als 30 Kuchen kaufen, sonst verkaufst du nicht. Schließlich soll nicht der ganze gute Kuchen in ein und demselben Magen landen!

Der folgende Code beschreibt die Situation:

```
cakes = int(input("Enter the number of cakes to buy: "))
if 0 < cakes < 30: ❶
    print("Here are your " + str(cakes) + " cakes.")
elif cakes == 0: ❷
    print("Don't you want some delicious cake?")
else: ❸
    print("That's too many cakes! Don't be selfish!")
```

Wenn die Variable `cakes` einen Wert zwischen 0 und 30 hat – beispielsweise 15 –, wird "Here are your 15 cakes" ausgegeben. ❶. In jedem anderen Fall wird eine andere Meldung angezeigt. Hat `cakes` nun den Wert 0, dann zeigen wir "Don't you want some delicious cake?" ❷ an, und bei einem Wert größer 30 lautet die Meldung "That's too many cakes! Don't be selfish!" ❸.

Einen komplexeren Ausdruck können wir durch Hinzufügen eines booleschen Operators testen. Wenn ich jetzt aus Gründen, die sich nur mir erschließen, Brotlaibe nicht in einer Größenordnung zwischen 20 und 30 verkaufen möchte, kann ich das mit dem folgenden `not`-Operator umsetzen:

```
bread = int(input("Enter the amount of bread: "))
if not 20 <= bread <= 30:
    print("Here are your " + bread + " breads.")
else:
    print("I don't sell that amount of bread for some reason.")
```

Hier teste ich mit dem `not`-Operator und führe einen Größer-oder-gleich-Test aus, um als erste Bedingung einen Wertebereich zu prüfen. Hiermit kann ich feststellen, ob die Menge des Brots, das jemand kaufen möchte, zwischen 20 und 30 liegt. Danach macht `not` aus einem `True` ein `False` und umgekehrt. Liegt also `bread` im gewünschten Bereich, dann wird der Gesamtausdruck als `False` ausgewertet und folglich der Code in der `else`-Anweisung ausgeführt. Liegt `bread` hingegen nicht im Bereich zwischen 20 und 30 (sondern beispielsweise bei 40), dann ist der gesamte Ausdruck `True` und wir geben "Here are your 40 breads" aus.

Jemand möchte 23 Brote bei mir kaufen? Keine Chance! Aber 17 oder 32 Brote sind okay.

Mission 31: Teleportationsziele beschränken

Erinnerst du dich an das Programm aus Kapitel 2 namens *teleport.py*? In dieser Mission verwendest du Bereichsprüfungen und *if*-Anweisungen, um die möglichen Zielorte für eine Teleportation zu beschränken. Wenn du Minecraft auf dem Raspberry Pi einsetzt, gibt es Orte außerhalb der Spielwelt, die nicht existieren, die du aber durch programmierte Teleportation trotzdem aufsuchen kannst. In der Desktop-Edition von Minecraft ist deine Welt viel größer, d. h., sie unterliegt nicht den Beschränkungen der Pi-Edition. Trotzdem ist dieses Programm nützlich. Du kannst etwa, wenn du mit anderen Spielern Verstecken spielst, den Bereich begrenzen, in dem sich die Spieler verstecken können.

Listing 6.6 soll die Koordinaten *x*, *y* und *z* der Eingabe des Benutzers entnehmen und ihn dann an die gewünschte Position teleportieren. Allerdings ist das Programm noch nicht vollständig.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
valid = True

x = int(input("Enter x: "))
y = int(input("Enter y: "))
z = int(input("Enter z: "))

if not -127 < x < 127:
    valid = False

# Überprüfen, ob y zwischen -63 und 63 liegt

# Überprüfen, ob z zwischen -127 und 127 liegt

if valid:
    mc.player.setPos(x, y, z)
else:
    mc.postToChat("Please enter a valid location")
```

teleportLimit.py

Listing 6.6: Programm zum Beschränken von Teleportationsmöglichkeiten

Um die möglichen Teleportationsziele des Spielers einzuschränken, erstellen wir zunächst eine Variable namens *valid*. Diese stellt durch ihren Wert (*True* oder *False*) dar, ob alle Koordinaten des angegebenen Ziels gültig sind. Dann wird der Benutzer aufgefordert, Werte für *x*, *y* und *z* einzugeben. Danach lassen wir eine *if*-Anweisung prüfen, ob die Variable *x* zwischen -127 und 127 liegt. Tut sie dies nicht, dann ist die *x*-Koordinate ungültig, und die Variable *valid* wird auf *False* gesetzt.

Erreicht das Programm die letzte if-Anweisung, dann wird `setPos()` nur dann aufgerufen, wenn `valid` den Wert `True` hat. Dies wiederum ist aber nur der Fall, wenn alle drei Bedingungen erfüllt wurden, ansonsten wird der Spieler nicht teleportiert, sondern per Chat aufgefordert, eine gültige Position einzugeben.

Erstelle eine neue Datei in IDLE, und kopiere Listing 6.6 dorthin. Speichere das Programm als *teleportLimit.py* im Ordner *ifStatements*.

Vervollständige das Programm mit den erforderlichen if-Anweisungen und Bereichsprüfungen für die Variablen `y` und `z`, damit `valid` auf `False` gesetzt wird, wenn die Werte ungültig sind.

Wenn du glaubst, dass alles funktioniert, führe das Programm aus. Es sollte dich nun teleportieren, sofern du Werte eingegeben hast, die für `x` und `z` im Bereich zwischen `-127` und `127` und für `y` im Bereich zwischen `-63` und `63` liegen. Gibst du einen Wert außerhalb dieser Bereiche ein, dann sollte das Programm dich nicht teleportieren. Bild 6.6 zeigt eine Situation, in der der Benutzer eine ungültige Zahl eingegeben hat.

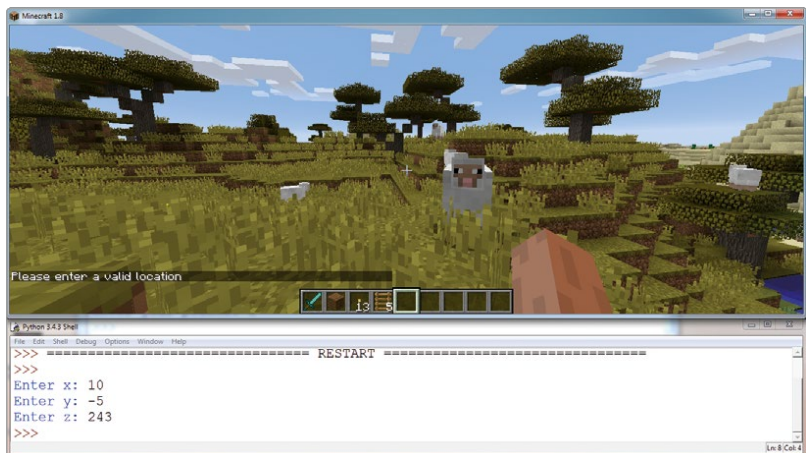


Bild 6.6: *z ist zu groß – keine Teleportation*

Bonus: Oben bleiben

Eines der Probleme beim Teleportationsprogramm besteht darin, dass du unter die Erde teleportiert werden kannst. Und dort bist du dann gefangen. Ändere das Programm doch einmal ab, damit so etwas nicht mehr möglich ist. Dazu vergleichst du die y-Koordinate, die der Benutzer in die Funktion `getHeight()` eingibt, um dafür zu sorgen, dass der Spieler sich an einen Ort über der Erde teleportiert.

Boolesche Operatoren und if-Anweisungen

Weiter oben haben wir `not` in `if`-Anweisungen verwendet. Genauso gut kannst du `and` und `or` einsetzen. In diesem Fall agiert die `if`-Anweisung genau so wie bei nur einer einfachen Bedingung: Wenn der Gesamtausdruck `True` ist, wird der Anweisungskörper ausgeführt. Hier folgt ein Programm, mit dem jemand gefragt wird, ob er Kuchen hat und uns ggf. etwas davon abgeben würde:

```
hasCake = input("Do you have any cake? Y/N")
wouldShare = input("Would you give me some cake? Y/N")

if hasCake == "Y" and wouldShare == "Y":
    print("Yay!")
else:
    print("Boo!")
```

Dieser Code verwendet den `and`-Operator, d. h., Python gibt nur dann "Yay!" aus, wenn die Person Kuchen hat (`hasCake == "Y"` ist `True`) und diesen zu teilen bereit ist (`wouldShare == "Y"` ist `True`). Wenn einer dieser Vergleiche nicht `True` ist, gibt der Code in der `else`-Anweisung "Boo!" aus.

Du kannst `and` durch den `or`-Operator ersetzen, wenn Python "Yay!" ausgeben soll, wenn der Befragte entweder Kuchen hat oder diesen zu teilen bereit ist:

```
hasCake = input("Do you have any cake? Y/N")
wouldShare = input("Would you give me some cake? Y/N")

if hasCake == "Y" or wouldShare == "Y":
    print("Yay!")
else:
    print("Boo!")
```

Wenn entweder `hasCake == "Y"` oder `wouldShare == "Y"` den Wert `True` hat, wird der gesamte Ausdruck als `True` ausgewertet, und es wird "Yay!" ausgegeben. "Boo!" hingegen erscheint nur dann, wenn keine Bedingung den Wert `True` hat: Der Angesprochene hat keinen Kuchen und würde ihn andernfalls sowieso nicht teilen.

Verwenden wir einmal den `not`-Operator mit einer `if`-Anweisung:

```
wearingShoes = input("Are you wearing shoes? Y/N")
if not wearingShoes == "Y":
    print("You're not wearing shoes.")
```

Dieses Programm fordert den Benutzer auf, Y einzugeben, wenn er Schuhe trägt, und N, wenn er dies nicht tut. Die Eingabe wird in `wearingShoes` gespeichert. Daraufhin erfolgt ein Vergleich zwischen `wearingShoes` und "Y", um festzustellen, ob diese beiden Werte gleich sind. Der `not`-Operator invertiert das Ergebnis eines Vergleichs: `True` wird `False` und `False` wird `True`. Wenn also der Benutzer Y eingegeben hat, steht am Anfang des Vergleichs `True`, und `not` macht daraus – und damit auch aus dem gesamten Ausdruck – `False`. Wir geben keine Meldung aus. Hat der Benutzer etwas anderes als Y eingegeben, dann wird der Vergleich `False` und nachfolgend durch `not` zu `True`. Der Gesamtausdruck wird als `True` ausgewertet, und folglich wird "You're not wearing shoes" ausgegeben.

Mission 32: Dusche gefällig?

Die besten Minecraft-Häuser zeichnen sich durch große Liebe zum Detail aus. Viele Leute verlegen Holzböden, bauen offene Kamine und hängen Bilder an die Wand, um sich ein wirklich gemütliches Plätzchen zu schaffen. Wir toppen das noch, denn wir werden eine funktionsfähige Dusche bauen.

Damit das klappt, brauchen wir eine Anzahl von Bereichsüberprüfungen und booleschen Operationen. Du wirst zunächst einen Duschbereich aufbauen. Wenn der Spieler in die Dusche steigt, schaltet sich das Wasser automatisch ein. Anders formuliert: Wenn der Spieler einen bestimmten Koordinatenbereich betritt, soll das Programm Wasserblöcke über ihm erstellen.

Listing 6.7 zeigt die Grundstruktur des Programms und enthält bereits ein paar Codezeilen als Ausgangspunkt. Deine Aufgabe besteht nun darin, den Code sinnvoll zu erweitern.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

shwrX = 1
shwrY =
shwrZ =

width = 5
height = 5
length = 5

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z

if shwrX <= x < shwrX + width and
    mc.setBlocks(shwrX, shwrY + height, shwrZ,
                 shwrX + width, shwrY + height, shwrZ + length, 8)
else:
    mc.setBlocks(shwrX, shwrY + height, shwrZ,
                 shwrX + width, shwrY + height, shwrZ + length, 0)
```

shower.py

Listing 6.7: Gerüst des Duschenprogramms

Kopiere Listing 6.7, und speichere es als *shower.py* im Verzeichnis *ifStatements*.

Um das Programm zu vervollständigen, fügst du die Koordinaten deiner Dusche in die Variablen *shwrX*, *shwrY* und *shwrZ* ❶ ein. Danach legst du die Größe der Dusche mit den Koordinaten *width*, *height* und *length* ❷ fest. Hier habe ich als Wert 5 vorgegeben, aber du kannst die Größe gern an deinen persönlichen Geschmack anpassen.

Ergänze dann die *if*-Anweisung so, dass überprüft wird, ob die Variablen *y* und *z* sich im Bereich der Dusche befinden ❸. Um dir ein bisschen unter die Arme zu greifen, habe ich die Überprüfung für die *x*-Position bereits vorgenommen (*shwrX* < *x* < *shwrX* + *width*). Die Ausdrücke für die Positionen *y* und *z* sehen ganz ähnlich aus. Hinweis: Du solltest diese Überprüfungen mit *and* kombinieren.

Ein- und ausgeschaltet wird die Dusche mit der Funktion *setBlocks()* ❹. Diese Blöcke werden beim Einschalten auf Wasser (Block-ID 8) und beim Ausschalten auf Luft (ID 0) gesetzt.

Die *setBlocks()*-Funktionen in der letzten *if/else*-Anweisung sind auf zwei Zeilen verteilt, weil ihre Argumente sehr lang sind. In Python ist das möglich. Ich hätte sie auch in eine einzelne Zeile schreiben können, aber so sind sie einfach besser zu lesen.

Bild 6.7 zeigt meine funktionsfähige Dusche.

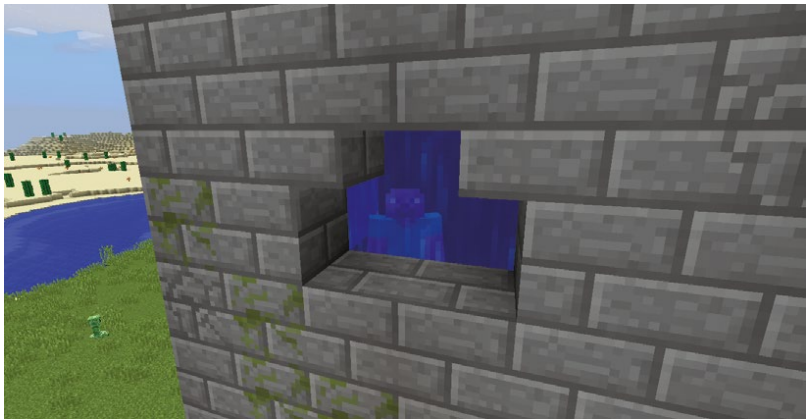


Bild 6.7: Ich – unter der Dusche

Wenn du das Programm ausführst, erstellt es Wasser über dir, sobald du in der Dusche stehst. Der Wasserfluss endet erst, wenn du die Dusche verlässt und das Programm erneut ausführst. Viel Spaß!

Bonus: Wasser sparen

Programmiere einmal eine Zeitbeschränkung, damit deine Dusche nicht die ganze Zeit läuft.

Was du gelernt hast

Deine Programme können jetzt Entscheidungen treffen. Du hast in diesem Kapitel gelernt, wie Bedingungen in Verbindung mit `if`-, `else`- und `elif`-Anweisungen verwendet werden. In Kapitel 7 wirst du *while-Schleifen* kennenlernen. Wie `if`-Anweisungen helfen auch `while`-Schleifen deinem Programm zu entscheiden, was wann getan werden muss. Allerdings führen `while`-Schleifen Code aus, solange eine Bedingung erfüllt ist – anders als `if`- und `else`-Anweisungen, mit denen du jeweils unterschiedlichen Code ausführst, abhängig davon, ob eine Bedingung wahr ist oder nicht. Erst, wenn die Bedingung nicht mehr erfüllt ist, endet eine solche Schleife.

Kapitel 7

Tanztees und Blumenkorsos mit `while`-Schleifen

Schleifen sind dazu da, einen Codeabschnitt immer und immer wieder auszuführen. Anstatt also denselben Codeabschnitt über die Zwischenablage zu kopieren, kannst du ihn beliebig oft ausführen. Wir werden in diesem Kapitel mit Schleifen dafür sorgen, dass deine Programme wiederholt werden, ohne neu gestartet werden zu müssen. Dabei konzentrieren wir uns auf einen speziellen Schleifentyp in Python: die `while`-Schleife.

Eine einfache `while`-Schleife

Mit `while`-Schleifen kannst du Codeabschnitte wiederholen. Ähnlich wie die `if`-Anweisung führt eine `while`-Schleife den enthaltenen Code aus, solange eine Bedingung `True` ist. Folglich muss die Bedingung erfüllt sein, damit der Anweisungskörper ausgeführt wird.

Der Unterschied zwischen einer `while`-Schleife und einer `if`-Anweisung besteht darin, dass der Code in der `if`-Anweisung höchstens einmal durchlaufen wird, während dies bei der `while`-Schleife mehrfach der Fall ist. Programmierer nennen eine solche Codewiederholung *Iteration*. Wenn eine Schleife wiederholt wird, spricht man davon, dass sie *iteriert*.

Der folgende Code beispielsweise verwendet eine `while`-Schleife und gibt mit ihr die Zahlen 1 bis 5 aus:

```
count = 1
while count <= 5:
    print(count)
    count += 1
print("Loop finished")
```

Die Variable `count` zählt, wie häufig die Schleife wiederholt wurde. Sie beginnt beim Wert 1. Die Bedingung in der `while`-Schleife überprüft, ob die Anzahl kleiner als oder gleich 5 ist.

Hinweis

In Kapitel 3 hast du gelernt, dass `+=` ein verkürzter Operator ist. Du könntest stattdessen auch den normalen Additionsoperator `count = count + 1` verwenden.

Wenn die Schleife zum ersten Mal ausgeführt wird, lautet der Wert von `count` 1. Er ist also kleiner als 5. Die Bedingung der Schleife hat den Wert `True`, und der Schleifenkörper wird ausgeführt. Danach gibt das Programm den Wert von `count` in der Python-Shell aus und addiert anschließend 1 zu diesem Wert hinzu. Die `while`-Schleife wird nun wiederholt und überprüft die Bedingung erneut. Dabei wird jeder Schritt verarbeitet, bis der Wert von `count` größer als 5 wird.

Nach dem Ende der Schleife folgt eine abschließende Zeile, die `"Loop finished"` ausgibt.

Speichere das Programm, und führe es aus. Die folgende Ausgabe sollte erscheinen:

```
1
2
3
4
5
Loop finished
```

Experimentiere ein bisschen mit diesem Code. Ändere die Bedingungen ab, damit mehr als fünf Zahlen ausgegeben werden, oder wähle einen anderen Wert, der zur Variablen `count` hinzuaddiert wird. Hier ist noch eine kurze Beschreibung, wie dieser Code funktioniert. In der `while`-Anweisung werden die folgenden Schritte abgearbeitet:

1. Überprüfe, ob die Bedingung `True` ist.
2. Wenn die Bedingung `True` ist:
 - a. Führe den Anweisungskörper aus.
 - b. Gehe wieder zu Schritt 1.
3. Wenn die Bedingung `False` ist:
 - a. Ignoriere den Anweisungskörper.
4. Fahre mit der Zeile nach dem Ende der `while`-Schleife fort.

Wir wollen eine `while`-Schleife einmal in Minecraft verwenden, um viele unbekannte Orte zu besuchen!

Mission 33: Teleportationstour auf Zufallsbasis

In Mission 3 (Seite 51) hast du den Spieler an mehrere unterschiedliche Standorte im Spiel teleportiert. Wir wollen dieses Programm mit einer `while`-Schleife neu schreiben, sodass du die Teleportation immer und immer wieder wiederholen kannst.

Wenn du Code, der den Spieler an eine zufällige Position teleportiert, in einer Schleife ausführst, kannst du dieses Programm noch leistungsfähiger und gleichzeitig *deutlich* verständlicher machen. Praktisch, nicht?

Der folgende Code teleportiert den Spieler an einen zufälligen Ort, indem er den Variablen `x`, `y` und `z` zufällige Werte innerhalb der Spielwelt zuweist. Danach verwendet er diese Variablen, um die Position des Spielers festzulegen.

```
import random
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

# Zählervariable hier hinzufügen ❶
# while-Schleife hier starten ❷
x = random.randint(-127, 127) # Code ab dieser Zeile einziehen ❸
y = random.randint(0, 64)
z = random.randint(-127, 127)

mc.player.setTilePos(x, y, z)
# Zählervariable um 1 hochzählen ❹
```

Bis jetzt teleportiert der Code den Spieler allerdings nur genau einmal. Das ist schon ziemlich fantastisch, aber es geht noch viel besser! Wir wollen eine Schleife schreiben, damit der Code fünfmal ausgeführt wird – so wird das Ganze zur Achterbahnfahrt.

Um die Schleife in den Code zu integrieren, musst du die folgenden vier Schritte ausführen:

1. Erstelle eine Variable `count` zur Steuerung der Schleife ❶.
2. Füge eine `while`-Schleife mit einer Bedingung hinzu, die auf `count` basiert ❷.
3. Rücke den `while`-Anweisungskörper ein ❸.
4. Zähle den Wert von `count` bei jedem Schleifendurchlauf hoch ❹.

Wir verwenden die Variable `count` und zählen sie hoch, um festzuhalten, wie häufig die Schleife bereits wiederholt wurde. Ich werde im nächsten Abschnitt näher darauf eingehen. An dieser Stelle musst du lediglich wissen, dass wir mit `count` im Blick behalten, wie häufig der Code bereits wiederholt wurde.

Listing 7.1 zeigt den Code mit den erforderlichen Änderungen.

random
Teleport.py

```
import random
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

count = 0
while count < 5:
    x = random.randint(-127, 127)
    y = random.randint(0, 64)
    z = random.randint(-127, 127)

    mc.player.setTilePos(x, y, z)
    count += 1
```

Listing 7.1: Code zum zufälligen Teleportieren des Spielers in der Spielwelt

Kopiere Listing 7.1 in eine neue Datei, speichere sie unter dem Namen *randomTeleport.py* in einem neuen Ordner namens *whileLoops*, und führe den Code aus. Nun sollte der Spieler in der Minecraft-Welt hin und her springen. Allerdings läuft der Code viel zu schnell! Die ganze Reise dauert weniger als eine Sekunde. Da müssen wir Abhilfe schaffen.

Mit dem `time`-Modul kannst du den Code verlangsamen. Führe dazu die folgenden Schritte aus:

1. Füge in der ersten Programmzeile die Anweisung `import time` hinzu. Hierdurch wird das `time`-Modul von Python importiert, das eine Anzahl praktischer Funktionen zur Zeitsteuerung enthält.

2. Füge die Zeile `time.sleep(10)` am Ende des `while`-Schleifenkörpers hinzu, um an jedem Ort, zu dem du teleportiert wirst, 10 Sekunden zu verweilen. Vergiss nicht, diese neue letzte Zeile deines Programms einzuziehen, damit sie sich innerhalb der `while`-Schleife befindet!

Speichere das Programm, und führe es aus. Nun sollte der Spieler alle 10 Sekunden zu einem neuen, zufällig ausgewählten Ort teleportiert werden. Bild 7.1 zeigt mein Programm während der Ausführung.



Bild 7.1: Alle 10 Sekunden an einem neuen unbekannten Ort. Fantastisch!

Bonus: Schlafe ruhig, schlafe sacht

Im Moment wartet das Programm am Ende jeder Schleife 10 Sekunden lang. Was würde passieren, wenn du die Anweisung `time.sleep(10)` an den Anfang der Schleife setzen würdest?

Schleifen mit einer Zählervariablen steuern

Zählervariablen stellen eine oft verwendete Möglichkeit dar, die Häufigkeit der Wiederholung eines Codes zu speichern. Wir haben diese Variablen bereits einige Male in Aktion gesehen. Betrachten wir ein weiteres Beispiel:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Die Bedingung der `while`-Schleife überprüft, ob der Wert der Variablen `count` kleiner als 5 ist. Ich habe den Wert von `count` im Schleifenkörper so geändert, dass er die Anzahl der Wiederholungen vermerkt. Das Hinzufügen eines Werts zu einer Zählervariablen heißt *Inkrementieren*.

Die letzte Codezeile erhöht den Wert von `count` jeweils um 1. Bei jeder Wiederholung des Codes wird der neue Wert der Variablen `count` abgefragt, um festzustellen, ob sie immer noch kleiner als 5 ist. Ist die Variable gleich oder größer als 5, dann endet die Abarbeitung der Schleife.

Wenn du vergisst, die Variable `count` hochzuzählen, dann erstellst du eine sogenannte *Endlosschleife*, d. h., die Schleife wird auf ewig wiederholt. Das folgende Beispiel zeigt dies:

```
count = 0
while count < 5:
    print(count)
```

Der Wert von `count` ist immer 0, da er nie hochgezählt wird. Deswegen ist die Bedingung der Schleife immer `True`, und die Schleife endet *nie*. Du glaubst mir nicht? Probier's aus!

```
0
0
0
0
0
-- usw. --
```

Um die Ausführung dieses endlosen Programms zu unterbrechen, drückst du **Strg+C**. Danach kannst du den Code berichtigen, indem du einfach die Zeile `count += 1` im Schleifenkörper hinzufügst. So bleibst du nicht in der Endlosschleife gefangen. Schwein gehabt!

Zählervariablen müssen nicht immer um 1 hochgezählt werden. Es kann durchaus Situationen geben, in denen eine Erhöhung um einen anderen Wert sinnvoller ist. Im folgenden Beispiel wird die Zählervariable jedes Mal um 2 hochgezählt. Im Ergebnis gibt der Code alle geraden Zahlen zwischen 0 und 100 aus:

```
count = 0
while count < 100:
    print(count)
    count += 2
```

Du kannst den Wert der Zählervariablen aber auch herunterzählen (*dekrementieren*), indem du eine negative Schrittweite angibst. Der folgende Code zählt von 100 auf 0 *herunter*:

```
count = 100
while count > 0:
    print(count)
    count -= 1
```

Der einzige Unterschied zwischen diesem Beispiel und den vorherigen betrifft die Bedingung. Hier habe ich als Vergleichsoperator das Größer-als-Zeichen (`>`) verwendet. Solange die Zählervariable größer als 0 ist, wird die Schleife weiter wiederholt; erst wenn der Wert 0 erreicht wird, endet sie.

Hinweis

Die zur Steuerung einer Schleife verwendete Variable muss natürlich nicht immer *count* heißen. Man kann sie auch *repeats* oder sonst wie nennen. In den Codes anderer Programmierer wirst du eine Vielzahl von alternativen Bezeichnungen dafür finden.

Mission 34: Der Wasserfluch

Jetzt wollen wir einmal etwas gemein werden und einen Fluch schreiben, der den Spieler eine Zeit lang irritieren soll. Normalerweise *schwächen* Flüche in Videospielen den Spieler auf irgendeine Weise, machen ihn langsamer oder schwächer – oft auch nur vorübergehend.

Wir werden einen Fluch erschaffen, der 30 Sekunden lang einmal pro Sekunde einen Block fließenden Wassers an der Spielerposition platziert. Dies macht es für den Spieler schwieriger, sich zu bewegen, ohne von dem reißenden Strom durchgespült zu werden.

Der folgende Code sorgt für einen Block mit fließendem Wasser an der Spielerposition:

waterCurse.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

pos = mc.player.getPos()
mc.setBlock(pos.x, pos.y, pos.z, 8)
```

Hiermit wird der Wasserblock jedoch nur einmal an der Position des Spielers platziert. Für die Wiederholungen bist du zuständig. Der fertige Code sollte sich 30-mal wiederholen, und jede Iteration der Schleife sollte mindestens eine Sekunde lang dauern.

Speichere diesen Code als *waterCurse.py* im Verzeichnis *whileLoops*, und führe ihn einmal aus, um dich zu vergewissern, dass er funktioniert. Vor dem Ende des Programms solltest du einen einzelnen Wasserblock an der Spielerposition sehen.

Jetzt beschreibe ich noch kurz, was du außerdem hinzufügen musst, damit dieser Fluch länger wirkt. Nutze dein Wissen über *while*-Schleifen und Zählervariablen, um folgende Schritte durchzuführen:

1. Füge eine Variable `count` zum Programm hinzu.
2. Erweitere das Programm mit einer Schleife, die die letzten beiden Codezeilen wiederholt. Die Schleife selbst sollte sich ihrerseits 30-mal wiederholen.
3. Zähle die Variablen `count` am Ende der Schleife hoch.
4. Importiere das `time`-Modul (in die erste Zeile deines Programms), und füge nach der letzten Zeile der `while`-Schleife eine Pause hinzu, die eine Sekunde dauert.

Speichere das Programm, und teste es dann. Wenn du in der Spielwelt unterwegs bist, sollte dieses Programm 30 Sekunden lang in jeder Sekunde einen Wasserblock erstellen. Falls du mit dem Code nicht zurechtkommst, sieh dir noch einmal die Schritte in Mission 33 (Seite 163) an.

Bild 7.2 zeigt den Fluch in Aktion.



Bild 7.2: Oh nein! Das Wasser verfolgt mich!

Bonus: Das Wasser steigt schneller

Wie könntest du dafür Sorge tragen, dass die Schleife sich doppelt so schnell wiederholt (d. h. jede halbe Sekunde) und trotzdem 30 Sekunden dauert?

while-Endlosschleifen

Meistens ist es sehr wichtig, dass die boolesche Bedingung in deiner while-Schleife früher oder später False wird, denn sonst iteriert die Schleife endlos und der Computer könnte abstürzen.

Es gibt aber auch Fälle, in denen eine solche Endlosschleife tatsächlich benötigt wird. In Games beispielsweise wird eine solche Schleife häufig verwendet, um Benutzereingaben abzufragen und die Bewegung des Spielers zu verwalten. Natürlich enthalten solche Spiele Möglichkeiten (z. B. Schaltflächen), mit denen du die Endlosschleifen unterbrechen oder beenden kannst, wenn du eine Pause brauchst.

Eine einfache Möglichkeit, eine Endlosschleife zu erstellen, ist die Verwendung einer True-Bedingung in der Definition einer while-Schleife. Das könnte etwa wie folgt aussehen:

```
while True:
    print("Hello")
```

Dieser Code gibt fortlaufend den String "Hello" aus. Absichtlich wie unabsichtlich erstellte Endlosschleifen lassen sich jederzeit abbrechen, wenn man **Strg+C** in der Python-Shell drückt. In IDLE kannst du die Schleife unterbrechen, indem du **Shell ▶ Restart Shell** auswählst.

Denke daran, dass sämtlicher Code, der *nach* einer while-Endlosschleife platziert ist, niemals ausgeführt wird. Im nächsten Beispiel ist die letzte Codezeile aufgrund der vorangehenden while-Endlosschleife unerreichbar:

```
while True:
    print("Hello")
print("This line is never reached")
```

Zwar können Endlosschleifen manchmal etwas kompliziert sein, aber man kann damit auch eine ganze Menge coole Sachen anstellen. Und genau das wollen wir jetzt ausprobieren.

Mission 35: Der Blumenkorso

Das Programm, das wir in dieser Mission schreiben wollen, ähnelt dem aus Mission 34. Anstatt allerdings Wasserblöcke aufzustellen, sorgen wir dafür, dass hinter dem Spieler eine Spur aus Blumen erscheint. Blumen sind schließlich viel schöner als stürmische Wogen!

Öffne die Datei *waterCurse.py* im Ordner *whileLoops*, und speichere sie dann als *flowerTrail.py*.

Um hinter dem Spieler einen endlosen Blumenkorso zu erstellen, nimm die folgenden Änderungen am Programm vor:

1. Setze die Bedingung der `while`-Schleife auf `True`.
2. Lösche die Variable `count` und die Funktionalität zum Hochzählen.
3. Ändere das `Blocktyp`-Argument der Funktion `setBlock()` von 8 in 38.
4. Verringere den Wert des Arguments für die Funktion `sleep()` auf 0.2, um in jeder Sekunde fünf neue Blumen zu erstellen.
5. Speichere das Programm, und führe es aus. Bild 7.3 zeigt, wie das Ergebnis aussehen sollte.

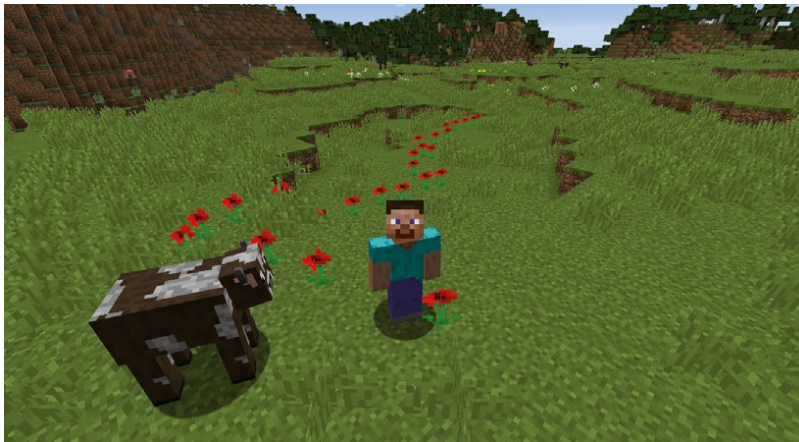


Bild 7.3: All die schönen Blumen!

Bonus: Eine Spur der Zerstörung

Das Programm *flowerTrail.py* ist ausgesprochen flexibel. Ändere doch einmal den Blocktyp, der vom Programm platziert wird. Spaßig wäre es beispielsweise, einen explosiven TNT-Block (`setBlock(x, y, z, 46, 1)`) aufzustellen. Beachte das zusätzliche Argument 1 nach der 46 der ID für den TNT-Block). Die 1 sorgt dafür, dass das TNT bereits explodiert, wenn man nur draufschlägt; ein Feuerzeug ist hierfür nicht erforderlich. Zeige einfach auf einen TNT-Block, und klicke ein paarmal mit der linken Maustaste darauf, um ihn zur Explosion zu bringen!

Kunstvolle Bedingungen

Da `while`-Schleifen einen booleschen Wert für ihre Bedingung erwarten, kannst du alle Vergleichsoperatoren und booleschen Operatoren verwenden, die du bereits kennengelernt hast. Wir haben etwa schon gesehen, dass die Größer-als- und Kleiner-als-Operatoren genau so wie in den vorangegangenen Kapiteln funktionieren.

Du kannst `while`-Schleifen aber auch auf andere Weise mit Vergleichsoperatoren und booleschen Operatoren steuern. Sehen wir uns das an.

Zunächst einmal wollen wir eine etwas interaktivere Bedingung verfassen. Der folgende Code erstellt die Variable `continueAnswer` vor Beginn der Schleife und überprüft dann, ob der Wert "Y" ist. Du würdest vielleicht erwarten, dass wir naheliegenderweise das Wort `continue` (»fortsetzen«) als Variablennamen verwenden würden, aber das geht nicht, da dieses Wort in Python reserviert ist.

```
continueAnswer = "Y"
coins = 0
while continueAnswer == "Y":
    coins = coins + 1
    continueAnswer = input("Continue? Y/N")
print("You have " + str(coins) + " coins")
```

In der letzten Zeile der `while`-Schleife fordert das Programm beim Benutzer eine Eingabe an. Wenn der Benutzer daraufhin etwas anderes als "Y" eingibt, wird die Schleife beendet. Der Benutzer kann immer wieder **Y**

drücken, und jedes Mal wird der Wert der Variablen `coins` sich um 1 erhöhen.

Beachte, dass die abgefragte Variable `continueAnswer` bereits vor Beginn der Schleife erstellt wird. Wäre dies nicht der Fall, würde ein Fehler ausgegeben. Deswegen muss die Variable, mit der wir die Bedingung prüfen, bereits existieren, bevor wir sie verwenden. Außerdem muss sie `Y` sein, wenn das Programm die `while`-Schleife erstmals erreicht, denn sonst wäre die Bedingung nicht erfüllt, und der Anweisungskörper der `while`-Schleife würde gar nicht ausgeführt.

Mission 36: Ein Tauchwettbewerb

Wir wollen einmal ein paar nette Sachen mit `while`-Schleifen und dem Gleichheitsoperator (`==`) machen. In dieser Mission erstellst du ein Minispiel, in dem der Spieler möglichst lange unter Wasser tauchen muss. Das Programm zeigt an, wie lange der Spieler unter Wasser ist, und gibt am Ende das Ergebnis aus. Wenn es dem Spieler gelingt, länger als 6 Sekunden unter Wasser zu bleiben, überschüttet das Programm ihn mit Blumen, um ihm zu gratulieren.

Hier ist etwas Code als Ausgangspunkt:

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
import time

score = 0
pos = mc.player.getPos()
blockAbove = mc.getBlock(pos.x, pos.y + 2, pos.z) ❶

# Hier eine while-Schleife einfügen ❷
time.sleep(1)
pos = mc.player.getPos()
blockAbove = mc.getBlock(pos.x, pos.y + 2, pos.z) ❸
score = score + 1 ❹
mc.postToChat("Current score: " + str(score))

mc.postToChat("Final score: " + str(score))

if score > 6: ❺
    finalPos = mc.player.getTilePos()
    mc.setBlocks(finalPos.x - 5, finalPos.y + 10, finalPos.z - 5,
                 finalPos.x + 5, finalPos.y + 10, finalPos.z + 5, 38)
```

*divingContest
.py*

Speichere das Programm als *divingContest.py* im Ordner *whileLoops*. Die Variable `score` verzeichnet, wie lange unser Spieler schon unter Wasser ist.

Führe den Code aus, und sieh dir an, was passiert. Das Programm ist noch nicht vollständig, sondern es überprüft bislang nur, ob sich der Spieler einmal unter Wasser befindet. Dann endet es.

Bevor wir das korrigieren, wollen wir uns ansehen, was der Rest des Codes macht. Die Variable `blockAbove` speichert den Blocktyp am Kopf des Spielers ❶. Befindet sich dieser Kopf unter Wasser, dann speichert die Variable den Wert 8 (Wasserblock-ID). Im weiteren Verlauf des Codes speicherst du die ID des Blocks über dem Kopf des Spielers erneut in `blockAbove` ❸. Das bedeutet: Wenn du deine `while`-Schleife erstellst, wird `blockAbove` auf den aktuellen Block über dem Kopf des Spielers festgelegt. Bei ❹ addiert das Programm 1 zur Gesamtzahl der Sekunden hinzu, die der Spieler unter Wasser verbracht hat, und bei ❺ wird mit einer `if`-Anweisung ein Blumenregen über dem Spieler erstellt, wenn das Resultat größer als 6 ist.

Deine Aufgabe besteht darin, eine Schleife zum Programm hinzuzufügen, die die Variable `blockAbove` als Bedingung bei ❷ verwendet. Mit der `while`-Schleife überprüfst du, ob `blockAbove` den Blocktyp »Wasser« (8) oder »fließendes Wasser« (9) hat. Dies kannst du mit der folgenden Bedingung in der `while`-Schleife tun: `while blockAbove == 8 or blockAbove == 9`. Hiermit wird überprüft, ob der Spieler sich zurzeit unter Wasser befindet. Diese Prüfung wird bei jedem erneuten Durchlauf der Schleife wiederholt.

Such dir zum Testen des Programms eine Stelle, an der das Wasser mindestens drei Blöcke tief ist, und springe hinein. Das Programm läuft nur dann, wenn du bereits im Wasser bist. Wenn du das Programm ausführst, sollte es anzeigen, wie viele Sekunden du bereits unter Wasser bist. Schwimme dann irgendwann an die Oberfläche. Das Programm zeigt nun dein Resultat an und überschüttet dich mit Blumen, wenn du mindestens 6 Sekunden untergetaucht warst. Bild 7.4 zeigt den Spieler unter Wasser und das aktuelle Resultat; Bild 7.5 zeigt die Blumen, die über ihm ausgeschüttet werden, nachdem er den Tauchgang erfolgreich absolviert hat.



Bild 7.4: Atem anhalten – so lange bin ich schon unter Wasser.



Bild 7.5: Ich habe einen Blumenregen gewonnen. Entzückend!

Bonus: So sehen Sieger aus

Füge doch noch ein paar zusätzliche Preise hinzu, indem du mehr Code in die if-Anweisung am Ende des Programms schreibst. Erzielt der Spieler einen Highscore, dann könntest du ihm zur Belohnung einen Goldblock geben. Versuche doch einmal, verschiedene Schwierigkeitsgrade mit jeweils unterschiedlichen Belohnungen hinzuzufügen.

Boolesche Operatoren und while-Schleifen

Wenn deine while-Schleife mehrere Bedingungen kombinieren soll, kannst du die booleschen Operatoren *and*, *or* und *not* benutzen. Die folgende Schleife beispielsweise iteriert, solange der Benutzer das richtige Passwort noch nicht eingegeben und dies auch noch nicht dreimal versucht hat:

```
password = "cats"
passwordInput = input("Please enter the password: ")
attempts = 0

while password != passwordInput and attempts < 3: ❶
    attempts += 1 ❷
    passwordInput = input("Incorrect. Please enter the password: ") ❸

if password == passwordInput: ❹
    print("Password accepted.")
```

Die Bedingung ❶ der while-Schleife erledigt zwei Aufgaben: Sie überprüft, ob sich das Passwort von der Eingabe des Benutzers unterscheidet (`password != passwordInput`) und ob der Benutzer bislang weniger als dreimal versucht hat, das Passwort einzugeben (`attempts < 3`). Mit dem *and*-Operator kann die while-Schleife beide Bedingungen gleichzeitig testen. Ist die Bedingung *False*, dann zählt die Schleife die Variable `attempts` hoch ❷ und fordert den Benutzer erneut auf, das Passwort einzugeben ❸. Die Schleife wird beendet, wenn der Benutzer das richtige Passwort eingibt oder die Variable `attempts` größer als 3 wird. Nach dem Abschluss der Schleife gibt das Programm `Password accepted` nur dann aus, wenn der Benutzer das richtige Passwort eingegeben hat ❹.

Bereichsprüfung in while-Schleifen

Du kannst in while-Schleifen auch überprüfen, ob sich ein Wert in einem bestimmten Bereich befindet. Mit dem folgenden Code etwa stellst du fest, ob der vom Benutzer eingegebene Wert zwischen 0 und 10 liegt. Tut er dies nicht, wird die Schleife beendet.

```
position = 0
while 0 <= position <= 10: ❶
    position = int(input("Enter your position 0-10: "))
    print(position)
```

Wenn die Variable `position` größer als 10 wird, wird die Schleife nicht wiederholt ❶. Dasselbe passiert, wenn der Wert kleiner als 0 ist. In Minecraft ist so etwas nützlich, wenn du feststellen möchtest, ob die aktuelle Spielerposition sich innerhalb eines bestimmten Bereichs in der Spielwelt befindet. Wir werden das in der nächsten Mission sehen.

Mission 37: Der König des Tanzbodens

Zeit für ein Tänzchen! Bevor du allerdings deine Fähigkeiten präsentieren kannst, ist erst einmal der Bau einer Tanzfläche angesagt. Mit dem Programm in dieser Mission baust du eine Tanzfläche, die zweimal pro Sekunde in verschiedenen Farben blinkt, solange der Spieler sich darauf befindet.

Unten findest du wie üblich das Codegerüst. Damit wird eine Tanzfläche an der aktuellen Position des Spielers gebaut. Mit einer `if`-Anweisung werden die Farben geändert. Allerdings ist das Programm noch nicht vollständig.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
import time

pos = mc.player.getTilePos()
floorX = pos.x - 2
floorY = pos.y - 1
floorZ = pos.z - 2
width = 5
length = 5
block = 41
mc.setBlocks(floorX, floorY, floorZ, ❶
             floorX + width, floorY, floorZ + length, block)

while floorX <= pos.x <= floorX + width and # Überprüfung einfügen, ↵
ob z auf der Tanzfläche ist ❷
    if block == 41: ❸
        block = 57
    else:
        block = 41
    mc.setBlocks(floorX, floorY, floorZ,
                 floorX + width, floorY, floorZ + length, block)
    pos = mc.player.getTilePos()
    time.sleep(0.5)
```

danceFloor.py

Öffne IDLE, erstelle eine neue Datei, und speichere das Programm als *danceFloor.py* im Ordner *whileLoops*. Der Code baut die Tanzfläche an der

aktuellen Position des Spielers ❶ und speichert ihre Position und Größe dann in den Variablen `floorX`, `floorY`, `floorZ`, `width` und `length`. Innerhalb der `while`-Schleife verwendet der Code eine `if`-Anweisung, um die Blöcke zu wechseln, aus denen die Tanzfläche besteht ❸. Hierdurch entsteht der Eindruck der blinkenden Tanzfläche.

Damit das Programm richtig funktioniert, musst du die Bedingung der `while`-Schleife so abändern, dass überprüft wird, ob die z-Koordinate des Spielers sich auf der Tanzfläche befindet ❷. Mit anderen Worten: Überprüfe, ob `pos.z` größer als oder gleich `floorZ` und kleiner als oder gleich `floorZ` plus `length` ist. Wie das prinzipiell funktioniert, habe ich bei der Überprüfung, ob `pos.x` sich auf der Tanzfläche befindet, vorgemacht (`floorX <= pos.x <= floorX + width`). Bild 7.6 zeigt das Programm in Aktion.

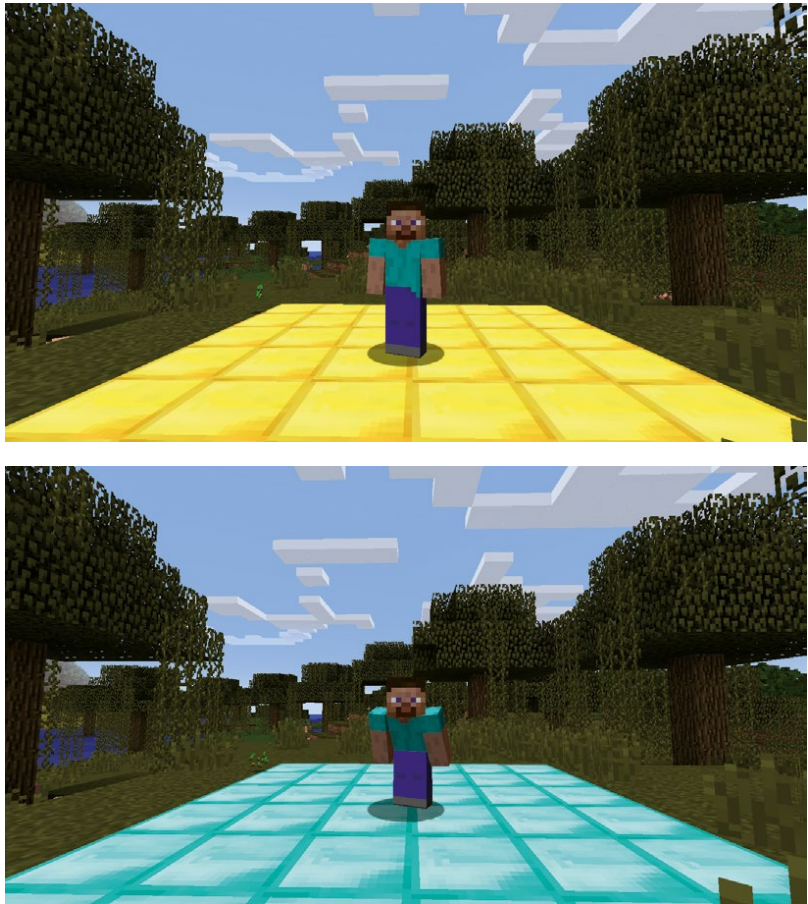


Bild 7.6: Jetzt gehen wir in die Disko!

Wenn du das Programm ergänzt hast, dann speichere es, und führe es aus. Unter dem Spieler sollte nun eine Tanzfläche erscheinen, die ihr Aussehen jede halbe Sekunde ändert. Tanze ein bisschen, und amüsiere dich. Wenn du keine Lust mehr hast, gehe von der Tanzfläche. Diese sollte dann aufhören zu blinken. Beachte, dass sie sich anschließend nicht wieder einschaltet: Du musst das Programm erneut starten, um auf diese Weise eine neue Tanzfläche zu erstellen.

Bonus: Die Party ist vorbei

Wenn der Spieler mit dem Tanzen fertig ist, sollte die Tanzfläche wieder verschwinden. Hierzu musst du die Tanzfläche in Luft verwandeln, wenn die Schleife beendet wird.

Verschachtelte if-Anweisungen und while-Schleifen

Mit if-Anweisungen und verschachtelten if-Anweisungen innerhalb von while-Schleifen hast du programmiertechnisch fast unendliche Möglichkeiten. Vielleicht hast du im Code von Mission 37 (Seite 177) die verschachtelte if-Anweisung bemerkt.

Im folgenden Beispiel überprüft die verschachtelte if-Anweisung, welches Wort zuletzt ausgegeben wurde, und entscheidet daraufhin, ob es seinerseits die Wörter "mine" und "craft" ausgibt.

```
word = "mine"
count = 0
while True:
    print(word)
    if word == "mine":
        word = "craft"
    else:
        word = "mine"
```

Die Variable word speichert das erste Wort, das ausgegeben wird. Mit der if-Anweisung in der Schleife wird dann festgestellt, ob das aktuelle Wort "mine" heißt. Ist dies der Fall, dann wird es in "craft" geändert und bei der nächsten Iteration der Schleife ausgegeben. Ist das Wort nicht "mine", dann wird es in "mine" geändert. Hierbei handelt es sich um eine Endlosschleife, d. h., du musst den Programmablauf mit **Strg+C** abbrechen.

Du kannst auch `elif`-Anweisungen und weitere `while`-Schleifen in `while`-Schleifen verschachteln.

Das folgende Programm fragt den Benutzer, ob er alle Zahlen zwischen 1 und 1 Mio. ausgeben möchte:

```

userAnswer = input("Print the numbers between 1 and 1000000? (yes/no): ")

if userAnswer == "yes": ❶
    count = 1
    while count <= 1000000: ❷
        print(count)
        count += 1

```

Die `if`-Anweisung überprüft, ob die Eingabe des Benutzers `yes` lautet ❶. Tut sie dies, dann führt das Programm die Schleife aus, die in der `if`-Anweisung verschachtelt ist ❷. Bei jeder anderen Eingabe wird die Schleife hingegen nicht ausgeführt, sondern das Programm wird beendet.

Mission 38: Die Berührung des Midas

Midas ist ein König aus dem Sagenreich. Der Legende nach wurde alles, was er berührte, zu Gold. Deine Aufgabe besteht nun darin, ein Programm zu schreiben, das jeden Block unter dem Spieler in Gold verwandelt – außer natürlich Luft- und Wasserblöcke (sonst hättest du ein ernsthaftes Problem). Zur Erinnerung: Gold hat die Blocktyp-ID 41, stilles Wasser die ID 9 und Luft die ID 0.

midas.py

```

from mcpi.minecraft import Minecraft
mc = Minecraft.create()

air = 0
water = 9

# Hier eine Endlosschleife einfügen ❶
pos = mc.player.getTilePos()
blockBelow = mc.getBlock(pos.x, pos.y - 1, pos.z)

# Hier eine if-Anweisung hinzufügen ❷
mc.setBlock(pos.x, pos.y - 1, pos.z, 41)

```

Öffne IDLE, und erstelle eine neue Datei. Speichere diese dann als *midas.py* im Ordner *whileLoops*. Du musst das Programm noch etwas erweitern, damit es tut, was es tun soll. Zunächst fügst du eine `while`-Endlosschleife hinzu ❶. Vergiss nicht, dass eine `while`-Endlosschleife eine Bedingung

hat, die stets `True` ist. Außerdem musst du eine `if`-Anweisung hinzufügen, die überprüft, ob der Block unter dem Spieler nicht Luft oder stilles Wasser ist ②. Die ID des Blocks unter dem Spieler ist in der Variablen `blockBelow` gespeichert, die Werte für Luft und Wasser in den Variablen `air` bzw. `water`.

Hast du das Programm ergänzt, dann speichere es und führe es aus. Der Spieler sollte nun eine Spur aus Gold hinter sich herziehen. Wenn du jedoch ins Wasser springst oder durch die Luft fliegst, sollten sich die Blöcke unter dir nicht ändern. Bild 7.6 zeigt das Programm bei der Ausführung.



Bild 7.7: Was ich auch berühre, es wird zu Gold!

Zum Beenden der Endlosschleife rufst du **Shell ▶ Restart Shell** in deiner IDLE-Shell auf bzw. klickst in die Shell und drückst **Strg+C**.

Bonus: Ich pflüge tagein, tagaus

Das Programm `midas.py` kannst du an die unterschiedlichsten Anforderungen anpassen. Wie müsstest du es abändern, damit Erdblöcke automatisch in urbares Ackerland verwandelt werden? Oder in Grasland?

Eine while-Schleife mit break beenden

Bei while-Schleifen hast du die volle Kontrolle darüber, wann und wie deine Schleife endet. Bislang haben wir Schleifen nur durch Bedingungen beendet, aber du kannst auch eine spezielle Anweisung dafür verwenden: die break-Anweisung. Mit dieser Anweisung wird eine while-Schleife sofort beendet. Sehen wir uns das Konzept genauer an.

Eine Möglichkeit, break-Anweisungen zu verwenden, besteht darin, sie in einer if-Anweisung abzulegen, die in der Schleife verschachtelt ist. Hierdurch wird die Schleife sofort beendet, sobald die Bedingung der if-Anweisung True wird. Der folgende Code fordert vom Benutzer so lange eine Eingabe an, bis er "exit" eintippt:

```
while True: ❶
    userInput = input("Enter a command: ") ❷
    if userInput == "exit": ❸
        break ❹
    print(userInput)
print("Loop exited") ❺
```

Dies ist eine Endlosschleife, denn hier wird while True benutzt ❶. Bei jeder Wiederholung der Schleife wird der Benutzer um Eingabe eines Befehls gebeten ❷. Mit einer if-Anweisung überprüft das Programm, ob die Eingabe "exit" lautet ❸. Wenn die Eingabe mit der Bedingung übereinstimmt, beendet die break-Anweisung die Schleife ❹, und das Programm wird nach dem Schleifenkörper fortgesetzt (d. h., in der Python-Shell wird "Loop exited" ausgegeben ❺).

Mission 39: Permanenten Chat mit Schleife erstellen

In Mission 13 (Seite 92) hast du ein Programm erstellt, das mithilfe von Strings, Eingabe und Ausgabe Meldungen des Benutzers im Chat postet. Das war zwar schon recht nützlich, aber am Ende doch etwas beschränkt, denn du musstest das Programm jedes Mal, wenn du eine neue Meldung posten wolltest, neu starten.

In dieser Mission werden wir dein Chat-Programm mit einer while-Schleife verbessern, damit Benutzer beliebig viele Meldungen posten können, ohne das Programm neu starten zu müssen.

Öffne die Datei *userChat.py* im Ordner *strings*, und speichere sie als *chatLoop.py* im Ordner *whileLoops*.

Damit das Programm jede neue Meldung postet, ohne zuvor neu gestartet zu werden, musst du den Code wie folgt erweitern:

1. Füge eine *while*-Endlosschleife zum Programm hinzu.
2. Füge in der Schleife eine *if*-Anweisung hinzu, mit der überprüft wird, ob der Benutzer "exit" eingegeben hat. Falls dies der Fall ist, wird die Schleife unterbrochen.
3. Definiere die Variable *userName* in jedem Fall vor dem Beginn der Schleife.

Hast du alle Änderungen vorgenommen, dann speicherst du das Programm und führst es aus. In der Python-Shell wirst du aufgefordert, deinen Benutzernamen einzugeben. Tue dies, und drücke dann **Enter**. Als Nächstes fordert dich das Programm zur Eingabe einer Meldung auf. Gib also eine Meldung ein, und drücke erneut **Enter**. Das Programm wird seine Frage so lange wiederholen, bis du *exit* eingibst. Bild 7.8 zeigt meinen Chat in Aktion.

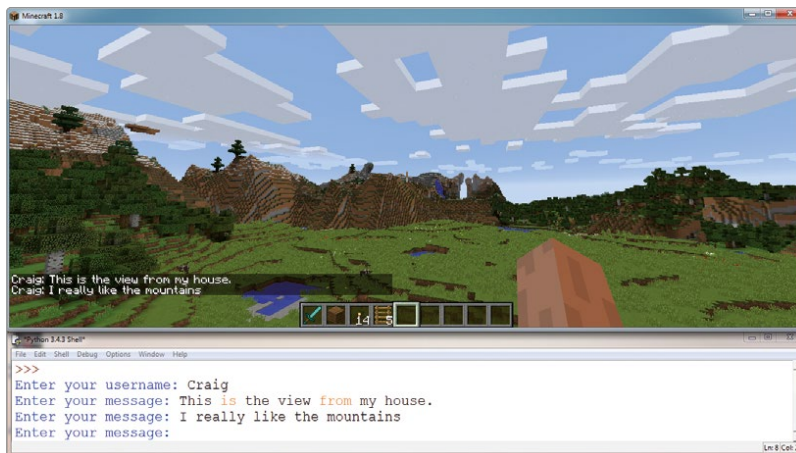


Bild 7.8: Ich führe Selbstgespräche.

Bonus: Block-Chat

Erweitere die Chat-Funktion so, dass Benutzer Blöcke erstellen können. Wenn der Benutzer beispielsweise "wool" eingibt, soll das Programm einen Wollblock erstellen. Zu diesem Zweck kannst du mit `elif`-Anweisungen, die deine `if`-Anweisung erweitern, die Benutzereingabe überprüfen.

while-else-Anweisungen

Wie `if`-Anweisungen können auch `while`-Schleifen sekundäre Bedingungen aufweisen, die durch `else`-Anweisungen ausgelöst werden.

Die `else`-Anweisung wird ausgeführt, wenn die Bedingung einer `while`-Anweisung `False` ist. Anders als der `while`-Anweisungskörper wird die `else`-Anweisung nur ein einziges Mal ausgeführt, wie du hier siehst:

```
message = input("Please enter a message.")

while message != "exit":
    print(message)
    message = input("Please enter a message.")
else:
    print("User has left the chat.")
```

Diese Schleife wiederholt sich, solange die eingegebene Nachricht nicht "exit" ist. Sobald jedoch die Nachricht "exit" eingegeben wird, wird die Wiederholung der Schleife abgebrochen, und über den `else`-Anweisungskörper wird "User has left the chat" ausgegeben.

Wenn du in die `while`-Schleife eine `break`-Anweisung einbaust, wird der `else`-Teil nicht ausgeführt. Der folgende Code ähnelt dem vorangegangenen Beispiel, enthält aber eine verschachtelte `if`-Anweisung und eine `break`-Anweisung. Gibt der Benutzer `abort` anstelle von `exit` ein, dann wird die Chat-Schleife beendet, ohne die Meldung "User has left the chat" an den Chat zu posten.

```
message = input("Please enter a message.")

while message != "exit":
    print(message)
    message = input("Please enter a message.")
```

```
if message == "abort":
    break
else:
    print("User has left the chat.")
```

Die `if`-Anweisung überprüft, ob die eingegebene Nachricht "abort" lautet. Trifft dies zu, dann wird die `break`-Anweisung ausgeführt, und die Schleife wird unterbrochen. Da die `break`-Anweisung verwendet wurde, wird der `else`-Anweisungskörper nicht ausgeführt und "User has left the chat" nicht ausgegeben.

Mission 40: Heiß oder kalt?

In dieser Mission erstellen wir das Heiß-kalt-Spiel in Minecraft. Du kennst es vielleicht: Ein Freund von dir versteckt einen Gegenstand, den du dann suchen musst. Bei der Suche erhältst du von deinem Freund Hinweise zur Entfernung vom versteckten Gegenstand. Wenn du dich in Richtung des Gegenstandes bewegst, sagt dein Freund: »Wärmer.« Bewegst du dich vom Gegenstand weg, dann sagt er: »Kälter.« Befindest du dich in unmittelbarer Nähe des gesuchten Objekts, dann ruft dein Freund: »Heiß!« Bist du dagegen sehr weit vom verborgenen Gegenstand entfernt, dann sagt er: »Eiskalt!«

Das Ziel dieses Minispiels in Minecraft ist es, einen Diamantblock zu finden, der zufällig an irgendeiner Stelle in der Spielwelt platziert wurde, und sich dann darauf zu stellen. Diese Version des Spiels kannst du allein spielen; wie weit du von dem verborgenen Block entfernt bist, sagt dir Python. Das Spiel endet, wenn du auf dem Diamantblock stehst.

Listing 7.2 platziert einen Block an einer zufälligen Position.

```
from mcpi.minecraft import Minecraft
import math
import time
import random
mc = Minecraft.create()

destX = random.randint(-127, 127)
destZ = random.randint(-127, 127)
destY = mc.getHeight(destX, destZ) ❶

block = 57
mc.setBlock(destX, destY, destZ, block) ❷
mc.postToChat("Block set")

while True:
```

blockHunter.py

```

pos = mc.player.getPos()
distance = math.sqrt((pos.x - destX) ** 2 + (pos.z - destZ)
** 2) ❸

if distance > 100: ❹
    mc.postToChat("Freezing")
elif distance > 50:
    mc.postToChat("Cold")
elif distance > 25:
    mc.postToChat("Warm")
elif distance > 12:
    mc.postToChat("Boiling")
elif distance > 6:
    mc.postToChat("On fire!")
elif distance == 0:
    mc.postToChat("Found it") ❺

```

Listing 7.2: Anfang des Heiß-kalt-Programms

Bevor das Programm den Block irgendwo aufstellt, sorgt es dafür, dass dies nicht unter der Erde geschieht. Hierzu verwendet es die Funktion `getHeight()` ❶, die für jede Position im Spiel den Block mit der höchsten y-Koordinate (d. h. den Block an der Oberfläche) ermittelt. Danach wird ein Diamantblock an einer zufälligen Position abgelegt ❷.

Der Code bei ❸ berechnet den Abstand zum Diamantblock. Hierzu verwendet er die Funktion `sqrt()`, die Bestandteil des Moduls `math` ist. (Dies ist auch der Grund dafür, warum am Anfang des Programms `import math` steht.) Die Funktion `sqrt()` berechnet die Quadratwurzel einer Zahl.

Hinweis

Listing 7.2 verwendet den *Satz des Pythagoras*. Mithilfe dieser Formel lässt sich, wenn die Längen zweier Seiten eines Dreiecks bekannt sind, die Länge der dritten ermitteln. In diesem Fall verwende ich den Abstand zwischen der Spielerposition und dem verborgenen Block auf der x- und der z-Achse zur Berechnung der kürzesten (also geraden) Entfernung vom verborgenen Block.

Welche Meldung das Programm anzeigt, hängt davon ab, wie weit du von dem Block entfernt bist. Dies finden wir mit einer `if`-Anweisung und der Variablen `distance` heraus ❹. Das Programm zeigt "Freezing" an, wenn du sehr weit entfernt bist; bist du hingegen ganz in der Nähe des Objekts der Begierde, dann zeigt es "On fire!" an.

Kopiere Listing 7.2 in eine neue IDLE-Datei, und speichere sie als *blockHunter.py* im Ordner *whileLoops*.

Im Moment funktioniert das Programm zwar, es endet jedoch nicht, sobald du den Block gefunden hast. Zur Vervollständigung des Codes musst du eine *break*-Anweisung hinzufügen, die greift, sobald die Entfernung des Spielers zum gesuchten Block 0 ist ⑤.

Hast du das Programm ergänzt, dann speichere es, und führe es aus. Ein zufälliger Block wird generiert, und es gilt, diesen zu finden. Das Programm sollte enden, sobald du den Block gefunden hast und auf ihm stehst. Bild 7.9 zeigt, wie ich gerade einen Block gefunden habe.

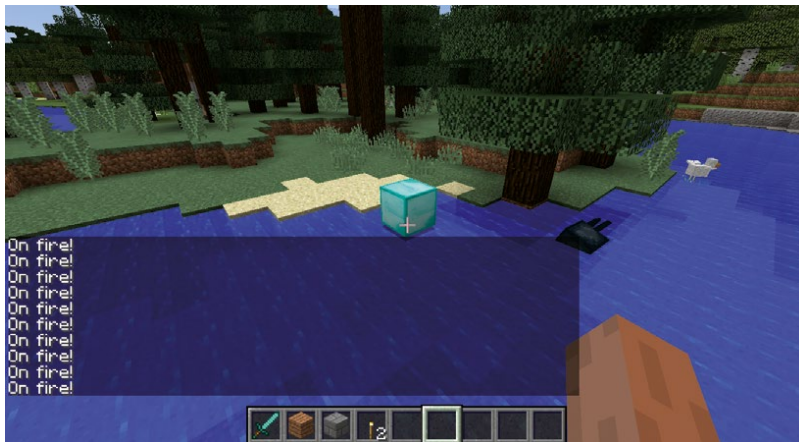


Bild 7.9: Block gefunden! Jetzt muss ich mich nur noch draufstellen.

Bonus: Die Uhr tickt

Das Programm *blockHunter.py* gibt dir unbegrenzt viel Zeit, um nach dem Block zu suchen. Versuche doch mal, eine Möglichkeit zu finden, mit der sich anzeigen lässt, wie lange der Spieler braucht, um den Block zu finden. Du könntest sogar die Zeit beschränken, die dem Spieler für die Suche zur Verfügung steht. Kriegst du das hin?

Was du gelernt hast

Gut gemacht! Du hast jetzt eine Menge über while-Schleifen gelernt. Du kannst while-Schleifen und while-Endlosschleifen erstellen und Schleifen mit Bedingungen und booleschen Operatoren verwenden. Mit Schleifen kannst du nun Programme schreiben, die Code wiederholt ausführen. So sparst du eine Menge Zeit und kannst dich darauf konzentrieren, Minecraft zu meistern. In Kapitel 8 lernst du eine weitere Möglichkeit kennen, um Code wiederzuverwenden – in Form von Funktionen.

Kapitel 8

Meine geheime Superkraft: Funktionen

Funktionen sind wiederverwendbare Codeabschnitte, die bestimmte Aufgaben erledigen. Angenommen, du möchtest einen Code schreiben, der in Minecraft einen Baum erstellt. Natürlich könntest du einen solchen Code jedes Mal, wenn du ihn in deinem Programm brauchst, neu schreiben (bzw. per Zwischenablage kopieren), aber das wäre ausgesprochen ineffizient – vor allem, wenn du daran eine Änderung vornehmen wolltest.

Insofern bietet es sich eher an, den Code zum Erstellen von Bäumen als *Funktion* zu schreiben. In den vorangegangenen Kapiteln haben wir bereits Funktionen verwendet, etwa `str()`, `input()` oder `int()`. All das sind Funktionen, die in Python bereits vorhanden sind. Außerdem kennst du auch schon ein paar Minecraft-Funktionen wie etwa `getBlocks()` oder `setPos()`, die Bestandteil der Minecraft-Python-API sind. In diesem Kapitel lernst du nun, *eigene* Funktionen zu verfassen.

Du erstellst und verwendest Funktionen aus den folgenden Gründen:

- **Wiederverwendbarkeit.** Funktionen sparen Zeit. Da du denselben Code nicht immer und immer wieder schreiben musst, wird das Entwickeln von Programmen deutlich beschleunigt und vereinfacht.
- **Debugging.** Durch das Zusammenfassen von Aufgaben in Codegruppen ist es einfacher, Problemursachen zu erkennen und notwendige Korrekturen vorzunehmen.

- **Modularität.** Du kannst verschiedene Funktionen entwerfen, die du dann unabhängig voneinander im selben Programm verwenden kannst. So kannst du Code einfacher mit anderen Leuten teilen und Funktionen auch in anderen Programmen wiederverwenden.
- **Skalierbarkeit.** Dank Funktionen wird es leichter, ein Programm zu erweitern und auch mehr Daten damit zu verarbeiten.

Eigene Funktionen definieren

Wir wollen uns einmal ansehen, wie du Funktionen in deinem Code verwenden kannst. Im folgenden Beispiel erstelle ich eine Funktion namens `greet ing()`, die nichts anderes tut, als zwei Zeilen auszugeben:

```
def greet ing():  
    print("Hello")  
    print("Nice to meet you")
```

Das Schlüsselwort `def` (eine Abkürzung für *Definieren*) zeigt Python an, dass du eine Funktion schreibst. Jedes Mal, wenn du eine Funktion entwerfen möchtest, musst du zuallererst `def`, gefolgt vom Funktionsnamen, schreiben. In unserem Beispiel heißt die Funktion `greet ing`. Vergiss nicht, die Klammern und den Doppelpunkt am Ende der ersten Zeile zu setzen. Die Zeilen, die auf den Doppelpunkt folgen, bilden den *Funktionskörper*, d. h. den Code, der beim Aufruf der Funktion ausgeführt wird.

Hinweis

Achte in deinem Code auf eine einheitliche Einrückung. Stelle dabei sicher, dass der Funktionskörper immer mit vier Leerzeichen eingezogen wird.

Eine Funktion kann beliebig viele Anweisungen enthalten. Dazu gehören auch `if`-Anweisungen, ferner Schleifen, Variablen, Bedingungen, Rechenoperatoren usw. Wenn du das Ende des Funktionscodes erreicht hast, musst du aufhören, die Zeilen einzuziehen, damit Python weiß, welche Anweisungen zur Funktion und welche zu anderen Teilen des Codes gehören.

Innerhalb eines Programms kannst du beliebig viele Funktionen erstellen. Sie müssen lediglich unterschiedliche Namen haben.

Funktionen aufrufen

Um eine Funktion zu verwenden oder *aufzurufen*, schreibst du den Funktionsnamen und übergibst dann alle erforderlichen Argumente in Klammern. Nimmt deine Funktion keine Argumente entgegen, dann notierst du einfach den Funktionsnamen und setzt eine öffnende und eine schließende Klammer direkt dahinter.

Um die oben definierte Funktion `greeting()` aufzurufen, würdest du beispielsweise folgenden Code verwenden:

```
greeting()
```

Eine Funktion kann beliebig oft aufgerufen werden. Rufen wir die Funktion `greeting()` also spaßeshalber dreimal auf:

```
greeting()  
greeting()  
greeting()
```

Wenn du das Programm ausführst, sollte das Ergebnis der Funktion dreimal angezeigt werden – also so:

```
Hello  
Nice to meet you  
Hello  
Nice to meet you  
Hello  
Nice to meet you
```

Du musst die Funktion innerhalb des Programms aufrufen, sonst tut sie gar nichts. Dies ist ein häufig gemachter Fehler. Wenn du ein Programm ausführst, das einige Funktionen definiert, und dein Code trotzdem nichts tut, dann könnte das daran liegen, dass du vergessen hast, die erstellten Funktionen aufzurufen.

Du kannst Funktionen auch aus anderen Funktionen heraus aufrufen. Dies gilt gleichermaßen für integrierte Python-Funktionen wie auch für solche, die du selbst erstellt hast. Wie das in Aktion aussieht, werden wir gleich sehen.

Funktionen arbeiten mit Argumenten

Die Klammern in einem Funktionsaufruf enthalten die Argumente, also Werte, die von der Funktion verwendet werden. Diese Werte werden innerhalb der Funktion bei der Ausführung in bestimmte Variablen eingesetzt. Allerdings benötigt nicht jede Funktion Argumente – unsere Funktion `greeting()` etwa nicht.

Aber nehmen wir einmal an, ich möchte jemanden mit seinem Namen begrüßen. Wenn ich dies als Funktion schreibe, kann ich den Code wiederverwenden, um verschiedene Personen namentlich zu begrüßen:

```
def fancyGreeting(personName):  
    print("Hello, " + personName)  
  
fancyGreeting("Mario")  
fancyGreeting("Steve")
```

In diesem Beispiel wird die Funktion zweimal mit unterschiedlichen Argumenten aufgerufen (nämlich "Mario" und "Steve"). Führt du das Programm aus, dann sieht das wie folgt aus:

```
Hello, Mario  
Hello, Steve
```

Vergisst du beim Aufruf einer Funktion, die ein Argument benötigt, ein solches zu übergeben, dann erhältst du einen Fehler. Ebenfalls wird ein Fehler ausgegeben, wenn eine Funktion mehrere Argumente benötigt und du eines davon vergisst. Versuchen wir einmal, die Funktion `fancyGreeting()` ohne Argumente aufzurufen:

```
fancyGreeting()
```

Daraufhin wird folgende Fehlermeldung angezeigt:

```
Traceback (most recent call last):  
  File "<pyshell#2>", line 1, in <module>  
    fancyGreeting()  
TypeError: fancyGreeting() takes exactly 1 argument (0 given) ❶
```

Die Meldung ist nützlich, denn die letzte Zeile erläutert, welches Problem vorliegt ❶. Die Funktion `fancyGreeting()` nimmt per Definition genau ein Argument entgegen, aber ich habe keines übergeben – daher die Fehlermeldung.

Du kannst Funktionen so definieren, dass sie mehrere Argumente entgegennehmen. Das folgende Programm etwa enthält eine Funktion, die jemanden mit »Hello« anspricht und dann nach kurzer Zeit »Goodbye« sagt. Diese Funktion nimmt als Argumente den Namen der Person und die Anzahl der Sekunden für die Wartedauer entgegen:

```
import time

def helloAndGoodbye(personName, secsToWait): ❶
    print("Hello, " + personName)
    time.sleep(secsToWait)
    print("Goodbye, " + personName)

helloAndGoodbye("Mario", 10) ❷
helloAndGoodbye("Steve", 23)
```

Die Argumente werden bei der Definition einer Funktion durch Kommata getrennt ❶. Beim Aufruf der Funktion müssen die Argumente dann in derselben Reihenfolge übergeben werden, in der sie definiert wurden ❷.

Hinweis

Die Begriffe *Argument* und *Parameter* werden praktisch vollständig synonym verwendet. Die Parameter einer Funktion definieren die Art der akzeptierten oder erforderlichen Argumente, und Argumente selbst sind die Werte, die du beim Aufruf an die Funktion übergibst. Der Übersicht halber verwenden wir in diesem Buch ausschließlich den Begriff *Argument*.

Mission 41: Einen ganzen Wald pflanzen

Deine Aufgabe besteht darin, in Minecraft einen Wald mit ganz vielen Bäumen zu erstellen. Da ein Wald nichts anderes ist als ein großer Haufen Bäume, werden wir ihn mit einer Funktion erstellen, die genau einen Baum anlegt, und diese Funktion dann vielfach aufrufen, um am Ende einen Wald zu erhalten.

Listing 8.1 stellt die Ausgangsbasis für den Code dar, den du entwickeln wirst.

forest.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

❶ def growTree(x, y, z):
    # Einen Baum an den angegebenen Koordinaten erstellen
    # Hier Code zum Erstellen eines Baums einfügen

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z

❷ growTree(x + 1, y, z)
```

Listing 8.1: Die Struktur eines Programms, das Funktionen verwendet, um einen Wald aus lauter Bäumen zu erstellen

Die Funktion `growTree()` ❶, die ich in diesem Code geschrieben habe, nimmt Argumente für die Koordinaten entgegen, an denen der Baum erstellt werden soll. Du musst nun den Code im Funktionskörper schreiben, der einen Baum an den angegebenen Koordinaten anlegt. Zu diesem Zweck verwendest du die Funktionen `setBlock()` und `setBlocks()`.

Kopiere Listing 8.1 in eine neue IDLE-Datei und speichere sie als *forest.py* in einem neuen Ordner namens *functions*.

Wenn du erfolgreich etwas erstellt hast, was entfernt an einen Baum erinnert, und dieses Objekt dann auf dem Bildschirm erscheint, schreibe weitere Funktionsaufrufe mit anderen Argumenten, damit die Bäume an unterschiedlichen Positionen entstehen. Den ersten Baum habe ich für dich bereits übernommen ❷. Versuche einmal, bei jedem Programmaufruf mindestens neun Bäume vor dem Spieler zu erstellen. Bild 8.1 zeigt die Bäume, die mein Programm angelegt hat.



Bild 8.1: Eine Reihe prächtiger Bäume

Bonus: Der zufällige Wald

Mit der Funktion `randint()` im Modul `random` kannst du die Abstände zwischen den Bäumen in deinem Wald variieren.

Refactoring

Es kommt durchaus häufig vor, dass derselbe Codeblock in einem Programm mehrfach verwendet wird. Würde man denselben Code an mehreren Stellen im Programm benutzen und müsste man dann Änderungen daran vornehmen, dann wäre dies eine zeitaufwendige und ermüdende Angelegenheit. Vielleicht hast du das früher einmal in Programmen gemacht, aber es gibt eine viel bessere Methode dafür.

Du kannst deine Programme für die Verwendung von Funktionen umstrukturieren. Hierzu verschiebst du Code, der mehrfach ausgeführt werden soll, in eine einzelne Funktion. Diese kannst du dann beliebig oft aufrufen. Und weil du Änderungen nur einmal an zentraler Stelle vornehmen musst, sparst du Platz und vereinfachst auch die Wartung deines Programms. Der Vorgang der Umstrukturierung deines Codes in dieser Weise heißt *Refactoring*.

Nehmen wir an, der folgende Code bittet drei Personen um die Eingabe ihrer Namen und gibt dann eine Begrüßung aus:

```
name1 = input("Hello, what is your name?")
print("Pleased to meet you, " + name1)
name2 = input("Hello, what is your name?")
print("Pleased to meet you, " + name2)
name3 = input("Hello, what is your name?")
print("Pleased to meet you, " + name3)
```

Der Code wiederholt hier dieselben beiden Anweisungen dreimal. Und jetzt nehmen wir einmal an, du möchtest die Frage ändern. Bei nur drei Leuten ist das sicher kein Problem, aber wie sähe es bei hundert aus?

Die Alternative besteht darin, den Code als Funktion zu schreiben und dreimal aufzurufen. Und so sieht der Code nach dem Refactoring aus:

```
def helloFriend():
    name = input("Hello, what is your name?")
    print("Pleased to meet you, " + name)

helloFriend()
helloFriend()
helloFriend()
```

Wenn das Programm jetzt ausgeführt wird, fordert es eine Eingabe an und gibt dann einen String aus. Und all das tut es dreimal. Hier siehst du die Ein- und Ausgabe:

```
Hello, what is your name? Craig
Pleased to meet you, Craig
Hello, what is your name? Still Craig
Pleased to meet you, Still Craig
Hello, what is your name? Craig again
Pleased to meet you, Craig again
```

Diese zweite Version des Codes ist vom Resultat her identisch mit der ursprünglichen Variante, aber es springt sofort ins Auge, dass sie leichter zu lesen und wesentlich leichter zu ändern ist.

Mission 42: Einfach wegrationalisiert

Manchmal wirst du ein Programm schreiben, nur um danach festzustellen, dass du vielleicht doch Funktionen hättest verwenden sollen. (Mir passiert das ständig.) Das Refactoring von Code mit Funktionen ist eine extrem wichtige Technik, die du beherrschen solltest.

In dieser Mission wirst du das Refactoring eines Programms mit Funktionen üben, um das ständige Wiederholen derselben Anweisungen zu vermeiden.

Listing 8.2 platziert alle zehn Sekunden unter dem Spieler einen Melonenblock. Wir werden diesen Code mit Funktionen vereinfachen. In der aktuellen Version platziert das Programm drei Blöcke, indem es dreimal dieselbe Codezeile angibt. Bild 8.2 stellt das Ergebnis dar.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
```

```
import time
```

```
pos = mc.player.getPos()
x = pos.x
y = pos.y
z = pos.z
mc.setBlock(x, y - 1, z, 103)
time.sleep(10)
```

```
pos = mc.player.getPos()
x = pos.x
y = pos.y - 1
z = pos.z
mc.setBlock(x, y, z, 103)
time.sleep(10)
```

```
pos = mc.player.getPos()
x = pos.x
y = pos.y - 1
z = pos.z
mc.setBlock(x, y, z, 103)
time.sleep(10)
```

*melon
Function.py*

Listing 8.2: Das schreit nach Refactoring.

Das ist nicht hübsch, oder? Nein, wirklich nicht. Einige Zeilen werden wiederholt – ein untrügliches Zeichen dafür, dass der Code ein Refactoring unter Verwendung einer Funktionsdefinition vertragen könnte.

Tipp

Stelle zunächst fest, welche Codeabschnitte sich wiederholen, um einen Eindruck davon zu erhalten, was genau deine Funktion tun soll.



Bild 8.2: Drei köstliche Melonen auf dem Boden

Ändere den Code nun so ab, dass er insgesamt sechs Blöcke platziert. Dazu musst du deine Funktion sechsmal aufrufen. Erstelle eine neue Datei und speichere sie als *melonFunction.py* im Ordner *functions*. Kopiere Listing 8.2 in deine Datei und führe ein Refactoring mit einer Funktion durch. Diese neue Funktion nennst du `makeMelon()`.

Bonus: Blöcke zu deinen Füßen

Erweitere deine Funktion `makeMelon()` mit Argumenten, um den Blocktyp, die Wartezeit und den Abstand unter dem Spieler zu steuern.

Kommentieren mit Docstrings

Die Verwendung von Kommentaren in Python bietet dir die Möglichkeit zu erklären, was dein Code macht. Wenn Python ein Programm ausführt, dann wird alles ignoriert, was in einem Kommentar steht. Insofern haben Kommentare keine Auswirkungen auf die Ausführung des Codes. Der wesentliche Zweck von Kommentaren besteht darin, den Personen, die deinen Code studieren oder verwenden, zu erklären, was genau dort gerade passiert. Aber auch für dich selbst können Kommentare beim späteren Überprüfen deines Codes eine große Hilfe darstellen.

Da Funktionen für die Wiederverwendung gedacht sind, ist es durchaus sinnvoll, ihren Zweck zu erläutern. Um solche Funktionserklärungen zu schreiben, verwenden wir längere Kommentare, die als *Docstrings* bezeichnet werden. Ein Docstring ist eine mehrzeilige Anmerkung, die du an den Anfang deiner Funktion stellst, um ihre Verwendung zu erläutern.

Die Funktion `duplicateWord()` im folgenden Beispiel enthält einen Docstring, der den Sinn der Funktion beschreibt:

```
def duplicateString(stringToDb1):  
    """ Gibt einen String zweimal in derselben Zeile aus. ❶  
    Das Argument stringToDb1 muss ein String sein. """  
    print(stringToDb1 * 2)
```

Der Docstring sollte in der ersten Zeile einer Funktion stehen ❶. Am Anfang und am Ende des Docstrings stehen jeweils drei aufeinanderfolgende Anführungszeichen ("""). Bei Bedarf kann der Docstring so auch über mehrere Zeilen geschrieben werden.

Zeilenwechsel in Argumenten

Damit gerade lange Listen mit Argumenten einfacher zu lesen sind, erlaubt Python die Verteilung von Argumenten über mehrere Zeilen hinweg. Der Funktionsaufruf im folgenden Programm etwa führt seine Argumente in mehreren Zeilen auf, um die Lesbarkeit zu verbessern:

```
from mcpi.minecraft import Minecraft  
mc = Minecraft.create()  
  
pos = mc.player.getPos()  
width = 10  
height = 12  
length = 13  
block = 103  
mc.setBlocks(pos.x, pos.y, pos.z,  
             pos.x + width, pos.y + height, pos.z + length, block)
```

Zeilenwechsel in Argumenten sind besonders nützlich, wenn du Rechenoperatoren für Argumente benutzen, lange Variablennamen als Argumente verwenden oder viele Argumente für eine Funktion angeben möchtest.

Rückgabewerte von Funktionen

Es gibt zwei Arten von Funktionen: solche, die einen Wert zurückgeben, und solche, die das nicht tun. Bislang haben wir nur Funktionen ohne Rückgabewert erstellt. Sehen wir uns doch jetzt einmal den anderen Funktionstyp an.

Rückgabewerte von Funktionen sind etwas sehr Praktisches, denn so kann eine Funktion Daten bearbeiten und dann einen Wert an den Hauptteil des Programms zurückgeben. Stell dir beispielsweise vor, du verkaufst Kekse. Zur Berechnung des Preises, den du für jeden Keks nehmen musst, um genügend zu verdienen, addierst du zwei Goldmünzen zu der Summe hinzu, die du für die Herstellung des Kekses bezahlt hast, und multiplizierst die Summe mit 10. Wenn du eine Funktion entwickelst, die einen Wert zurückgibt, kannst du diese Berechnung schreiben und sie in Python wiederverwenden.

Beim Entwickeln eigener Funktionen kannst du mit dem Schlüsselwort `return` einen Wert aus der Funktion zurückgeben. Hier folgt der Code, mit dem du den Verkaufspreis für einen Keks ermittelst:

```
def calculateCookiePrice(cost):  
    price = cost + 2  
    price = price * 10  
    return price
```

Um einen Wert zurückzugeben, musst du einfach nur `return`, gefolgt vom gewünschten Wert, notieren – in diesem Fall `price`. Wenn du eine Funktion verwenden möchtest, von der ein Wert zurückgegeben wird, dann musst du sie an einer Stelle aufrufen, an der ein Rückgabewert erwartet wird. Um beispielsweise die Variable `priceOfCookie` festzulegen, rufst du die Funktion `calculateCookiePrice()` auf und übergibst einen Kostenwert, z. B. 6:

```
priceOfCookie = calculateCookiePrice(6) # Wert wird 80
```

Du kannst Funktionen mit Rückgabewert verwenden, um Variablenwerte festzulegen. Außerdem kannst du sie überall dort einsetzen, wo eine Werteingabe erwartet wird – sogar als Argument für eine andere Funktion.

Funktionen, die keinen Wert zurückgeben, können dagegen nicht zum Festlegen von Variablenwerten benutzt werden. Wir wollen uns den Unterschied einmal genauer ansehen.

Da die folgende Funktion einen Wert zurückgibt, kannst du sie dort verwenden, wo ein Wert erwartet wird: zum Festlegen einer Variablen oder sogar als Argument in einem anderen Funktionsaufruf:

```
def numberOfChickens():  
    return 5  
  
coop = numberOfChickens()  
print(numberOfChickens())
```

Führe den Code aus und sieh dir die Ausgabe an. Das Ergebnis der Funktion kannst du wie einen Wert behandeln und sogar Berechnungen damit anstellen. Hier addiere ich 4 zum Rückgabewert hinzu und speichere ihn in einer Variablen namens `extraChickens`:

```
extraChickens = 4 + numberOfChickens() # Wert 9
```

Die folgende Funktion hingegen enthält keine `return`-Anweisung, d. h., du kannst sie nicht anstelle eines Werts verwenden, sondern lediglich aufrufen:

```
def chickenNoise():  
    print("Cluck")  
  
chickenNoise()
```

Wenn du den Code in den Editor schreibst und ausführst, wird "Cluck" ausgegeben. Die Ausgabe kann jedoch nicht in anderen Anweisungen verwendet werden, weil es sich nicht um einen Rückgabewert handelt. Ich könnte natürlich auch versuchen, die Funktion mit einem String zu verketten:

```
multipleNoises = chickenNoise() + ", Bork"
```

Würde ich dieses Programm jedoch ausführen, würde ich folgende Fehlermeldung erhalten:

```
Traceback (most recent call last):  
  File "<pyshell#3>", line 1, in <module>  
    multipleNoises = chickenNoise() + ", Bork"  
TypeError: unsupported operand type(s) for +: 'function' and 'str'
```

Die Meldung besagt, dass du diese Funktion nicht mit einem String kombinieren kannst, weil sie – genau! – keinen Wert zurückgibt.

Allerdings könnte ich den Code natürlich auch abändern, sodass keine Ausgabe mehr erfolgt, sondern ein Wert zurückgegeben wird:

```
def chickenNoise():  
    return "Cluck"  
  
multipleNoises = chickenNoise() + ", Bork"  
print(multipleNoises)
```

Beim Ausführen würde dann die folgende Ausgabe erscheinen:

```
Cluck, Bork
```

Diesen Unterschied musst du dir merken: Wenn deine Funktion einen Wert zurückgeben soll, musst du eine `return`-Anweisung einbinden, andernfalls darfst du das nicht tun. Je mehr Erfahrungen du mit Funktionen sammelst, desto einfacher wird dir die Entscheidung fallen, ob deine Funktion einen Wert zurückgeben soll oder nicht.

Mission 43: Gedächtnisstütze für Block-IDs

Weil Minecraft so viele verschiedene Blöcke umfasst, fällt es oft schwer, alle Block-IDs im Kopf zu behalten. Die Werte für Melone (103) und Luft (0) habe ich immer drauf, aber die meisten anderen vergesse ich ständig. (Deswegen werde ich auch in Zukunft viel zu viele Häuser aus Melonen bauen.)

Damit das künftig einfacher wird, möchte ich, dass du mir ein Programm schreibst, das die Werte für die verschiedenen Blöcke zurückgibt. Dein Programm sollte zahlreiche Funktionen enthalten, die mir helfen, mir die Block-IDs zu merken. Der Name der einzelnen Funktion sollte jeweils mit dem Namen des Blocks identisch sein, dessen Wert sie zurückgibt. Listing 8.3 ist eine Funktion namens `melon()`, die den Wert für den Melonenblock (103) zurückgibt.

```
def melon():  
    """ Gibt den Wert für den Melonenblock zurück """  
    return 103
```

blockIds.py

Listing 8.3: Der Anfang des Programms, mit dem ich mir die Block-IDs merke

Erstelle eine neue Datei in IDLE und speichere sie als *blockIds.py* im Ordner *functions*. Kopiere Listing 8.3 in die Datei und füge Funktionen hinzu, die die Werte für die folgenden Blocktypen zurückgeben (siehe die Übersichten unter den auf Seite xv genannten URLs):

- Wasser
- Wolle
- Lava
- TNT
- Blume
- Diamantblock

Nachdem du deine Funktionen hinzugefügt hast, teste sie, indem du sie aufrufst, um Blöcke zu erstellen. Da neue Funktionen einen Blockwert zurückgeben, kannst du sie verwenden, um den Wert einer Variablen festzulegen, die du an die Funktion `setBlock()` übergibst. Der folgende Code stellt einen guten Einstieg dar:

```
from mcpi.minecraft import Minecraft  
mc = Minecraft.create()  
  
# Hier Funktionen einfügen  
  
block = melon()  
pos = mc.player.getTilePos()  
mc.setBlock(pos.x, pos.y, pos.z, block)
```

Bild 8.3 zeigt das Ergebnis des vollständigen Programms mit einem Test der Funktion `melon()`. Beachte, dass die Platzierung von Blöcken fest in das Programm einkodiert ist – der Block landet immer an deiner aktuellen Position.



Bild 8.3: Nie wieder Blocktypen merken – dank dieser praktischen Funktion

Tipp

Zum Platzieren eines Diamant-, TNT- oder sonstigen Blocks musst du zunächst die Funktion definieren, die den Wert des gewünschten Blocks zurückgibt. Danach rufst du die Funktion in deinem Code genau so auf, wie ich das oben mit der Funktion `melon()` getan habe.

Bonus: Noch mehr Blöcke

Füge weitere Funktionen über beliebige andere Blocktypen hinzu.

if-Anweisungen und while-Schleifen in Funktionen verwenden

In den Kapiteln 6 und 7 hast du gelernt, wie du `if`-Anweisungen in anderen `if`-Anweisungen und `while`-Schleifen in anderen `while`-Schleifen ablegst. Wir haben außerdem gesehen, dass sich `if`-Anweisungen in `while`-Schleifen ablegen lassen – und umgekehrt. In diesem Abschnitt erfährst du, wie `if`-Anweisungen und Schleifen in Funktionen eingesetzt

werden. Dies macht deine Funktionen extrem flexibel, denn du kannst nun Entscheidungen treffen und Code wiederholen.

if-Anweisungen

Wenn du eine if-Anweisung in einer Funktion schreibst, ist die Syntax identisch mit der einer regulären if-Anweisung. Du musst lediglich daran denken, die if-Anweisung um vier zusätzliche Leerzeichen am Anfang jeder Zeile einzuziehen, damit Python weiß, dass es sich um einen Teil der Funktion handelt.

Der folgende Code nimmt ein als String notiertes Zahlwort entgegen und gibt dessen numerischen Wert als Integer zurück. Für das Argument "four" beispielsweise wird 4 zurückgegeben:

```
def wordToNumber(numToConvert):
    """ Wandelt ein Zahlwort in einen Integer um """
    if numToConvert == "one":
        numAsInt = 1
    elif numToConvert == "two":
        numAsInt = 2
    elif numToConvert == "three":
        numAsInt = 3
    elif numToConvert == "four":
        numAsInt = 4
    elif numToConvert == "five":
        numAsInt = 5

    return numAsInt
```

Betrachten wir ein weiteres Beispiel. Die folgende Funktion überprüft, ob du eine Person bereits getroffen hast, und verwendet je nach Ergebnis die passende Begrüßung:

```
def chooseGreeting(metBefore): ❶
    """ Wählt eine Begrüßung abhängig davon aus, ob du denjenigen ↵
        schon einmal getroffen hast.
        Das Argument metBefore muss ein boolescher Wert sein """
    if metBefore:
        print("Nice to see you again") ❷
    else:
        print("Nice to meet you") ❸
```

chooseGreeting(True)
chooseGreeting(False)

Die Funktion `chooseGreeting()` nimmt ein boolesches Argument entgegen, das wir `metBefore` genannt haben ❶. Die `if`-Anweisung innerhalb der Funktion erzeugt dann abhängig vom Wert des Arguments eine passende Ausgabe. Ist der Wert `True`, dann lautet die Ausgabe "Nice to see you again" ❷, bei `False` ❸ heißt sie einfach "Nice to meet you".

Mission 44: Wollfärberei

Du hast die Methoden `setBlock()` und `setBlocks()` bereits mit Argumenten verwendet, um Blockkoordinaten und Blocktypen festzulegen. Allerdings haben diese Methoden noch ein zusätzliches optionales Argument, mit dem sich der *Blockzustand* festlegen lässt.

Jeder Block in Minecraft hat 16 Zustände mit den Werten 0 bis 15. Bei Wolle beispielsweise steht jeder Zustand für eine andere Farbe. TNT (Block-ID 46) explodiert erst, wenn du den Block zerschlägst, nachdem du den Zustand vom Standardwert 0 auf 1 umgestellt hast. Zwar haben alle Blöcke 16 Zustände, doch unterscheiden sich die Verhaltensweisen nicht in jedem einzelnen Zustand.

Um den Zustand eines Blocks festzulegen, gibst du bei den Funktionen `setBlock()` und `setBlocks()` jeweils ein zusätzliches Argument an. Der folgende Code erzeugt einen Block aus pinkfarbener Wolle:

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

block = 35
state = 6
# Einzelnen Block aus pinkfarbener Wolle erzeugen
mc.setBlock(10, 3, -4, block, state)

# Quader aus pinkfarbener Wolle erzeugen
mc.setBlocks(11, 3, -4, 20, 6, -8, block, state)
```

Wolle (Block-ID 35) erfüllt in Minecraft aufgrund ihrer verschiedenen Farben unterschiedliche Zwecke. Zwar ist es schwierig, sich die unterschiedlichen Blockzustände zu merken, aber das ist auch gar nicht mehr notwendig: Ein Programm erledigt das von nun an für uns.

Als Nächstes wollen wir deshalb ein Programm schreiben, das die Zustände von Wollblöcken enthält. Dieses Programm soll eine Funktion mit einem Argument enthalten, das die gewünschte Farbe als String erhält. Die Funktion gibt den Blockzustand, der die Farbe bestimmt, dann

als Integer zurück. Der größte Teil des Codes für dieses Programm wird in dieser Funktion enthalten sein, aber wir werden noch einige Codezeilen ergänzen, um die Benutzereingabe entgegenzunehmen und den Block zu platzieren. Mit unserer ausgefuchsten neuen Funktion werden wir dann die Farbe festlegen.

Zunächst müssen wir die Blockzustände ermitteln, die den verschiedenen Farben von Wolle entsprechen. Diese findest du unter dem auf Seite xv genannten URL im offiziellen Minecraft-Wiki, wenn du nach »Block Farbe Wolle« suchst. Hier siehst du etwas Code als Ausgangspunkt (pink-farben entspricht Blockzustand 6):

```
def getWoolState(color):  
    """ nimmt eine Farbe als String entgegen und gibt den  
    Wollblockzustand für diese Farbe zurück """  
    if color == "pink": ❶  
        blockState = 6  
    elif # elif-Anweisungen für die anderen Farben hinzufügen  
        # Hier blockState zurückgeben  
  
colorString = input("Enter a block color: ") ❷  
state = getWoolState(colorString)  
  
pos = mc.player.getTilePos() ❸  
mc.setBlock(pos.x, pos.y, pos.z, 35, state)
```

woolColors.py

Bislang enthält das Programm nur den Anfang der Funktion `getWoolState()`: eine `if`-Anweisung für die Farbe Pink ❶. Ebenfalls enthalten ist Code am Ende des Programms, der die Benutzereingabe für die Blockfarbe entgegennimmt ❷, sowie Code zum Platzieren des Wollblocks an der Position des Spielers ❸.

Deine Aufgabe besteht darin, die Funktionen `getWoolState()`-Funktion mit `elif`-Anweisungen für die anderen Farben und die entsprechenden Blockzustände zu ergänzen. Das Programm sollte als Argument ein Farbwort entgegennehmen und den Blockzustand als Integer zurückgeben. Wird beispielsweise das Argument "pink" eingegeben, dann muss der Rückgabewert 6 lauten. Außerdem musst du eine `return`-Anweisung zum Programm hinzufügen. Orientiere dich bei der Programmentwicklung an den Kommentaren.

Speichere die Datei abschließend als *woolColors.py* im Ordner *functions*.

Möchtest du die Benutzerfreundlichkeit des Programms erhöhen, dann kannst du eine Nachricht im Chat anzeigen, wenn das Argument keine

gültige Farbe ist. Bild 8.4 zeigt die Eingabe in der Python-Shell und die Platzierung des Wollblocks im Spiel.



Bild 8.4: Wollblöcke in beliebiger Farbe – schnell und einfach produziert

while-Schleifen

Wie `if`-Anweisungen lassen sich auch Schleifen innerhalb von Funktionen notieren. Dabei entspricht die Syntax einer Schleife in einer Funktion der einer regulären Schleife. Du musst lediglich daran denken, die Schleife um vier zusätzliche Leerzeichen am Anfang jeder Zeile einzuziehen, um anzugeben, dass sie zur Funktion gehört.

Im folgenden Beispiel gibt die `while`-Schleife innerhalb der Funktion das Argument `toPrint` aus. Wie häufig die Schleife wiederholt wird, wird durch den Wert des Arguments `repeats` bestimmt.

```
def printMultiple(toPrint, repeats):
    """ Gibt mit der durch die Variable repeats angegebenen
    Häufigkeit einen String aus """
    count = 0
    while count < repeats:
        print(toPrint)
        count += 1
```

Du kannst auch `return`-Anweisungen und `while`-Schleifen in derselben Funktion verwenden. In den meisten Fällen sollte die `return`-Anweisung dabei außerhalb der Schleife liegen. (Andernfalls nämlich wird die

Schleife durch die return-Anweisung abgebrochen und die Funktion beendet.) Betrachten wir ein Beispiel:

```
def doubleUntilHundred(numberToDb1):  
    """ Verdoppelt eine Zahl, bis sie größer als 100 ist. ↵  
    Rückgabewert ist die  
    Anzahl der Verdopplungen """  
    count = 0  
    while numToDb1 < 100:  
        numberToDb1 = numberToDb1 * 2  
        count += 1  
    return count ⓘ  
  
print(doubleUntilHundred(2))
```

Dieses Programm verdoppelt eine Zahl so lange, bis sie größer als 100 ist. Danach gibt es die Wiederholungshäufigkeit der Schleife zurück ⓘ.

Wie ich in den vorigen Kapiteln beschrieben habe, kannst du auch Funktionsaufrufe in Schleifen ablegen.

Mission 45: Blöcke, überall Blöcke

Wenn du Schleifen innerhalb von Funktionen verwendest, kannst du mithilfe eines Arguments feststellen, wie oft eine Schleife wiederholt wird. Zudem kannst du mit der Funktion `setBlock()` innerhalb der Schleife den Typ des Blocks bestimmen, den du platzieren willst.

Achtung

Das Programm in dieser Mission kann zerstörerisch wirken. Ich empfehle dir deswegen, es in einer neuen Welt auszuprobieren, um deine wertvollen Kreationen zu bewahren.

Du wirst in dieser Mission eine Funktion erstellen, die wahllos Blöcke an zufälligen Orten erstellt. Die Anzahl und der Typ dieser Blöcke werden von den Funktionsargumenten bestimmt.

Listing 8.4 erzeugt irgendwo auf der Karte eine Melone.

blocks
Everywhere.py

```

from mcpi.minecraft import Minecraft
mc = Minecraft.create()
import random

def randomBlockLocations(blockType, repeats):
    count = 0 ❶
    # Hier die Schleife hinzufügen ❷
    x = random.randint(-127, 127)
    z = random.randint(-127, 127)
    y = mc.getHeight(x, z) ❸
    mc.setBlock(x, y, z, blockType)
    count += 1

```

Listing 8.4: Beim Aufruf dieser Funktion entsteht irgendwo im Spiel ein neuer Block.

Kopiere Listing 8.4 in eine neue IDLE-Datei und speichere sie als *blocksEverywhere.py* im Ordner *functions*. Füge dann bei ❷ eine *while*-Schleife innerhalb der Funktion hinzu, sodass der Code wiederholt wird. Die Variable *count* ❶ macht es einfacher, im Blick zu behalten, wie oft die Schleife bereits wiederholt wurde. Vergleiche in der Schleifenbedingung das Argument *repeats* mit der Variablen *count*, um festzulegen, wie oft die Schleife wiederholt werden soll. Ziehe alle Zeilen innerhalb der Funktion nach ❷ ein, damit sie ebenfalls zur Schleife gehören. Die Funktion *getHeight()* stellt sicher, dass der Block über der Erde platziert wird ❸.

Füge abschließend drei Funktionsaufrufe hinzu, um die Blöcke zu erstellen. Dabei soll die erste Funktion zehn Blöcke, die zweite 37 Blöcke und die dritte 102 Blöcke erstellen. Du kannst dafür beliebige Blocktypen auswählen.

Speichere das Programm und führe es aus. Das Programm sollte nun Blöcke irgendwo auf der Karte erstellen. Bild 8.5 zeigt ein Beispiel.

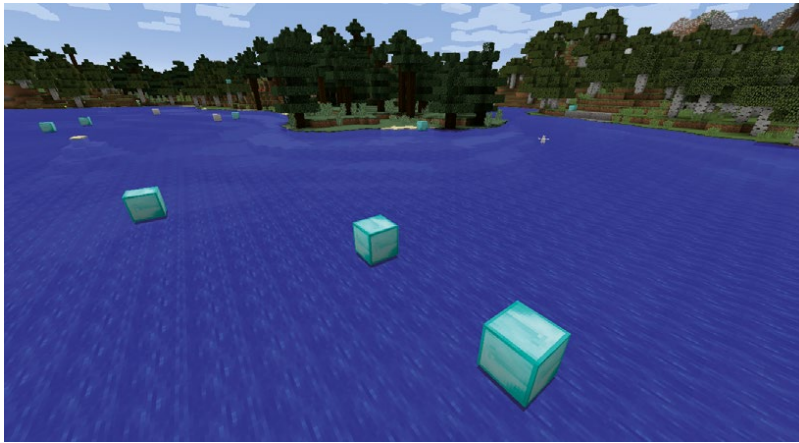


Bild 8.5: Hier sind einige der vom Programm wahllos erstellten Blöcke. Ich habe zu Demonstrationszwecken eine neue Welt erstellt, damit ich keine Gebäude in meiner Spielwelt zerstöre.

Lokale und globale Variablen

Bei der Definition von Funktionen bekommen wir es mit einer neuen Herausforderung zu tun: dem Geltungsbereich einer Variablen. Der *Geltungsbereich* beschreibt, wie dein Programm auf seine Daten zugreifen kann. Dies lernt man am besten, wenn man es in Aktion sieht. Sehen wir uns also einen Beispielcode an. Der folgende Code erhöht die Anzahl der Eier, die du für eine Party besorgt hast:

```
eggs = 12 ❶  
  
def increaseEggs():  
    eggs += 1 ❷  
    print(eggs)  
  
increaseEggs()
```

Es gibt hier zwei Variablen namens eggs: eine außerhalb der Funktion ❶ und eine weitere darin ❷. Das sieht alles gar nicht so schlimm aus, aber Python gibt trotzdem einen Fehler aus. Hier siehst du einen Teil der Fehlermeldung:

```
UnboundLocalError: local variable 'eggs' referenced before assignment
```

Das Problem besteht darin, dass die Variable `eggs` außerhalb der Funktion definiert wurde, du sie aber innerhalb der Funktion hochzuzählen versuchst – und dort kann Python die Variable nicht erkennen. Für Python sind Variablen innerhalb von Funktionen nämlich vollkommen getrennt von solchen außerhalb der Funktion – und zwar auch dann, wenn sie identische Namen haben. Dies ist ganz bewusst so gehalten, um zu verhindern, dass Variablen in verschiedenen Funktionen zufällig den gleichen Namen haben, was unerwartete Bugs zur Folge haben kann.

Im Python-Code gibt es zwei Herangehensweisen an Variablen in einer Datei: Du kannst eine Variable entweder *global* machen, was bedeutet, dass sie im ganzen Programm oder der gesamten Datei gültig ist, oder du machst die Variable *lokal*, woraufhin sie nur noch innerhalb der betreffenden Funktion oder Schleife relevant ist. Mit anderen Worten: Du kannst dieselbe Variable innerhalb und außerhalb einer Funktion verwenden oder aber zwei verschiedene Variablen erstellen, die unterschiedliche Teile des Codes betreffen.

Eine *globale* Variable wird innerhalb wie außerhalb einer Funktion als genau dieselbe Variable behandelt. Das heißt, Änderungen an der Variablen in der Funktion wirken sich auch auf die Variable aus, die außerhalb der Funktion definiert ist, und umgekehrt. Um eine Variable als global zu definieren, verwendest du das Schlüsselwort `global` ❶:

```
eggs = 12

def increaseEggs():
    global eggs ❶
    eggs += 1
    print(eggs)

increaseEggs()
```

In diesem Beispiel ist der Wert von `eggs` bei der Ausgabe 13.

Um eine andere Wirkung zu erzielen, kannst du eine Variable stattdessen auch als *lokal* behandeln. Tust du dies, dann werden Variablen innerhalb und außerhalb der Funktion als verschiedene Variablen betrachtet. Änderungen an der Variablen innerhalb der Funktion haben dann keine Auswirkungen außerhalb der Funktion und umgekehrt. Du könntest den Code wie folgt abändern, um die Variable als lokal zu konfigurieren ❶:

```
eggs = 12

def increaseEggs():
    eggs = 0 ❶
    eggs += 1
    print(eggs) ❷

increaseEggs()
print(eggs) ❸
```

Wenn der Wert von eggs in der Funktion ausgegeben wird ❷, ist er 1, denn der Wert der Variablen eggs außerhalb der Funktion beeinflusst die lokale Variable innerhalb der Funktion nicht. Der Wert von eggs in der Funktion `increaseEggs()` ist 1, und die globale Variable eggs hat weiterhin den Wert 12 ❸.

Mission 46: Es bewegt sich!

Vor einiger Zeit kam ich auf die Idee, einen Block zu programmieren, der sich selbstständig in der Minecraft-Welt bewegt. Der Block soll sich pro Sekunde einen Schritt vorwärtsbewegen. Trifft er auf eine Mauer, einen Baum oder irgendein großes Objekt, dann soll er sich drehen und seinen Weg in eine andere Richtung fortsetzen. Allerdings würde er, wenn er in ein Loch fiele, feststecken und nicht aus ihm entkommen können.

Listing 8.5 zeigt das Gerüst eines Programms für einen Zauberblock, der sich ohne Zutun bewegt.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
```

```
import time
```

```
def calculateMove():
    """ Ändert die Variablen x und z für einen Block. Wenn der Block
        vor unserem Block weniger als 2 Blöcke höher ist, bewegt sich
        unser Block vorwärts, andernfalls versucht er, sich
        nacheinander nach links, rückwärts oder nach rechts zu bewegen. """
    # Hier globale Variablen erstellen ❶

    currentHeight = mc.getHeight(x, z) - 1

    forwardHeight = mc.getHeight(x + 1, z)
    rightHeight = mc.getHeight(x, z + 1)
    backwardHeight = mc.getHeight(x - 1, z)
```

*movingBlock
.py*

```

leftHeight = mc.getHeight(x, z - 1)

if forwardHeight - currentHeight < 3:
    x += 1
elif rightHeight - currentHeight < 3:
    z += 1
elif leftHeight - currentHeight < 3:
    z -= 1
elif backwardHeight - currentHeight < 3:
    x -= 1

y = mc.getHeight(x, z)

pos = mc.player.getTilePos()
x = pos.x
z = pos.z
y = mc.getHeight(x, z)

while True:
    # Blockbewegung berechnen
    calculateMove()

    # Block platzieren
    mc.setBlock(x, y, z, 103)

    # Warten
    time.sleep(1)

    # Block entfernen
    mc.setBlock(x, y, z, 0)

```

Listing 8.5: Leider funktioniert dieser Code erst, nachdem globale Variablen hinzugefügt wurden.

Die Variablen in der Funktion `calculateMove()` sind allerdings noch gar nicht global.

Deine Aufgabe: Vervollständige den Code in Listing 8.5. Kopiere es dann nach IDLE und speichere es als *movingBlock.py* im Verzeichnis *functions*. Füge am Anfang der Funktion den erforderlichen Code hinzu, um *x*, *y* und *z* zu globalen Variablen zu machen. Die globalen Definitionen sollten bei ❶ eingefügt werden.

Hast du die globalen Variablen deklariert, dann führe das Programm aus. Dein Block sollte sich nun in Bewegung setzen. Bild 8.6 zeigt, wie sich der Block zunächst auf eine Mauer zu bewegt und dann darum herumgeht.

Bonus: Ein intelligenter Melonenblock

Wenn du das Programm *movingBlock.py* ausführst, wirst du feststellen, dass der Block sich normalerweise in Richtung der x-Achse bewegt. Das kann dazu führen, dass er in einer Schleife zwischen zwei Blöcken hängen bleibt. Die Ursache hierfür besteht darin, dass der Code die Richtung, die der Block bereits genommen hat, nicht berücksichtigt – der Block wird immer zuerst versuchen, sich entlang der x-Achse zu bewegen. Versuche doch einmal, einen Weg zu finden, wie sich die letzte Richtung, die der Block genommen hat, speichern lässt, und ändere die *i f*-Anweisung so ab, dass der Block sich zuerst in diese Richtung bewegt.

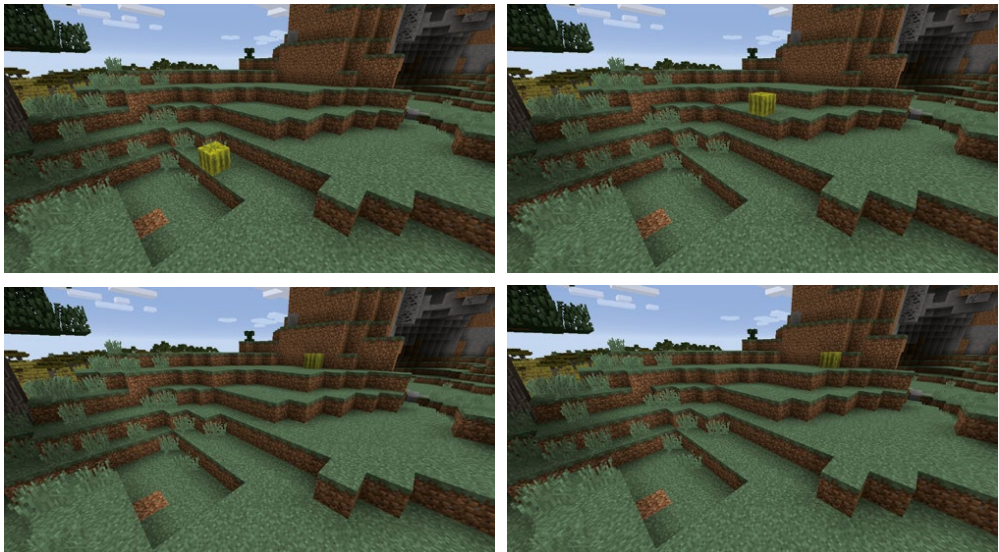


Bild 8.6: Es hat echt Spaß gemacht, der Melone beim Wandern und bei ihrem Versuch zuzusehen, die Mauer zu überwinden.

Was du gelernt hast

Hurra! Du hast in diesem Kapitel gelernt, wie Funktionen erstellt und aufgerufen werden. Mit `return`-Anweisungen kannst du Funktionen nun dazu bringen, Werte zurückzugeben, und du kannst auch Schleifen und `if`-Anweisungen innerhalb von Funktionen nutzen. In Kapitel 9 dreht sich alles um Listen. Mit ihnen kannst du mehrere Datenelemente in einer einzigen Variablen speichern.

Kapitel 9

Treffer, versenkt – dank Listen und Wörterbüchern

Listen – wie beispielsweise Einkaufszettel oder Bedienungsanleitungen – verwenden wir tagtäglich. Zweck der Übung ist es, sich eine Gruppe zusammengehörender Elemente zu merken oder Arbeitsschritte in einer bestimmten Reihenfolge abzuarbeiten. Listen in Python funktionieren ganz ähnlich, denn hiermit wird eine Sammlung von Daten in einer Sequenz abgespeichert. Eine *Liste* kann dabei verschiedene Datentypen speichern, z. B. Strings, Zahlen, boolesche Variablen oder sogar andere Listen.

Normalerweise können Variablen immer nur einen Wert aufnehmen. Insofern sind Listen praktisch, denn sie ermöglichen das Speichern mehrerer Werte in derselben Variablen. Beispiele wären die Zahlen von 1 bis 100 oder die Vornamen deiner besten Freunde. In anderen Programmiersprachen werden Listen auch als *Arrays* bezeichnet.

Auch in deiner Minecraft-Welt können Listen – etwa mit Block-IDs, Koordinaten oder vielen anderen Elementen – deine Möglichkeiten beträchtlich erweitern. Da Listen unterschiedliche Wertetypen in derselben Variablen speichern, bieten sie im Vergleich zu normalen Variablen erheblich mehr Flexibilität.

In diesem Kapitel erfährst du, wie Listen in Verbindung mit der Minecraft-Python-API zum Programmieren eines Minigames, zum Ermitteln der Positionshöhe, zum Erstellen eines Fortschrittsbalkens und zum Schreiben eines Programms verwendet werden, das den Spieler wahllos hin und her bewegt.

Listen verwenden

Das Erstellen einer Liste in Python ist ein Kinderspiel. Zum Definieren setzt du einfach eine beliebige Anzahl Werte in eckige Klammern. Du kannst die Klammern auch leer lassen, dann spricht man von einer *leeren Liste*. Die einzelnen Elemente einer Liste werden durch Kommata getrennt.

So könnte beispielsweise eine Liste mit Zutaten für eine Nudelsuppe wie folgt aussehen:

```
>>> noodleSoup = ["water", "soy sauce", "spring onions", "noodles",  
"beef"]
```

Die Liste `noodleSoup` enthält verschiedene Elemente, und sie alle sind Strings.

Eine leere Liste erstellst du wie folgt:

```
>>> emptyList = []
```

Leere Listen kannst du verwenden, wenn du die Listenelemente erst im weiteren Verlauf deines Programms hinzufügen möchtest.

Du kannst beliebige Datentypen – auch in gemischter Form – in deiner Liste ablegen. Eine Liste kann also auch gleichzeitig Integer und Strings enthalten:

```
>>> wackyList = ["cardigan", 33, "goofballs"]
```

Listen können sehr lang werden und sind dann zumindest für uns Menschen schwierig zu lesen. Du kannst lange Listen in Python allerdings über mehrere Zeilen schreiben, was die Erfassung erleichtert. Für die Ausführung des Codes spielt es keine Rolle, ob Listen über mehrere Zeilen verteilt sind. So funktioniert das folgende Format der Zutatenliste genau so wie die obige Liste:

```
>>> noodleSoup = ["water",  
                  "soy sauce",  
                  "spring onions",  
                  "noodles",  
                  "beef"]
```

Als Nächstes wollen wir uns ansehen, wie Elemente in einer Liste aufgerufen und geändert werden können.

Listenelemente aufrufen

Um auf einen Wert in einer Liste zuzugreifen, referenzierst du seine Position in der Liste. Diese Position wird als *Index* bezeichnet. In unserem Beispiel erfolgt der Zugriff auf das erste Element der Liste wie folgt:

```
>>> print(noodleSoup[0])  
water
```

Beachte, dass der erste Index in einer Liste stets 0 ist. Der Index des zweiten Elements ist 1, der des dritten 2 usw. Das liegt daran, dass Computer bei Verwendung von Listen bei 0 zu zählen beginnen.

Ab 0 zu zählen mag albern klingen, aber es gibt einen guten Grund hierfür. Frühe Computer waren sehr langsam und hatten extrem wenig Arbeitsspeicher. Deswegen brachte das Nummerieren von Indizes, das schon mit 0 beginnt, sowohl einen Geschwindigkeits- als auch einen Effizienzgewinn. Zwar sind moderne Computer heutzutage sehr viel schneller, aber noch immer beginnen sie mit dem Zählen bei 0.

Wichtig anzumerken ist auch, dass eine Fehlermeldung ausgegeben wird, wenn du versuchst, einen Listenindex aufzurufen, der größer ist als die Anzahl der Elemente in der Liste. Die folgende Zeile versucht, das Element an der Indexposition 5 auszugeben:

```
>>> print(noodleSoup[5])
```

Hier ist ein Teil der Fehlermeldung:

```
IndexError: list index out of range
```

Dem `IndexError` entnehme ich, dass an der Indexposition, die ich aufrufen möchte, keine Daten vorhanden sind. Das ist deswegen so, weil diese Indexposition außerhalb der Listenlänge liegt. Python kann keinen Wert zurückgeben, der gar nicht existiert!

Listenelemente ändern

Du kannst einzelne Elemente in einer Liste ebenso ändern wie ganz normale Variablenwerte. Das liegt daran, dass Listen *veränderlich* sind, also bearbeitet werden können. Um ein Element in einer Liste zu ändern, gibst du die Indexposition an und legst den Wert dann auf die gleiche Weise fest wie bei einer Variablen (nämlich mit einem Gleichheitszeichen).

Ersetzen wir das Rindfleisch in unserer Suppe durch Huhn. Rindfleisch ist das fünfte Element der Liste, hat also den Index 4 (wie gesagt: wir zählen ab 0). Element 4 können wir ganz einfach ändern:

```
>>> noodleSoup[4] = "chicken"
```

Und jetzt wollen wir sehen, was für coole Sachen man mit Listen in Minecraft machen kann.

Mission 47: Oben und unten

Wenn ich meine Minecraft-Welt erforsche, dann ist es interessant, den bisher zurückgelegten Weg zu betrachten. Ich liebe es, die Spielwelt zu durchforsten – von den höchsten Gipfeln bis zu den tiefsten Höhlen. Wenn ich mit Freunden spiele, machen wir manchmal einen Wettbewerb, wer in der kürzesten Zeit den höchsten oder niedrigsten Punkt im Spiel erreicht. Damit niemand flunkert, habe ich ein Programm geschrieben, das die höchste und die niedrigste y-Koordinate speichert, die der Spieler innerhalb von 60 Sekunden erreicht.

Wenn ich also das Programm ausführe, dann nennt es mir den höchsten und den niedrigsten Punkt, den ich im Zeitraum einer Minute aufgesucht habe. Listing 9.1 enthält wie üblich das Codegerüst. Kopiere es in eine neue Datei und speichere diese als *highAndLow.py* in einem neuen Ordner namens *lists*.

```

from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

heights = [100, 0] ❶
count = 0

while count < 60:
    pos = mc.player.getTilePos()

    if pos.y < heights[0]:
        # Niedrigste Position für die Variable y ermitteln ❷
    elif pos.y > heights[1]:
        # Höchste Position für die Variable y ermitteln ❸

    count += 1
    time.sleep(1)

mc.postToChat("Lowest: ")      # Niedrigste Position ausgeben ❹
mc.postToChat("Highest: ")    # Höchste Position ausgeben ❺

```

highAndLow
.py

Listing 9.1: Anfang des Codes zum Ermitteln der niedrigsten und höchsten Positionen, die der Spieler besucht hat

Das Programm speichert die niedrigste und die höchste y-Koordinate, die du besucht hast, in einer Liste namens `heights` ❶. Das erste Element dieser Liste (Indexposition 0) nimmt dabei die niedrigste, das zweite (Indexposition 1) die höchste Koordinate auf. Beginnen müssen wir mit einem höchsten »niedrigsten« und einem niedrigsten »höchsten« Wert, damit die Position des Spielers bei der ersten Programmausführung die neuen niedrigsten oder höchsten Werte bestimmt und im Chat angezeigt wird. Als Vorgabe habe ich 100 für den niedrigsten und 0 für den höchsten Wert verwendet.

Die `while`-Schleife wird 60 Sekunden lang ausgeführt und aktualisiert einmal pro Sekunde die Werte in `heights`. Mit der `if`-Anweisung wird überprüft, ob die aktuelle Höhe des Spielers niedriger ist als der niedrigste in der Liste gespeicherte Wert ❷. Danach wird mit der `elif`-Anweisung festgestellt, ob die aktuelle Höhe größer ist als der in der Liste gespeicherte höchste Wert ❸.

Zur Vervollständigung des Codes musst du den Wert für die niedrigste Höhe (`height[0]`) bei ❷ auf den Wert von `pos.y` setzen. Da, wie du bereits weißt, die Werte in Listen genau so wie Variablen festgelegt werden, muss die Codezeile wie folgt aussehen: `height[0] = pos.y`. Außer-

dem musst du auch den Wert für die größte Höhe (`height[1]`) auf den Wert von `pos.y` setzen ③.

Schließlich müssen die Werte der niedrigsten ④ und höchsten Positionen ⑤ in den letzten beiden Programmzeilen ausgegeben werden. Hierzu benötigst du Zugriff auf die Indexpositionen der niedrigsten und der höchsten Position in der Liste `heights` (wie bekannt ist Index 0 die niedrigste und Index 1 die höchste Position).

Führe das Programm aus und laufe ein wenig im Spiel umher. Probiere, möglichst hohe und niedrige Positionen zu erreichen. Nach 60 Sekunden endet die Schleife, und das Programm zeigt die erreichten Werte an. Wiederhole das Ganze ein paar Mal und versuche, deine Rekorde zu überbieten!

Bild 9.1 zeigt einen meiner Versuche.



Bild 9.1: Die niedrigste erreichte y-Koordinate war 15, die höchste 102.

Bonus: Ein unerwarteter Bug

In `highAndLow.py` sind die Standardwerte für die niedrigste und die höchste Position auf 100 bzw. 0 festgelegt. Das ist unproblematisch, solange du dich nicht höher als 100 oder niedriger als 0 bewegst. Andernfalls jedoch wird der Wert sich nicht mehr ändern, und somit kann dein Programm ungenau werden. Hast du eine Ahnung, wie du das beheben kannst?

Listen manipulieren

Es gibt es eine Reihe von Funktionen, um Listen zu manipulieren. Hierzu gehören Standardfunktionen wie das Einfügen oder Entfernen von Listenelementen.

Element anhängen

Mit der Funktion `append()` kannst du am Ende der Liste ein Element anhängen. Dazu musst du nur den Wert des Elements, das du hinzufügen willst, als Argument übergeben.

Unsere Nudelsuppe wäre noch besser, wenn wir ein bisschen Gemüse hinzugeben. Das tun wir mit `append()`:

```
>>> noodleSoup.append("vegetables")
```

Ab sofort enthält die Liste `noodleSoup` einen String `"vegetables"` als letztes Listenelement.

Das Anhängen von Elementen ist sehr praktisch, wenn man mit einer leeren Liste beginnt. Mit `append()` gelangt das erste Element in deine Liste:

```
>>> food = []  
>>> food.append("cake")
```

Element einfügen

Du kannst Elemente aber nicht nur an die Liste anhängen, sondern auch mittendrin einfügen. Mit der Funktion `insert()` wird ein Element zwischen zwei bereits vorhandenen Elementen eingefügt. Gleichzeitig werden die Indexpositionen aller Elemente, die hinter dem neu eingefügten Element stehen, entsprechend verschoben.

Die Funktion nimmt zwei Argumente entgegen: die Indexposition, an der das Element eingefügt werden soll, und den einzufügenden Wert.

Hier ist unsere aktuelle Nudelsuppe:

```
>>> noodleSoup = ["water", "soy sauce", "spring onions", "noodles",  
"beef", "vegetables"]
```

Nun soll an der dritten Indexposition in der Liste Pfeffer eingefügt werden:

```
>>> noodleSoup.insert(3, "pepper")
```

Und so sieht die geänderte Liste nach dem Einfügen aus:

```
["water", "soy sauce", "spring onions", "pepper", "noodles", "beef",  
"vegetables"]
```

Wenn du ein Element an einer Indexposition einfügst, die größer ist als die gesamte Länge der Liste, dann wird das Element an letzter Stelle angehängt. Umfasst also deine Liste sieben Elemente und möchtest du an Position 10 etwas einfügen, kommt dein Element ans Listenende.

```
>>> noodleSoup.insert(10, "salt")
```

Nach Ausführung dieses Codes ist das letzte Element in der Liste "salt":

```
["water", "soy sauce", "spring onions", "pepper", "noodles", "beef",  
"vegetables", "salt"]
```

Du wirst feststellen, dass sich das Salz jedoch nicht an Indexposition 10 befindet, sondern – logisch! – an Position 7.

Elemente löschen

Unter Umständen kann es auch notwendig sein, ein Element aus einer Liste zu löschen. Hierfür verwendest du das Schlüsselwort `del`. Dieses stellst du vor den Namen der Liste. Danach gibst die Indexposition des Elements, das du löschen willst, in eckigen Klammern an.

Um beispielsweise das Element "beef" zu löschen, das sich gegenwärtig an Indexposition 5 in der Liste `noodleSoup` befindet, tust du Folgendes:

```
>>> del noodleSoup[5]
```

Du kannst das Schlüsselwort `del` auch in Kombination mit der Funktion `index()` verwenden, wenn du die Indexposition eines Werts ermitteln und diesen dann löschen möchtest:

```
>>> beefPosition = noodleSoup.index("beef")  
>>> del noodleSoup[beefPosition]
```

Nach dem Löschen eines Elements ändern sich die Indexpositionen in der Liste. Unsere Beispielliste sieht nach dem Löschen von "beef" wie folgt aus:

```
["water", "soy sauce", "spring onions", "pepper", "noodles",  
"vegetables", "salt"]
```

Die Indexposition von "vegetables" hat sich von 6 auf 5 geändert, und "salt" ist von Position 7 auf 6 gerutscht. Naheliegenderweise betrifft dies nur Indizes hinter dem gelöschten Element – Indizes, die davor stehen, sind von Änderungen nicht betroffen. Denke daran, wenn du Elemente aus deinen Listen löschst.

Mission 48: Fortschrittsbalken

In dieser Mission wollen wir einen Fortschrittsbalken in Minecraft basteln. Dazu verwenden wir Listenfunktionen. Was ein Fortschrittsbalken ist, weißt du bestimmt: Ein solcher Balken wird beispielsweise angezeigt, wenn du eine Datei aus dem Internet herunterlädst. Und in Rollenspielen wird damit veranschaulicht, wie lange es noch bis zum Abschluss des aktuellen Levels dauert.

In unserem Programm zählt der Fortschrittsbalken 10 Sekunden ab. Beim Programmstart besteht der Balken zunächst aus Glasblöcken. Danach wird nach jeder verstrichenen Sekunde einer dieser Glasblöcke durch einen Lapislazuliblock ersetzt. Bild 9.2 zeigt den Fortschrittsbalken nach den ersten 5 Sekunden.



Bild 9.2: Fortschrittsanzeige bei 50 % (Fünf von zehn Blöcken sind bereits Lapislazuli.)

Öffne IDLE und erstelle eine neue Datei. Speichere sie als *progressBar.py* im Ordner *lists*. Listing 9.2 zeigt eine erste unvollständige Version des Programms. Kopiere diese in deinen Texteditor.

progressBar
.py

```

from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

pos = mc.player.getTilePos()
x = pos.x + 1
y = pos.y
z = pos.z

# 10 Glasblöcke (ID 20) zur leeren Liste hinzufügen
blocks = [ ] ❶
barBlock = 22 # Lapislazuli

count = 0
while count <= len(blocks):

    mc.setBlock(x, y, z, blocks[0])
    mc.setBlock(x, y + 1, z, blocks[1])
    mc.setBlock(x, y + 2, z, blocks[2])
    # setBlock() für verbleibende Blöcke in der Liste hinzufügen ❷

    count += 1

    # Letzten Block in der Liste löschen ❸

    # Lapislazuliblock an der ersten Listenposition einfügen ❹

time.sleep(2)

```

Listing 9.2: Unvollständiger Code für den Fortschrittsbalken

Zum Vervollständigen des Programms in Listing 9.2 führst du folgende Schritte aus:

1. Füge bei ❶ zehn Glasblöcke (ID 20) zur leeren Liste `blocks` hinzu.
 2. Lege mit der Funktion `setBlock()` alle zehn Blöcke ❷ auf Grundlage der Liste in der Spielwelt fest. Die ersten drei Blöcke sind bereits vorhanden.
 3. Schreibe eine Anweisung, die den letzten Block in der Liste (Indexposition 9) löscht ❸. Du kennst ja bereits das Schlüsselwort `del`, das du hierzu verwendest.
-

4. Füge einen neuen Lapislazuliblock am Anfang der Liste ein ④. Verwende die Funktion `insert()` in Verbindung mit der Variablen `barBlock`, um einen neuen Lapislazuliblock an Indexposition 0 einzufügen.

Ich habe im Code Kommentare eingefügt, damit du siehst, an welcher Stelle welcher Schritt umgesetzt werden muss.

Bonus: Auf und wieder ab

In der aktuellen Variante zählt die Fortschrittsanzeige in *progress-Bar.py* nur hoch und endet, wenn sie vollständig ist. Kannst du dir denken, wie sie abwärts zählen könnte?

Strings als Listen behandeln

Auch Strings lassen sich wie Listen behandeln, denn schließlich ist ein String nichts anderes als eine *Zeichenfolge*. Du kannst also einzelne Zeichen in einem String über ihren Index abrufen. Allerdings ist es nicht möglich, einzelne Zeichen mit den Listenfunktionen `append()` oder `insert()` gezielt zu ändern, denn Strings sind *unveränderlich* – sie können nicht manipuliert werden.

Der folgende Code gibt den zweiten Buchstaben des Strings "Grape" aus:

```
>>> flavor = "Grape"
>>> print(flavor[1])
r
```

Der Code veranschaulicht, dass der Zugriff auf einzelne Teile eines Strings auf gleiche Weise erfolgt wie auf Elemente einer Liste. Du könntest beispielsweise die ersten Buchstaben des Vor- und des Nachnamens eines Benutzers extrahieren, um seine Initialen auszugeben:

```
>>> firstName = "Lyra"
>>> lastName = "Jones"
>>> initials = firstName[0] + " " + lastName[0]
>>> print(initials)
L J
```

Der neue String "L J", den du erhältst, wenn du Teile des Strings anhand ihrer Indexpositionen aufrufst, wird als *Substring* bezeichnet. Übrigens (und nur der Vollständigkeit halber): Auch die Indizes eines Strings beginnen ihre Zählung bei 0!

Tupel

Tupel sind eine Form einer unveränderlichen Liste. Wie bei anderen Listen handelt es sich auch hierbei um eine Folge von Elementen beliebiger Variablentypen. Allerdings verwenden Tupel runde anstelle eckiger Klammern und trennen die einzelnen Elemente mit Kommata.

Ein Beispiel: Angenommen, der einzige Olympionike einer kleinen Nation listet eine Reihe von Weitsprungergebnissen auf, die er in seinem hoffnungslos unterfinanzierten Trainingsprogramm erzielen konnte:

```
>>> distance = (5.17, 5.20, 4.56, 5.64, 9.58, 6.41, 2.20)
```

Wäre der Sportler nur einmal gesprungen, dann könnte man auch daraus ein Tupel erstellen – aber eben eines mit nur einem einzigen Wert. Wichtig ist, dass du auch dann ein Komma setzen musst, wenn du ein Tupel mit nur einem Wert notierst:

```
>>> distance = (5.17,)
```

Wenn du ein Tupel definierst, sind die Klammern optional, d. h., du könntest auch einfach Kommata zwischen die Werte setzen:

```
>>> distance = 5.17, 5.20, 4.56, 5.64, 9.58, 6.41, 2.20
```

Um Werte aus Tupeln abzurufen, verwendest du die von normalen Listen her bekannte Notation mit eckigen Klammern. Wir wollen einmal den Wert an Index 1 des Tupels *distance* der Variablen *jump* zuweisen:

```
>>> jump = distance[1]
>>> print(jump)
5.20
```

Der wesentliche Unterschied zwischen Listen und Tupeln besteht darin, dass Tupel unveränderlich sind – ihre Inhalte können nicht modifiziert werden. Du kannst also weder Elemente ans Ende eines Tupels anhängen

gen noch Elemente einfügen, löschen oder ändern. Du verwendest Tupel anstelle von Listen, wenn dein Programm die Werte der enthaltenen Elemente nicht ändern muss.

Variablen mit Tupeln festlegen

Eine praktische Eigenschaft von Tupeln ist, dass du mit ihnen mehrere Variablen gleichzeitig festlegen kannst. Dies spart Zeit, und du kannst zusammengehörende Variablen damit auch zusammenhalten.

In der Regel würdest du ein Tupel wie eine Liste referenzieren, nämlich indem du einen einzelnen Variablennamen angibst:

```
measurements = 6, 30
```

Wir wollen jedoch einmal annehmen, dass die Werte nicht in einer, sondern in zwei Variablen gespeichert werden sollen. Die hierfür erforderliche Syntax ist nicht allzu komplex. Du musst nur die Variablennamen durch Kommata trennen, ein Gleichheitszeichen angeben und die Tupel dann rechts davon aufführen. Jeder Tupelwert wird dann der Variable an der entsprechenden Position zugewiesen. Sehen wir es uns an.

Im folgenden Beispiel werden die beiden Variablen `width` und `height` auf die Werte 6 bzw. 30 gesetzt:

```
width, height = 6, 30
```

Und schon haben wir zwei Variablen: `width` mit dem Wert 6 und `height` mit dem Wert 30. Das alles haben wir in nur einer Codezeile bewerkstelligt.

Mission 49: Verschieben

Das Festlegen von Variablen mit Tupeln ist eine Möglichkeit, schnell und einfach Platz in deinen Programmen einzusparen. Außerdem ist es nützlich, um ähnliche Variablen an zentraler Stelle in deinem Programm festzulegen. In diesem Buch habe ich etwa bislang folgenden Code verwendet, um die Werte für die Variablen `x`, `y` und `z` festzulegen:

```
x = 10  
y = 11  
z = 12
```

Stattdessen könnte ich diese Werte aber auch in einer Zeile mit einem Tupel zuweisen:

```
x, y, z = 10, 11, 12
```

Wir wollen diese neuen Möglichkeiten der Codenotation jetzt einmal umsetzen. Deine Mission besteht darin, ein Programm zu entwerfen, das den Spieler in kleinen Schritten zufällig in deiner Welt umherbewegt – so, als ob er darin Schlittschuh laufen würde. In Listing 9.3 habe ich den Anfang dieses Programms aufgeführt. Es fehlt aber noch allerlei, und das sollst du ergänzen.

sliding.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import random
import time

# Position des Spielers abrufen ❶

# Variablen x, y und z in einer Zeile mit einem Tupel festlegen ❷

while True:
    x += random.uniform(-0.2, 0.2) ❸
    # Variable z mit einem zufällig generierten Float ändern
    z += ❹
    y = mc.getHeight(x, z)

    mc.player.setPos(x, y, z)
    time.sleep(0.1)
```

Listing 9.3: Anfang des Codes, durch den der Spieler auf der Karte umhergeschoben wird

Kopiere Listing 9.3 in eine neue Datei und speichere sie als *sliding.py* im Ordner *lists*. Zur Vervollständigung des Programms musst du die Ausgangsposition des Spielers abrufen ❶ und die Werte der Variablen *x*, *y* und *z* festlegen ❷. Hierzu verwendest du ein Tupel. Das Programm benutzt die Funktion `uniform()` ❸, die der Funktion `randint()` (siehe »Mehr Spaß mit Zufallszahlen« auf Seite 79) ähnelt, aber einen Float-Zufallswert anstelle eines Integers zurückgibt. Mit der Funktion `uniform()` änderst du die Variable *z* in der Schleife ❹. Für die Variable *x* habe ich das bereits erledigt ❸.

Bild 9.3 zeigt meinen Spieler, wie er durch das Spiel gleitet.



Bild 9.3: *Ich rutsche langsam rückwärts durch meinen Garten.*

Bonus: Blöcke verschieben

Das Programm *sliding.py* lässt den Spieler auf zufällige Weise im Spiel umhergleiten. Denke doch einmal darüber nach, wie man das Programm so abändern könnte, dass stattdessen ein Block verschoben wird.

Tupel zurückgeben

Einige der in Python integrierten Funktionen geben ein Tupel zurück. Auch Funktionen, die du selbst definierst, können Tupel zurückgeben. Hierzu nennst du ein Tupel nach dem Schlüsselwort `return`. Erstelle doch einmal eine Funktion, die ein Datum in ein Tupel umwandelt. Wir übergeben das Datum als Stringargument, und die Funktion gibt Jahr, Monat und Tag als Elemente eines Tupels zurück. Hier ist der Code dazu:

```
def getDateTuple(dateString):  
    year = int(dateString[0:4])  
    month = int(dateString[5:7])  
    day = int(dateString[8:10])  
    return year, month, day
```

Wenn wir die Funktion aufrufen und ein Datum als String übergeben, wird ein Tupel zurückgegeben, das Jahr, Monat und Tag in folgender Reihenfolge enthält:

```
>>> getDateTuple("1997-09-27")  
(1997, 9, 27)
```

Nach dem Aufruf der Funktion können wir das zurückgegebene Tupel in beliebiger Form speichern. Der folgende Code legt jeden Wert in einer separaten Variablen ab:

```
year, month, day = getDateTuple("1997-09-27")
```

So können wir nun Datumsstrings unkompliziert in einzelne Variablenwerte umwandeln. Bei meiner Arbeit als Softwareentwickler verwende ich solche Codes ständig.

Weitere nützliche Listenfunktionen

Mit Listen kann man noch vieles andere erledigen. In diesem Abschnitt erläutere ich, wie du die Länge einer Liste ermittelst, ein zufälliges Listenelement auswählst und mit einer `i`-Anweisung überprüfst, ob ein Wert in einer Liste enthalten ist.

Listenlänge

Die Funktion `len()` bietet die Möglichkeit, die Länge einer beliebigen Liste in Python schnell zu ermitteln. Sie nimmt den Namen der Liste entgegen und gibt die Anzahl der enthaltenen Elemente zurück. Wir wollen uns das mal in Aktion ansehen:

```
>>> noodleSoup = ["water", "soy sauce", "spring onions", "noodles",  
"beef", "vegetables"]  
>>> print(len(noodleSoup))  
6
```

Zwar beginnt Python mit der Indexzählung bei 0, aber die Anzahl der in einer Liste enthaltenen Elemente wird mit normalen Zahlenwerten angegeben. Der höchste Index in dieser Liste ist 5, aber Python weiß natürlich, dass insgesamt sechs Elemente vorhanden sind.

Mission 50: Blocktreffer

Die Minecraft-Python-API bietet eine praktische Funktion, die eine Liste aller Positionen zurückgibt, die du mit deinem Schwert bereits getroffen hast. Mithilfe der Elemente in dieser Liste kannst du die Koordinaten von Blöcken abrufen, die bereits getroffen wurden. Warum das so praktisch ist, wirst du im weiteren Verlauf dieses Kapitels und des gesamten Buchs erfahren.

Du kannst aber jetzt schon ein kleines Minigame erstellen, das angibt, wie viele Blöcke du in einer Minute mit dem Schwert triffst. Und genau das werden wir jetzt tun. Du kannst dann gegen einen Freund spielen, und ihr könnt versuchen, eure Bestmarken gegenseitig zu übertreffen. Außerdem kann das Programm erweitert werden, indem du etwa den Highscore speicherst.

Bild 9.4 zeigt das Programm bei der Ausführung.



Bild 9.4: In 60 Sekunden habe ich 197 Blöcke getroffen.

Für dieses Spiel ist nicht allzu viel Code erforderlich. Die Codestruktur lässt sich wie folgt zusammenfassen:

1. Stelle eine Verbindung zum Minecraft-Spiel her.
2. Warte 60 Sekunden.
3. Rufe die Liste mit den Blocktreffern ab.
4. Zeige die Länge der Blocktreffer im Chat an.

Der folgende Code enthält den einzigen Teil, den du noch nicht kennst: das Abrufen der Liste der Blocktreffer aus dem Spiel.

```
blockHits = mc.events.pollBlockHits()
```

Dieser Code verwendet die Funktion `pollBlockHits()`, um eine Liste aller Blocktreffer zurückzugeben, und speichert diese Liste dann in einer Variablen namens `blockHits`. Diese Variable funktioniert wie jede andere Liste auch, d. h., du kannst Daten von Indexpositionen oder auch die Länge der Liste abrufen.

Wenn du dieses Spiel spielst, musst du mit der rechten Maustaste auf die Blöcke klicken, damit sie gezählt werden. Das liegt daran, dass `pollBlockHits()` nur Blöcke registriert, die mit der *rechten* Maustaste angeklickt wurden. In der PC-Version von Minecraft sieht das Rechtsklicken mit dem Schwert zwar eher so aus, als würdest du dich verteidigen, aber tatsächlich werden die so getroffenen Blöcke registriert. Bild 9.5 zeigt, wie das aussieht. Denke also daran, nur mit der rechten Maustaste zu klicken – Linksklicks mit deinem Schwert werden nicht gezählt. Gleiches gilt

übrigens auch, wenn du mit einem anderen Item als dem Schwert rechtsklickst. Allerdings kannst du jedes beliebige Schwert verwenden – ob aus Eisen, Gold oder Diamant.



Bild 9.5: Wenn ich rechtsklicke, hält der Spieler sein Schwert so.

Wenn du die Ausgabe der Liste anzeigst, sollte diese ungefähr so aussehen, auch wenn sich die tatsächlichen Werte jedes Mal abhängig davon ändern, wo du getroffen hast:

```
[BlockEvent(BlockEvent.HIT, 76, -2, 144, 1, 452),  
BlockEvent(BlockEvent.HIT, 79, -2, 145, 1, 452),  
BlockEvent(BlockEvent.HIT, 80, -3, 147, 1, 452),  
BlockEvent(BlockEvent.HIT, 76, -3, 149, 1, 452)]
```

Die Liste gibt die Details von vier Blocktreffern aus. Jedes Element enthält die Koordinaten der Treffer. Wie du auf diese Koordinaten zugreifst, erfährst du in Mission 55 (Seite 254).

Damit du einen einfacheren Einstieg in das Programm bekommst, habe ich die Grundstruktur bereits in Listing 9.4 notiert.

swordHits.py

```
# Verbindung mit Minecraft herstellen
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

# 60 Sekunden warten
time.sleep(60)

# Liste mit den Blocktreffern abrufen
blockHits = ❶

# Länge der Blocktrefferliste im Chat anzeigen
blockHitsLength = ❷
mc.postToChat("Your score is " + str(blockHitsLength))
```

Listing 9.4: Anfang des Blocktrefferspiels

Zum Vervollständigen dieses Programms erstellst du eine neue Datei in IDLE und kopierst Listing 9.4 dorthin. Speichere diese Datei dann unter dem Namen *swordHits.py* im Ordner *lists*. Lege die Variable `blockHits` mit der Funktion `pollBlockHits()` ❶ fest. Die Variable `blockHitsLength` legst du fest, indem du die Länge der Variablen `blockHits` abrufst ❷.

Zufälliges Element auswählen

Du hast vielleicht schon gemerkt, dass ich Zufallselemente gern in meinen Programmen einsetze. Der Zufall macht ein Programm bei jeder Ausführung in gewissen Grenzen unvorhersehbar.

Wenn du Listen verwendest, wirst du hier und da auch einmal auf ein zufällig ausgewähltes Listenelement zugreifen wollen. Du könntest beispielsweise irgendeinen Block aus einer Liste mit Blöcken auswählen.

Die Funktion `choice()` im Modul `random` ist die richtige Wahl für die zufällige Auswahl eines Elements. Die Funktion nimmt ein Argument – die Liste, die du verwenden willst – entgegen und gibt dann ein zufällig ausgewähltes Element aus dieser Liste zurück.

In Listing 9.5 enthält die Liste `colors` die Namen verschiedener Farben. Mit der Funktion `choice()` wird eine dieser Farben zufällig ausgewählt und dann ausgegeben:

```
import random
colors = ["red", "green", "blue", "yellow", "orange", "purple"]
print(random.choice(colors))
```

Listing 9.5: Dieser Code gibt eine zufällige Farbe aus einer Liste aus.

Wenn du den Code ausführst, dann gibt das Programm ein zufällig ausgewähltes Listenelement aus.

Mission 51: Irgendein Block

In Minecraft kann das zufällige Auswählen einer Block-ID aus einem Wertebereich zu Problemen im Programm führen, weil einige IDs keinem Blocktyp zugeordnet sind. Eine Lösung des Problems besteht darin, eine Liste gültiger Blöcke zu verwenden, aus denen dann ausgewählt werden kann. Mit Listen kannst du die Anzahl der Elemente beschränken und dann eines davon mit der Funktion `choice()` zufällig auswählen.

Deine Aufgabe besteht darin, eine Liste mit Block-IDs zu erstellen, einen Block zufällig aus dieser Liste auszuwählen und diesen dann an der Position des Spielers zu platzieren. Du kannst Listing 9.5 als Ausgangspunkt hierfür verwenden.

Erstelle zunächst eine Liste der Block-IDs. Danach wählst du mit der Funktion `random.choice()` einen Block aus der Liste aus. Im dritten Schritt platzierst du dann mit der Funktion `setBlock()` diesen zufällig ausgewählten Block in deinem Minecraft-Spiel.

Speichere das Programm als *randomBlock.py* im Ordner *lists*.

Du kannst beliebig viele Blöcke in deine Liste aufnehmen. Für meine Liste wähle ich fünf Blöcke aus, darunter Melone, Diamant und Gold. Was passiert, wenn du das Programm ausführst, siehst du in Bild 9.6.



Bild 9.6: Ein Goldblock – ganz zufällig ausgewählt

Liste kopieren

Das Kopieren von Listen ist in den meisten Programmiersprachen ziemlich knifflig. Listenvariablen selbst enthalten ja keine Werte, sondern nur Verweise auf eine Adresse im Speicher des Computers, an der weitere Verweise auf die Werte zu finden sind, die in der Liste vorkommen. Zwar kümmert sich dein Computer hinter den Kulissen ganz autark um diese Funktionalität, doch trotzdem solltest du wissen, wie sie funktioniert, denn das macht dich zu einem besseren Programmierer. Die Speicheradresse einer Liste kannst du mit der Funktion `id()` anzeigen:

```
>>> cake = ["Eggs",  
            "Butter",  
            "Sugar",  
            "Milk",  
            "Flour"]  
>>> print(id(cake))
```

Auf meinem Computer hat dieser Code 3067456428 ausgegeben. Dieser Wert ist die Stelle im Speicher, an der `cake` gespeichert ist. Wenn du diesen Code auf deinem Computer ausführst, wird höchstwahrscheinlich eine andere Zahl ausgegeben, da der Wert bei dir an einer anderen Speicherposition abgelegt sein dürfte.

Du musst dieses Verhalten nicht vollständig nachvollziehen können, solltest aber wissen, dass es Auswirkungen hat, wenn du eine Liste auf eine andere Variable kopierst. Es werden nämlich nicht, wie man vielleicht

erwarten würde, die Listenwerte in die neue Variable kopiert, sondern der Speicherort der Ursprungsliste. Wenn du also einen Listenwert in der einen Variablen änderst, dann hat das auch Auswirkungen auf die andere Variable.

Das folgende Programm beispielsweise erstellt eine Liste namens `cake` und setzt dann den Wert von `chocolateCake` auf denselben Wert wie `cake`. Danach wird das Element "Chocolate" zur Liste `chocolateCake` hinzugefügt:

```
>>> cake = ["Eggs",
            "Butter",
            "Sugar",
            "Milk",
            "Flour"]

>>> # Liste in einer zweiten Variablen speichern
>>> chocolateCake = cake
>>> chocolateCake.append("Chocolate")
```

Bedauerlicherweise wird "Chocolate" auch zur Liste `cake` hinzugefügt, obwohl das gar nicht deine Absicht war. Das Problem ist zu erkennen, wenn man die Listen ausgibt:

```
>>> print(cake)
['Eggs', 'Butter', 'Sugar', 'Milk', 'Flour', 'Chocolate']
>>> print(chocolateCake)
['Eggs', 'Butter', 'Sugar', 'Milk', 'Flour', 'Chocolate']
```

Es tritt auf, weil die Variablen den Speicherort der Liste speichern, nicht die einzelnen Listenelemente selbst.

Eine einfache Möglichkeit, dieses Problem zu umgehen, besteht in der Verwendung von Listen-Slices. Das Slicing (dt. *in Scheiben schneiden*) in Python ist tatsächlich dem Schneiden von Brot und anderen Lebensmitteln in handliche Portionen ganz ähnlich. Beim Slicing von Listen wird ein Teil der Liste »herausgeschnitten«. Listen-Slices können auf bestimmte Listenelemente beschränkt sein, aber in unserem Fall verwenden wir ein Slice, das alle Elemente der Liste enthält. Um die Liste `cake` auf die Variable `chocolateCake` zu kopieren, verwendest du folgenden Code:

```
>>> chocolateCake = cake[:]
```

Die Variable `chocolateCake` enthält nachfolgend die Werte der Liste `cake`, aber diese befinden sich nun an einer anderen Speicheradresse.

Der Code für die Kuchenzutaten kann mit dem Listen-Slice korrigiert werden:

```
>>> cake = ["Eggs",
            "Butter",
            "Sugar",
            "Milk",
            "Flour"]

>>> # Liste in einer zweiten Variablen speichern
>>> chocolateCake = cake[:] ❶
>>> chocolateCake.append("Chocolate")
```

Wie du siehst, wurden die Elemente aus `cake` mit `[:]` bei ❶ nach `chocolateCake` kopiert.

Hier siehst du die Ausgabe:

```
>>> print(cake)
['Eggs', 'Butter', 'Sugar', 'Milk', 'Flour']
>>> print(chocolateCake)
['Eggs', 'Butter', 'Sugar', 'Milk', 'Flour', 'Chocolate']
```

Die Werte beider Listen unterscheiden sich nun: Der Wert `"Chocolate"` ist nur in `chocolateCake` enthalten.

Elemente und if-Anweisungen

Wenn du feststellen möchtest, ob ein Wert in einer Liste enthalten ist, kannst du den `in`-Operator verwenden. Dieser Operator wird zwischen einem Wert und der zu prüfenden Liste eingefügt. Ist der Wert in der Liste vorhanden, dann wird der Ausdruck als `True`; andernfalls als `False` ausgewertet.

Mit dem folgenden Beispiel überprüfen wir, ob der Wert `"Eggs"` Bestandteil der Liste `cake` ist:

```
>>> cake = ["Eggs", "Butter", "Sugar", "Milk", "Flour"]
>>> print("Eggs" in cake)
```

Die Ausgabe ist `True`, weil `"Eggs"` in dieser Liste vorhanden ist.

Natürlich kannst du den Operator `in` auch als Bestandteil der Bedingung einer `if`-Anweisung verwenden. Der folgende Code erweitert unser Beispiel mit einer `if`-Anweisung, statt den booleschen Wert auszugeben. Er überprüft, ob "Ham" auf der Liste `cake` vorhanden ist, und gibt je nachdem, ob dies so ist oder nicht, unterschiedliche Meldungen aus:

```
>>> cake = ["Eggs", "Butter", "Sugar", "Milk", "Flour"]
>>> if "Ham" in cake:
>>>     print("That cake sounds disgusting.")
>>> else:
>>>     print("Good. Ham in a cake is a terrible mistake.")
```

Um den gegenteiligen Effekt zu erzielen, kombinierst du `in` mit dem `not`-Operator. Statt `True` wird dann `False` zurückgegeben, wenn ein Element auf der Liste enthalten ist, und umgekehrt. Das sieht dann wie folgt aus (beachte, dass die Anweisungskörper der `if`- und der `else`-Anweisung ebenfalls getauscht wurden):

```
>>> cake = ["Eggs", "Butter", "Sugar", "Milk", "Flour"]
>>> if "Ham" not in cake:
>>>     print("Good. Ham in a cake is a terrible mistake.")
>>> else:
>>>     print("That cake sounds disgusting")
```

Du kannst beide Verfahren in deinen Programmen einsetzen. Entscheide dich einfach für dasjenige, das dir sinnvoller erscheint.

Mission 52: Schwert mit Nachtsichtfunktion

Möchtest du Höhlen in Minecraft erforschen und hast du schon wieder nicht genügend Fackeln dabei? Mir passiert das ständig. Ich vergesse, Fackeln mitzunehmen, und irgendwann bin ich schon zu weit in der Höhle, um noch zurückzugehen. Also taste ich mich im Dunkeln vorwärts, ohne zu wissen, ob ich überhaupt irgendetwas Nützliches finde. Aber dank meiner Python-Kenntnisse kann ich ein Programm schreiben, das mir dabei hilft, Diamanten mit meinem Schwert zu finden.

Wir wollen ein einfaches Programm entwickeln, das mithilfe der Funktion `pollBlockHits()` überprüft, ob Blöcke, die du getroffen hast, aus Diamanterz sind. Das ist praktisch, um Höhlen ohne Licht zu erforschen oder das beliebte Spiel »Suche Diamanterz« auch im Dunkeln zu spielen. Listing 9.6 zeigt den Code. Kopiere ihn in eine neue Datei und speichere ihn als `nightVisionSword.py` im Ordner `lists`.

nightVision
Sword.py

```

from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

blocks = []

while True:
    hits = mc.events.pollBlockHits()
    if len(hits) > 0:
        hit = hits[0]
        hitX, hitY, hitZ = hit.pos.x, hit.pos.y, hit.pos.z ❶
        block = mc.getBlock(hitX, hitY, hitZ)
        blocks.append(block)

        # Hier die if-Anweisung hinzufügen ❷

    time.sleep(0.2)

```

Listing 9.6: Erfolgreich Diamanterz im Dunkeln finden

Beachte die Verwendung von `hit.pos.x`, `hit.pos.y` und `hit.pos.z` ❶. Jeder Treffer speichert die Koordinaten des angeklickten Blocks als Tupel. Du kannst dann per Punktnotation darauf zugreifen. In diesem Beispiel wird der Variablenname `hit` zur Benennung der Liste verwendet, die die einzelnen Blocktreffer enthält. Deswegen kann ich über `hit.pos.x`, `hit.pos.y` und `hit.pos.z` auf die Koordinaten zugreifen. (Das Programm untersucht immer nur das erste Hit-Event über `hits[0]`. Da wir aber eine sehr kleine Wartezeit (0,2 sec) eingebaut haben, ist das hier kein Problem.)

Der Code ist beinahe vollständig; was noch fehlt, ist eine Überprüfung, ob du Diamant gefunden hast. Füge eine `if`-Anweisung bei ❷ hinzu, um festzustellen, ob sich Diamanterz (Block-ID 56) auf der Liste `blocks` befindet, und poste, falls dies so ist, die Meldung "You found some diamond ore!" im Chat. In der `if`-Anweisung musst du auch eine `break`-Anweisung ergänzen, damit die Schleifenausführung abgebrochen wird, wenn du Erz gefunden hast.

Bild 9.7 zeigt das Programm bei der Ausführung.

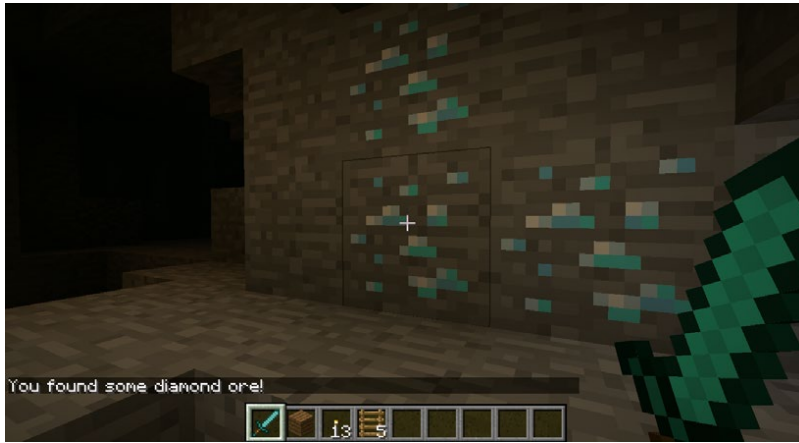


Bild 9.7: Dunkel? Egal, ich habe Diamanterz gefunden. Super!

Wenn du weniger vergesslich bist als ich und daran gedacht hast, Fackeln in die Höhle mitzunehmen, kannst du diesen Code trotzdem verwenden – als Spiel. Erstelle einen unterirdischen Raum und verstecke einen einzelnen Diamanterzblock in der Wand. Führe dann das Programm aus. Wie lange brauchst du, um den Erzblock in der Dunkelheit zu finden? Und nicht vergessen: Du musst ein Schwert verwenden und mit der rechten Maustaste klicken! Nur so kann die Funktion `pollBlockHits()` feststellen, welche Blöcke du getroffen hast.

Bonus: Die diamantene Aufgabe

Aus dem Programm `nightVisionSword.py` ein echtes Minispiel machen? Keine schlechte Idee. Kannst du automatisch einen Raum mit einem einzelnen, zufällig platzierten Diamantblock bauen, den Spieler in diesen Raum stellen und dann messen, wie lange er braucht, um den Block im Dunkeln zu finden?

Wörterbücher

Wörterbücher sind ebenfalls Listen, die aber einen anderen Ansatz verwenden. Statt eines Index werden von Wörterbüchern bestimmte Schlüssel zur Bezeichnung von Elementen verwendet, die vom Programmierer definiert wurden.

Das folgende Wörterbuch `raceTimes` etwa speichert die Namen der Teilnehmer eines Rennens und die jeweiligen Rennergebnisse:

```
raceTimes = {'Katy': 26,  
             'Alex': 30,  
             'Richard': 19}
```

Der Schlüssel kennzeichnet jeden Wert im Wörterbuch eindeutig. In unserem Beispiel ist der Schlüssel der Name der Person. So ist beispielsweise dem Schlüssel `'Katy'` der Wert 26 zugeordnet.

Wie Listen sind auch Wörterbücher veränderlich, d. h., ihre Inhalte können bearbeitet werden.

Wörterbuch definieren

Zum Definieren eines Wörterbuchs setzt du eine Anzahl Schlüssel-Wert-Paare in geschweifte Klammern. So kannst du etwa mit einem Wörterbuch eine Person beschreiben. Mit Schlüsseln wie `'name'` und `'favoriteAnimal'` speicherst du Informationen zur betreffenden Person:

```
person = {'name': 'David',  
          'age': 42,  
          'favoriteAnimal': 'Snake',  
          'favoritePlace': 'Inside a cardboard box'}
```

In diesem Beispiel ist jeder Schlüssel ein String. Die Verknüpfung von Schlüssel und Wert erfolgt mit einem Doppelpunkt. `'age'` beispielsweise ist ein Schlüssel und 42 der zugehörige Wert. Die einzelnen Elemente in einem Wörterbuch werden dann durch Kommata getrennt.

Du hast vielleicht schon bemerkt, dass der Programmierer durch Verwendung von Wörterbüchern einfacher versteht, was die einzelnen Elemente in der Liste darstellen. So kann man leicht nachvollziehen, dass der Schlüssel `'name'` einen Namen speichert – und keine Zahl oder irgend eine andere Information.

Du kannst auch Integer oder Floats als Schlüssel in Wörterbüchern verwenden. Die Verwendung von Floats oder Integer in Wörterbüchern ist vor allem dann sehr sinnvoll, wenn die Schlüssel, denen du Werte zuordnen möchtest, keine strikte Reihenfolge haben.

Das folgende Beispiel erstellt ein Wörterbuch mit einem Zugfahrplan. Die (als Float gespeicherte) Ankunftszeit des Zuges dient uns als Schlüssel, das Reiseziel als Wert:

```
trainTimes = {1.00: 'Castle Town',
              2.30: 'Sheep Farm',
              3.15: 'Lake City',
              3.45: 'Castle Town',
              3.55: 'Storage Land'
              }
```

Da in Wörterbüchern zwei Datenelemente als Paar gespeichert werden, eignen sie sich hervorragend für dieses Szenario. Hätte ich eine Liste von Zielbahnhöfen anstelle eines Wörterbuchs verwendet, dann hätte ich die Zeiten diesen Stationen nicht zuordnen können. Stattdessen hätte ich die Indexpositionen der Liste verwenden müssen.

Zugriff auf Elemente in Wörterbüchern

Um den Wert eines Elements in einem Wörterbuch abzurufen, verwendest du eckige Klammern und einen Schlüssel anstelle eines Index. Dieser Schlüssel ist in der Regel ein String oder ein Integer. Wenn du ein Wörterbuch erstellst, das Strings als Schlüssel verwendet, darfst du nicht vergessen, diese in Anführungszeichen zu setzen.

Möchtest du beispielsweise den Wert des Schlüssels 'name' im Wörterbuch `person` abrufen, das wir weiter oben erstellt haben, dann müsstest du folgende Syntax verwenden:

```
person = {'name': 'David',
          'age': 42,
          'favoriteAnimal': 'Snake',
          'favoritePlace': 'Inside a cardboard box'}
```

```
agentName = person['name']
```

Die Variable `agentName` enthält dann den Wert `'David'`, weil sie auf den Wert des Schlüssels `'name'` zugreift. Ebenso müsstest du, wenn du das Alter der Person ermitteln möchtest, den Schlüssel `'age'` verwenden:

```
agentAge = person['age']
```

Hierdurch würde der Wert 42 der Variablen `agentAge` zugeordnet.

Im `trainTimes`-Beispielcode kannst du die Werte im Wörterbuch (also die Reiseziele) über ihre Schlüsselwerte (die Ankunftszeiten) abrufen, bei denen es sich um Floats handelt:

```
trainTimes = {1.00: 'Castle Town',  
              2.30: 'Sheep Farm',  
              3.15: 'Lake City',  
              3.45: 'Castle Town',  
              3.55: 'Storage Land'  
              }
```

```
myTrain = trainTimes[3.15]
```

Wenn du auf den Schlüssel 3.15 im Wörterbuch `trainTimes` zugreifst, wird die Variable `myTrain` auf `'Lake City'` gesetzt.

Mission 53: Sightseeing

Wenn du Wörterbücher verwendest, kannst du beliebige Datentypen als Wert speichern – auch Listen und Tupel. Es ist beispielsweise möglich, ein Tupel mit Werten für `x`, `y` und `z` zu speichern. Hier ist ein Beispiel für einen Code, der so etwas tut:

```
places = {'Living room': (76, 1, -61), 'Bedroom': (61, 9, -61)}
```

Das Wörterbuch `places` speichert zwei Elemente. Der Wörterbuchschlüssel ist der Name eines Standorts in meinem Minecraft-Spiel (z. B. das Schlaf- oder das Wohnzimmer); der Wert ist ein Tupel mit den Koordinaten. Mit dem folgenden Code könnte ich dann die Koordinaten meines Wohnzimmers abrufen:

```
location = places['Living room']  
x, y, z = location[0], location[1], location[2]
```

Deine Aufgabe besteht darin, ein Programm zu schreiben, das mit einem Wörterbuch die Positionen der verschiedenen Standorte in deinem Minecraft-Spiel speichert, sodass du dich unter Angabe des betreffenden Namens dorthin teleportieren kannst. Du kannst beliebig viele Positionen in deinem Wörterbuch speichern. Um dich zu einem Standort zu teleportieren, musst du auf das Tupel mit den Koordinaten zugreifen, die im Wörterbuch gespeichert sind, und dann die Variablen *x*, *y* und *z* auf die Tupelwerte setzen. Die Kommentare im Code zeigen dir, wo das gemacht werden muss.

Kopiere Listing 9.7 in den IDLE-Editor und speichere den Code als *sightseeingGuide.py* im Ordner *lists*.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

# Positionen zum Wörterbuch hinzufügen
places = {}

choice = ""
while choice != "exit":
    choice = input("Enter a location ('exit' to close): ") ❶
    if choice in places: ❷
        # Wert des Wörterbuchelements unter Angabe des Schlüssels ↵
        (choice) speichern
        location =
        # Den Variablen x, y und z die im Tupel gespeicherten Werte ↵
        zuweisen
        x, y, z =
        mc.player.setTilePos(x, y, z)
```

*sightseeing
Guide.py*

Listing 9.7: Code, mit dem du dich zu deinen Sehenswürdigkeiten teleportieren kannst

Der Code enthält eine Anweisung, durch die du aufgefordert wirst, den Namen der Position anzugeben, die du aufsuchen möchtest. Die Eingabe wird dann in der Variablen *choice* gespeichert ❶. Danach überprüft das Programm mit einer *if*-Anweisung, ob der Wert von *choice* im Wörterbuch vorkommt ❷. In der letzten Zeile wird der Spieler dann unter Verwendung der Variablen *x*, *y* und *z* an die im Wörterbuch gespeicherte Position teleportiert.

Nach dem Start gibst du den Namen der gewünschten Position ein. Bild 9.8 zeigt meine Version des Programms zum Teleportieren an verschiedene Orte.



Bild 9.8: Ich habe mich in mein Wohnzimmer (oben) und dann in mein Schlafzimmer (unten) teleportiert.

Elemente im Wörterbuch ändern oder hinzufügen

Den Wert eines Elements im Wörterbuch zu ändern ist relativ unkompliziert. Du gibst einen Schlüssel in eckigen Klammern an, um auf das Element zuzugreifen, und legst es wie eine normale Variable (also mit einem Gleichheitszeichen) fest. Zudem kannst du neue Elemente auf gleiche Weise hinzufügen.

Wir wollen den Wert des Elements `age` im Wörterbuch `person` von 42 auf 43 setzen:

```
person['age'] = 43
```

Als Nächstes fügen wir ein neues Element namens `location` mit dem Wert `'USS Discovery'` hinzu:

```
person['location'] = 'USS Discovery'
```

Wenn du diesen Code ausführst, erhält das Wörterbuch einen neuen Schlüssel namens `location` mit dem Wert `'USS Discovery'`.

Elemente aus Wörterbüchern löschen

Manchmal wirst du auch Elemente aus einem Wörterbuch löschen müssen. Wie bei Listen tust du dies mit dem Schlüsselwort `del`. Um beispielsweise das Element `favoriteAnimal` aus dem Wörterbuch `person` zu löschen, gehst du wie folgt vor:

```
del person['favoriteAnimal']
```

Wie du siehst, funktioniert das tatsächlich wie bei Listen.

Mission 54: Getroffene Blöcke zählen

In Mission 50 (Seite 233) hast du ein Programm geschrieben, das die Anzahl der Blocktreffer mit dem Schwert innerhalb von 60 Sekunden zählt. Das ist zwar schon nicht schlecht, aber noch besser wäre es, wenn du die Resultate aller Mitspieler speichern könntest.

Um eine Ergebnistafel zum Spiel hinzuzufügen, kannst du ein Wörterbuch verwenden. In diesem Wörterbuch werden der Name des Spielers und sein Resultat gespeichert. Du kannst dann die Ergebnisse aller Spieler anzeigen.

Öffne zunächst `swordHits.py` und speichere es als `swordHitsScore.py` im Ordner `lists`. Ändere den Code so ab, dass er mit Listing 9.8 identisch ist. Ich habe dort ein paar Änderungen vorgenommen, damit das Programm wiederholt ausgeführt wird, den Spieler zur Eingabe seines Namens auffordert und dann alle Ergebnisse ausgibt. Außerdem sind die Lösungen für die fehlenden Codeabschnitte in `swordHits.py` enthalten. Die älteren Abschnitte sind abgeblendet. (Nicht vergessen: Alles innerhalb der Schleife muss eingegeben werden.)

swordHits
Score.py

```
# Verbindung mit Minecraft herstellen
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

name = ""
scoreboard = {}

while True:
    # Namen des Spielers abrufen
    name = input("What is your name? ")

    # Schleife abbrechen, wenn Name 'exit' ist
    if name == "exit":
        break
    mc.postToChat("Go!")

    # 60 Sekunden warten
    time.sleep(60)

    # Liste mit den Blocktreffern abrufen
    blockHits = mc.events.pollBlockHits()

    # Länge der Blocktrefferliste im Chat anzeigen
    blockHitsLength = len(blockHits)
    mc.postToChat("Your score is " + str(blockHitsLength))

    # Name des Spielers zur Ergebnistafel hinzufügen ❶

    # Ergebnistafel anzeigen
    print(scoreboard)
```

Listing 9.8: Mit dem richtigen Code wird eine Ergebnistafel zum Blocktrefferspiel hinzugefügt.

Um das Programm zu vervollständigen, musst du die Namen und den Spielstand aller Mitspieler speichern. Hierzu fügst du ein neues Wörterbuch hinzu und verwendest dazu die Datenelemente im Code bei ❶. Das Wörterbuch heißt `scoreboard`, und der Name des Spielers wird in der Variablen `name` gespeichert.

Bild 9.9 zeigt die Ausgabe in meinem Minecraft-Spiel.



Bild 9.9: Wir haben gegeneinander gespielt und Jim hat mit 274 Treffern gewonnen.

Hinweis

Du wirst vielleicht bemerken, dass das ausgegebene Wörterbuch scoreboard gar nicht so einfach zu lesen ist. Wir werden die Anzeige in Mission 59 (Seite 266) optimieren.

Bonus: Der beste Spielstand

In der aktuellen Version wird, wenn jemand das Spiel *swordHits-Score.py* häufiger spielt (und dabei jeweils denselben Benutzernamen eingibt), nur das aktuelle Ergebnis gespeichert. Dabei könnte man mit einer *if*-Anweisung durchaus überprüfen, ob der Spieler bereits ein Ergebnis eingegeben hat, und den neuen Spielstand nur speichern, wenn er besser ist als der vorherige. Hast du eine Idee, wie das klappen könnte? Unten siehst du zur Hilfestellung den Anfang des erforderlichen Codes. Hiermit wird geprüft, ob der Name bereits im Wörterbuch scoreboard vorhanden ist:

```
if name in scoreboard:
```

Was du gelernt hast

Saubere Arbeit! In diesem Kapitel hast du Listen, Tupel und Wörterbücher kennengelernt. Wie haben gesehen, wie du mehrere Datenwerte in einer einzigen Variablen speichern kannst. Hierbei handelt es sich um eine praktische Möglichkeit, Daten in deinen Programmen zu strukturieren und zu speichern.

Du hast in den Missionen verschiedene kurzweilige Programme erstellt, die Listen, Wörterbücher und Tupel verwenden. Mit Listen und Lapislazuli- und Glasblöcken hast du eine Fortschrittsanzeige erstellt. Du weißt, dass sich die Variablen `x`, `y` und `z` mit Tupeln noch schneller festlegen lassen, und zu guter Letzt hast du mit Wörterbüchern die Koordinaten von Objekten gespeichert, die du gebaut hast, und dich dann dorthin teleportiert.

In Kapitel 10 werden wir deine Kenntnisse zu Listen ausbauen – mit `for`-Schleifen. Dann wirst du ein paar wirklich coole Programme erstellen, etwa eines zum Duplizieren von selbst gebauten Objekten.

Kapitel 10

Wie von Geisterhand – oder mit der for-Schleife

Nun ist es an der Zeit, etwas Neues kennenzulernen: for-Schleifen. for-Schleifen sind extrem praktisch, denn sie iterieren über Listen – genau solche Listen, wie du sie in Kapitel 9 kennengelernt hast. Insofern sind sie für die Verwendung von Schleifen in Verbindung mit Listen auch die optimale Lösung.

In den Missionen in diesem Kapitel wirst du mit for-Schleifen Treppen, Säulen, Pyramiden und verwitterte Mauern bauen. Mit verschachtelten for-Schleifen und Listen kannst du Bilder erstellen und neue Strukturen generieren – und das in wenigen Sekunden. Wenn es um das Bauen in Minecraft geht, sind for-Schleifen eigentlich unschlagbar!

Eine einfache for-Schleife

Eine for-Schleife wiederholt einen Codeblock für jedes Element einer Liste bis zu deren Ende. Es wird also keine Bedingung eingesetzt, wie wir sie von `while`-Schleifen oder `if`-Anweisungen her kennen.

Die in einer for-Anweisung verwendete Liste kann dabei beliebig lang sein. Die for-Schleife iteriert dann in der durch den Index vorgegebenen Reihenfolge über jedes Listenelement. Um beispielsweise alle Zutaten unserer Nudelsuppe auszugeben, würden wir folgenden Code verwenden:

```
noodleSoup = ["water", "soy sauce", "spring onions", "pepper",  
"noodles", "beef", "vegetables"]  
  
for ingredient in noodleSoup:  
    print(ingredient)
```

Mit dem for-Operator zeigen wir Python an, dass wir eine Schleife benutzen.

Auf diesen Operator folgt eine Variable namens `ingredient`, die das von der Schleife gerade bearbeitete Element repräsentiert. Der Wert ändert sich bei jeder Schleifeniteration, bis die gesamte Liste abgearbeitet ist. Beim ersten Schleifendurchlauf wird als Wert das Element an der Indexposition 0 ("water") benutzt, beim zweiten das an Indexposition 1 ("soy sauce"), beim dritten das an Position 2 ("spring onions") usw.

Der `in`-Operator und der Listenname am Ende der Anweisung sagen Python, welche Liste verwendet wird. In diesem Beispiel ist das die Liste `noodleSoup`.

Die Schleife wird für jedes Element einmal ausgeführt, d. h., sie endet, sobald die Liste vollständig abgearbeitet ist. Die Ausgabe des Programms sieht wie folgt aus:

```
water  
soy sauce  
spring onions  
pepper  
noodles  
beef  
vegetables
```

Es wird also jedes einzelne Listenelement ausgegeben! Und jetzt wollen wir einmal sehen, was man mit for-Schleifen in Minecraft Sinnvolles anstellen kann.

Mission 55: Der Zauberstab

Jedes Werkzeug in Minecraft hat seinen ganz bestimmten Zweck. Mit Schaufeln gräbt man in der Erde, mit der Spitzhacke zerbricht man Steinblöcke und mit dem Schwert erlegt man Schufte. Normalerweise lässt sich das Verhalten von Werkzeugen nicht ändern – man muss einfach akzeptieren, dass das Schwert nur zur Bekämpfung von Feinden gut ist. Mit Python aber kannst du die Wirkungsweise deiner Werkzeuge beein-

flussen. Wir werden dazu ein Programm schreiben, das dein Schwert in einen Zauberstab verwandelt.

In Kapitel 9 hast du die `pollBlockHits()`-Funktion kennengelernt. Diese Funktion gibt eine Liste mit den Koordinaten von Blöcken zurück, die du mit dem Schwert getroffen hast. Mit einer `for`-Schleife kannst du nun auf alle Koordinatensätze in dieser Gruppe zugreifen. Lass uns doch einfach mal alle Blöcke, die du in den letzten 60 Sekunden getroffen hast, in Melonen verwandeln. Wie das aussehen soll, zeigt Bild 10.1.



Bild 10.1: Abrakadabra! Alle getroffenen Blöcke sind jetzt Melonen.

Listing 10.1 enthält den Anfang des Programms. Speichere es als *magicWand.py* in einem neuen Ordner namens *forLoops*.

```

from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

time.sleep(60)

hits = mc.events.pollBlockHits() ❶
block = 103

for ❷
    x, y, z = hit.pos.x, hit.pos.y, hit.pos.z ❸
    # Blöcke an den Koordinaten in Melonen umwandeln ❹

```

magicWand
.py

Listing 10.1: Anfang des Zauberstabprogramms

Um die Liste der Blocktreffer abzurufen, rufen wir `pollBlockHits()` auf und speichern das Ergebnis in der Variablen `hits` ❶.

Das Listing enthält eine Codezeile, mit der du die Position beliebiger getroffener Blöcke abrufst und die Koordinaten in den Variablen `x`, `y` und `z` speicherst ❷. Hier wird ein Tupel (vorgestellt in »Tupel« auf Seite 228) verwendet, um die drei Variablen mit nur einer Zeile mit Werten zu versehen.

In der obigen Version funktioniert diese Zeile noch nicht, weil die Variable `hit` nicht existiert. Erstelle eine `for`-Schleife bei ❸ und nenne die Variable dieser Schleife `hit`. Die `for`-Schleife sollte dann über die Liste `hits` iterieren. Der Code für den ersten Teil der `for`-Schleife sollte etwa so aussehen:

```
for hit in hits:
```

Vergiss nicht, die Codezeile, mit der die Werte von `x`, `y` und `z` abgerufen werden, innerhalb der `for`-Schleife bei ❹ einzuziehen. In der letzten Zeile der `for`-Schleife fügst du dann die Funktion `setBlock()` hinzu, um einen Melonenblock an die Position zu setzen, die durch `x`, `y` und `z` angegeben wird ❺.

Führt der Benutzer das fertige Programm aus, dann hat er 60 Sekunden, um durch die Gegend zu laufen und mit dem Schwert so viele Blöcke wie möglich per Rechtsklick zu treffen. Nach diesen 60 Sekunden werden dann alle mit dem Schwert getroffenen Blöcke zu Melonen.

Bonus: Du bist ein Zauberer

Ändere das Programm *magicWand.py* so ab, dass es den Spieler teleportiert: Der erste Treffer bestimmt die Position, der zweite bringt den Spieler dorthin.

Die Funktion `range()`

Die Funktion `range()` erstellt eine Liste mit Integeren. Mit ihr kannst du sehr schnell eine Liste mit Zahlen für deine `for`-Schleifen erstellen. Sehen wir uns das einmal an. Wir übergeben zwei Argumente:

```
aRange = range(0, 5)
```

Das geht wesentlich zügiger, als jedes Element einzeln in die Liste zu schreiben, was so aussähe:

```
aRange = [0, 1, 2, 3, 4]
```

Beachte, dass das zweite Argument für `range()` 5 lautet, das letzte Element in der Liste aber 4 ist. Das liegt daran, dass die Funktion nur Werte erstellt, die kleiner, aber nicht gleich dem zweiten Argument sind.

Zum Erstellen einer Schleife, die mithilfe der `range()`-Funktion die Zahlen 1 bis 15 ausgibt, würdest du folgenden Code benutzen:

```
for item in range(1, 16):  
    print(item)
```

Den doppelten Wert jedes Listenelements gibst du so aus:

```
for item in range(1, 16):  
    print(stringToDb1 * 2)
```

Das Gleiche kannst du auch mit der `while`-Schleife aus Kapitel 7 machen. Der folgende Code verwendet für die Ausgabe der Zahlen 1 bis 15 eine solche `while`-Schleife anstelle einer `for`-Schleife:

```
count = 1  
while count < 16:  
    print(count)  
    count += 1
```

Die `for`-Schleife ist eindeutig intuitiver und auch leichter zu lesen. In großen und komplexen Programmen ist eine `for`-Schleife im Vergleich zur `while`-Schleife mit `count` oft die bessere Wahl.

Mission 56: Die Zaubertreppe

Einer der größten Vorteile der Verwendung von Minecraft in Kombination mit Python besteht darin, dass du mit nur wenigen Codezeilen schnell Dinge bauen kannst. Statt Stunden über Stunden mit dem Bauen von Mauern und Wänden zu verbringen, führst du einfach nur einen kurzen Code aus und bist schon fertig. Hinzu kommt: Du kannst den Code beliebig oft wiederverwenden, was zusätzlich Zeit und Aufwand einspart.

Das Bauen von Treppen in Minecraft ist eine Sache, die oft extrem zeitaufwendig ist. Aber auch das lässt sich mit nur wenigen Zeilen Python-Code schnell erledigen. In dieser Mission benutzt du für den Bau einer Treppe in der Spielwelt eine for-Schleife.

Listing 10.2 erstellt eine Treppe in Minecraft mithilfe einer while-Schleife. Speichere das Programm als *stairs.py* im Ordner *forLoops*.

stairs.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

pos = mc.player.getTilePos()
x, y, z = pos.x, pos.y, pos.z

stairBlock = 53

step = 0
while step < 10:
    mc.setblock(x + step, y + step, z, stairBlock)
    step += 1
```

Listing 10.2: Programm zum Bau einer Treppe mit einer while-Schleife

Du kannst natürlich wie hier gezeigt eine while-Schleife verwenden, doch ist eine for-Schleife für diese Zwecke wesentlich geeigneter. Anders als die while-Schleife benötigt eine for-Schleife keine count- oder step-Variable. Stattdessen legst du mit der range()-Funktion die Wiederholungshäufigkeit der Schleife fest.

Um das Programm zu vervollständigen, änderst du den Code so ab, dass er anstelle der while- eine for-Schleife verwendet.

Was passiert, wenn du das Programm ausführst, siehst du in Bild 10.2.

Bonus: Und wie geht es nach unten?

In der aktuellen Version baut das Programm *stairs.py* Treppen nur in eine Richtung. Überarbeite das Programm so, dass auch andere Richtungen unterstützt werden. Hinweis: Hierzu verwendest du das optionale Blockzustandsargument in der Funktion `setBlock()` und führst entsprechende Additionen oder Subtraktionen an den Variablen `x` oder `z` durch.



Bild 10.2: Wohin mag diese Zaubertreppe führen?

Lustige Sachen mit `range()` machen

Du hast die Funktion `range()` nun kennengelernt und weißt, was passiert, wenn du ihr zwei Argumente übergibst. Aber angenommen, du nennst nur ein einzelnes Argument. Gib einmal den folgenden Code in die IDLE-Shell ein und sieh dir an, was passiert:

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

Wenn du der Funktion `range()` nur ein Argument übergibst, beginnt sie bei 0 zu zählen und speichert jeden Wert bis einschließlich desjenigen, der um 1 kleiner ist als das übergebene Argument. Anders formuliert: Dies ist so, als ob du 0 als erstes und 5 als zweites Argument übergeben hättest. In diesem Beispiel zeigt die Funktion `list()` die von der Funktion `range()` erstellte Liste `values` an (andernfalls könntest du sie nämlich gar nicht sehen). Der Wert von `list(aRange)` ist also eine Liste mit fünf Zahlen, die bei 0 beginnend aufwärts gezählt werden: `[0, 1, 2, 3, 4]`. Soll der benötigte Bereich bei 0 beginnen, dann lässt er sich auf diese Weise extrem schnell erstellen.

Wie wir gesehen haben, beginnt, wenn du zwei Argumente an `range()` übergibst, die Liste beim ersten übergebenen Argument und endet vor dem zweiten:

```
>>> aRange = range(2, 5)
>>> list(aRange)
[2, 3, 4]
```

Dieses Beispiel erstellt einen Bereich, der der Liste [2, 3, 4] entspricht.

Übergibst du drei Argumente an `range()`, dann definiert das dritte Argument die *Schrittweite* zwischen den einzelnen Listenelementen. Normalerweise ist jeder Wert in einer mit `range()` erstellten Liste um 1 größer als der vorherige. Änderst du dagegen die Schrittweite, dann ändert sich auch die Differenz zwischen den Werten. Bei einer Schrittweite von 2 etwa wäre der nächste Wert in der Liste um 2 größer als sein Vorgänger. Bei einer Schrittweite von 3 betrüge der Abstand zum Vorgänger jeweils 3 usw.

So fügt beispielsweise die folgende Liste 2 zu einem Wert hinzu, um die Größe des nächsten Werts zu ermitteln:

```
>>> aRange = range(3, 10, 2)
>>> list(aRange)
[3, 5, 7, 9]
```

Beachte, dass jedes Element um den Wert 2 größer ist als das vorherige: 5 ist $3 + 2$, 7 ist $5 + 2$ und 9 ist $7 + 2$.

Du kannst für `range()` sogar eine negative Schrittweite angeben:

```
>>> newRange = range(100, 0, -2)
>>> list(newRange)
[100, 98, 96, 94, 92, 90, 88, 86, 84, 82, 80, 78, 76, 74, 72, 70, 68,
66, 64, 62, 60, 58, 56, 54, 52, 50, 48, 46, 44, 42, 40, 38, 36, 34,
32, 30, 28, 26, 24, 22, 20, 18, 16, 14, 12, 10, 8, 6, 4, 2]
```

Beachte, dass die Werte in der Liste aufgrund der negativen Schrittweite um den Wert 2 abnehmen.

Weitere Listenfunktionen

Wo wir gerade mit Listen arbeiten, wollen wir uns noch ein paar Funktionen genauer ansehen, die speziell für die Interaktion mit Listen vorgesehen sind.

Die Funktion `reversed()` nimmt ein Argument – die zu verwendende Liste – entgegen und gibt diese dann in umgekehrter Reihenfolge zurück.

Das letzte Element wird also zum ersten, das vorletzte zum zweiten usw. Wir wollen eine Liste, die wir weiter oben bereits verwendet haben, einmal umdrehen:

```
>>> backwardsList = reversed(aRange)
>>> list(backwardsList)
[9, 7, 5, 3]
```

Die Reihenfolge der Elemente in dieser Liste wurde umgekehrt – genau das, was wir wollten. Derartige Manipulationen können recht praktisch sein, wenn man for-Schleifen schreibt.

Das folgende Beispiel erzeugt mithilfe der Funktion `range()` eine Liste der Zahlen von 1 bis 100. Diese Liste wird dann invertiert und mithilfe einer for-Schleife ausgegeben – im Endeffekt ein Countdown von 100 bis 1:

```
countDown = range(1, 101)
countDown = reversed(countDown)
for item in countDown:
    print(item)
```

Starte den Code, um die Ausgabe anzuzeigen!

```
100
99
98
97
96
-- usw. --
3
2
1
```

Du kannst die Liste auch invertieren, indem du die for-Schleife deklarierst, ohne eine Variable anzugeben, die die Liste aufnimmt:

```
for item in reversed(range(1, 101)):
    print(item)
```

Dieses Programm braucht weniger Codezeilen, um den gleichen Effekt zu erzielen. Nutze diesen Trick, um Zeit zu sparen und dich auf deine Minecraft-Konstruktionen zu konzentrieren.

Mission 57: Säulen

Wäre es nicht cool, in Minecraft einen Palast zu bauen? Und weil Paläste ja imposant wirken sollen, hätte der unsrige lange Gänge mit riesigen und beeindruckenden Säulen. Selbstverständlich wollen wir diese Säulen nicht per Hand bauen, sondern wir entscheiden uns für eine Lösung, die eine Schleife verwendet.

Wir werden eine Funktion erstellen, mit der eine Säule gebaut wird, und diese Funktion immer dann aufrufen, wenn wir eine Säule brauchen. Listing 10.3 enthält die Funktion zum Errichten einer Säule. Kopiere sie in eine neue Datei namens *pillars.py* und speichere sie im Ordner *forLoops*.

pillars.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

def setPillar(x, y, z, height):
    """ Errichtet eine Säule. Angabe von Position und Höhe durch
    Argumente """
    stairBlock = 156
    block = 155

    # Spitze der Säule
    mc.setBlocks(x - 1, y + height, z - 1, x + 1, y + height,
                 z + 1, block, 1)
    mc.setBlock(x - 1, y + height - 1, z, stairBlock, 12)
    mc.setBlock(x + 1, y + height - 1, z, stairBlock, 13)
    mc.setBlock(x, y + height - 1, z + 1, stairBlock, 15)
    mc.setBlock(x, y + height - 1, z - 1, stairBlock, 14)

    # Fuß der Säule
    mc.setBlocks(x - 1, y, z - 1, x + 1, y, z + 1, block, 1)
    mc.setBlock(x - 1, y + 1, z, stairBlock, 0)
    mc.setBlock(x + 1, y + 1, z, stairBlock, 1)
    mc.setBlock(x, y + 1, z + 1, stairBlock, 3)
    mc.setBlock(x, y + 1, z - 1, stairBlock, 2)

    # Mittelteil der Säule
    mc.setBlocks(x, y, z, x, y + height, z, block, 2)

pos = mc.player.getTilePos()
x, y, z = pos.x + 2, pos.y, pos.z

# for-Schleife hier hinzufügen ❶
# Funktion hier aufrufen ❷
```

Listing 10.3: Funktion zum Bauen einer Säul

Mit der Funktion `setPillar()` wird eine Säule gebaut. Sie nimmt vier Argumente entgegen: die Koordinaten `x`, `y` und `z` sowie die Säulenhöhe.

Um das Programm zu vervollständigen, fügst du eine `for`-Schleife ❶ hinzu, die die Funktion `setPillar()` ❷ aufruft. Wir wollen einmal eine Reihe mit 20 Säulen erstellen, die jeweils fünf Blöcke weit auseinander stehen. Hierzu verwenden wir eine `range()`-Aktion mit drei Argumenten, um festzulegen, wie die Säulen errichtet werden und wie weit sie auseinander stehen sollen. Indem du die Werte, die in der Variablen der `for`-Schleife gespeichert sind, zu den Variablen `x` oder `z` im Funktionsaufruf von `setPillar()` addierst, kannst du die Säulen in identischen Abständen aufstellen.

Bild 10.3 zeigt ein paar Säulen.



Bild 10.3: Eine wunderschöne Säulenreihe

Mission 58: Pyramide

Bauen wir noch mehr tolle Sachen mit `for`-Schleifen, zum Beispiel eine Pyramide. Eine Pyramide besteht aus zahlreichen Ebenen. Die unterste Ebene ist am breitesten, die oberste Ebene – also die Spitze – am schmalsten. Jede Ebene ist ein Rechteck aus Blöcken. Unsere Pyramide soll auf jeder Ebene jeweils zwei Blöcke schmäler sein als auf der darunter liegenden. Wenn beispielsweise die unterste Ebene sieben Blöcke breit ist, muss die nächste fünf Blöcke, die übernächste drei Blöcke und die oberste dann nur noch einen Block breit sein.

Listing 10.4 erstellt eine Pyramide. Kopiere es in eine neue Datei namens *pyramid.py* und speichere sie im Ordner *forLoops*.

pyramid.py

```

from mcpi.minecraft import Minecraft
mc = Minecraft.create()

block = 24 # sandstone
height = 10 ❶
levels = range(height) ❷

pos = mc.player.getTilePos()
x, y, z = pos.x + height, pos.y, pos.z ❸

for level in levels: ❹
    mc.setBlocks(x - level, y, z - level, x + level, y, z + 1
                 level, block) ❺
    y += 1

```

Listing 10.4: Programm für den Pyramidenbau (mit einem Bug)

Zwar baut uns Listing 10.4 eine Pyramide, doch enthält es einen kleinen Bug, den du korrigieren musst. Wir speichern die Höhe der Pyramide in der Variablen `height` ❶. Den Wert von `height` kannst du beliebig festlegen. Mithilfe der Funktion `range()` erstellt die Variable `levels` eine Liste, die für jede Ebene der Pyramide ein Element enthält ❷. Die Variable `height` wird zur x-Koordinate des Spielers hinzuaddiert, wenn wir die Variablen `x`, `y` und `z` festlegen ❸. Würden wir dies versäumen, dann wäre der Spieler nach dem Bau der Pyramide in ihrem Innern gefangen.

Die `for`-Schleife iteriert für jedes `level`-Element in der Liste `levels` ❹. Die Codezeile, mit der die einzelnen Pyramidenebenen erstellt werden, ermittelt anhand der Variablen `level` die Breite der einzelnen Blockrechtecke ❺. Breite und Länge der einzelnen Pyramidenebenen entsprechen stets dem Doppelten der Variablen `level`.

Aber da ist ja noch dieser Bug. Hast du ihn schon bemerkt? Falls nicht, führe das Programm einmal aus. Dann wirst du feststellen: Die Pyramide steht auf dem Kopf!

Um dieses Problem zu beheben und dafür zu sorgen, dass die Pyramide richtigerum steht, musst du mithilfe der Funktion `reversed()` die Variable `levels` bearbeiten, um eine Liste zu erstellen, deren Elemente immer kleiner werden. Oder du bist verschlagen und rufst die Funktion `range()` mit einem negativen Wert auf.

Bild 10.4 zeigt die fertige Pyramide.



Bild 10.4: Eine Pyramide, wie sie im Buche steht

Schleife über ein Wörterbuch erstellen

Du kannst eine for-Schleife auch bei Wörterbüchern verwenden. Die Syntax ist dabei identisch mit der zur Bearbeitung einer Liste mit einer for-Schleife; allerdings iteriert die Schleife nur über die Schlüssel des Wörterbuchs.

Der folgende Code etwa gibt die Variable der for-Schleife bei jeder Iteration aus. Die Ausgabe enthält in diesem Fall den Schlüssel jedes einzelnen Elements:

```
inventory = {'gems': 5, 'potions': 2, 'boxes': 1}
```

```
for key in inventory:  
    print(key)
```

Wenn der Code ausgeführt wird, wird Folgendes ausgegeben:

```
gems  
potions  
boxes
```

Um den Wert auszugeben, der mit dem jeweiligen Element im Wörterbuch verknüpft ist, verwendest du die Syntax `woerterbuch[schlues-
sel]`. Nachfolgend siehst du, wie der Code geändert werden muss, um den Wert jedes Elements sowie den Schlüssel auszugeben:

```
inventory = {'gems': 5, 'potions': 2, 'boxes': 1}

for key in inventory:
    print(key + " " + str(inventory[key]))
```

Dieses Beispiel gibt Folgendes aus:

```
gems 5
potions 2
boxes 1
```

Vielleicht ist dir aufgefallen, dass die Ausgabe wesentlich einfacher zu lesen ist als das Wörterbuch selbst. Wenn du eine Schleife zum Ausgeben von Wörterbuchwerten verwendest, hast du wesentlich mehr Kontrolle darüber, wie die Informationen dargestellt werden.

Mission 59: Ergebnistafel

Du erinnerst dich doch noch an das Spiel *swordHitsScore.py* aus Mission 54 (Seite 249), oder? Dieses Spiel zeichnete auf, wie viele Blöcke ein Spieler innerhalb einer Minute getroffen hat. Das Resultat wurde nebst dem Namen des Spielers in einem Wörterbuch gespeichert. Das Programm hat zwar einwandfrei funktioniert, aber das Ausgabeformat der Resultate und Namen auf der Ergebnistafel am Ende des Spiels war alles andere als gut lesbar. Ausgegeben wurde lediglich ein Wörterbuch ohne jedwede Formatierung.

Um dieses Programm zu verbessern, wirst du die Datei *swordHitsScore.py* in dieser Mission so optimieren, dass das Wörterbuch *scoreboard* in einer Form ausgegeben wird, die sich gut lesen lässt. Hierzu verwendest du eine *for*-Schleife.

Öffne das Programm *swordHitsScore.py* (es sollte sich im Ordner *lists* befinden) und speichere es als *scoreBoard.py* im Ordner *forLoops*. Suche im Programm die folgende Zeile und lösche sie:

```
print(scoreboard)
```

Ersetze sie durch eine *for*-Schleife, die den Namen des jeweiligen Spielers und sein Ergebnis ausgibt. Die Werte sind im Wörterbuch *scoreboard* gespeichert. In diesem Wörterbuch ist jeder Spielernamen ein Schlüssel und jedes Ergebnis ein Schlüsselwert.

Bild 10.5 zeigt die aktualisierte Ausgabe.



Bild 10.5: Deutlich ansprechender: die neue Programmausgabe

for-else-Schleifen

Du kannst die `else`-Anweisung auch kombiniert mit einer `for`-Schleife verwenden. Wenn du die `else`-Anweisung gemeinsam mit einer `for`-Schleife benutzt, wird sie ausgeführt, sobald die `for`-Schleife das Ende der Liste erreicht hat. Erreicht die `for`-Schleife das Ende der Liste nicht, dann wird die `else`-Anweisung auch nicht ausgeführt.

Der folgende Code gibt die Zutaten für ein Sandwich aus und enthält eine zusätzliche `else`-Anweisung:

```
sandwich = ["Bread", "Butter", "Tuna", "Lettuce", "Mayonnaise", "Bread"]

for ingredient in sandwich:
    print(ingredient)
else:
    print("This is the end of the sandwich.")
```

Wenn du den Code ausführst, wird Folgendes ausgegeben:

```
Bread
Butter
Tuna
Lettuce
Mayonnaise
Bread
This is the end of the sandwich.
```

Vielleicht glaubst du nun, dass man den Code genauso gut hätte so schreiben können:

```
for ingredient in sandwich:
    print(ingredient)
print("This is the end of the sandwich.")
```

Zugegeben: Es ist in der Tat so. Beide Codeabschnitte machen genau das Gleiche. Wozu verwenden wir also `else` mit einer `for`-Schleife? Ganz einfach: Wenn eine `break`-Anweisung mit ins Spiel kommt, ändert sich das Verhalten der `else`-Anweisung. Und das werden wir uns jetzt ansehen.

for-else-Schleife unterbrechen

Die Verwendung einer `break`-Anweisung zum Ausstieg aus einer `for`-Schleife stellt eine Möglichkeit dar, das Ausführen der `else`-Anweisung zu verhindern.

Das folgende Beispiel enthält eine `break`-Anweisung innerhalb einer `if`-Anweisung. Die Schleife wird unterbrochen, wenn das aktuelle Listenelement "Mayonnaise" ist:

```
sandwich = ["Bread", "Butter", "Tuna", "Lettuce", "Mayonnaise", "Bread"]

for ingredient in sandwich:
    if ingredient == "Mayonnaise":
        print("I don't like mayonnaise on my sandwich.")
        break
    else:
        print(ingredient)
else:
    print("This is the end of the sandwich.")
```

Kannst du die Ausgabe vorhersagen? Denke ein bisschen darüber nach, bevor du den Code ausführst, und starte ihn dann, um deine Meinung zu überprüfen.

Mission 60: Der Diamantensucher

Manchmal, wenn ich Minecraft mit Freunden spiele, erlauben sie mir nicht, Diamantblöcke mit Python-Programmen zu erzeugen. Trotzdem brauche ich Diamanten natürlich für Waffen, Werkzeuge und den Bau von Diamantschlössern. Einfach ein bisschen zu graben ist zwar einfach, aber leider findet man so nicht überall Diamanten.

Um etwas Zeit zu sparen, habe ich ein Programm geschrieben, das feststellt, ob direkt unter mir Diamanterz vorhanden ist. Das Programm ruft meine aktuelle Position ab und überprüft dann mithilfe einer for-Schleife einen Block nach dem anderen unter mir, um festzustellen, ob dieser aus Diamanterz besteht. Wird ein solcher Erzblock gefunden, dann teilt mir das Programm mit, wie tief er liegt, andernfalls informiert es mich darüber, dass kein Diamanterz vorhanden ist.

Erstelle ein neues Programm und speichere es als *diamondSurvey.py* im Ordner *forLoops*.

Mit einer for-Schleife änderst du dann bei jeder Iteration den Wert der Variablen *y* um -1 . Insgesamt sollte die Schleife 50 Wiederholungen ausführen, um die Überprüfung 50 Blöcke tief vorzunehmen. Für jede Iteration überprüfst du mit einer *if*-Anweisung, ob der Block an der betreffenden Position aus Diamanterz ist (Block-ID 56). Handelt es sich tatsächlich um einen Diamanterzblock, dann postest du eine Meldung im Chat, aus der hervorgeht, wie tief der Block sich unter dem Spieler befindet, und brichst dann die Schleife ab. Werden keine Diamanterzblöcke gefunden, dann postest du in einer *else*-Anweisung innerhalb deiner for-Schleife eine Meldung, die besagt, dass sich direkt unter dem Spieler keine Diamanterzblöcke befinden.

Bild 10.6 zeigt das laufende Programm.



Bild 10.6: Offenbar liegt vier Blöcke unter mir Diamanterz. Dann buddeln wir mal los!

Bonus: Goldrausch

Ändere das Programm *diamondSurvey.py* so ab, dass auch nach anderen Blöcken gesucht werden kann – beispielsweise Eisenerz oder Golderz.

Verschachtelte for-Schleifen und mehrdimensionale Listen

Du kannst in deinen Programmen mehrere Listen zu unterschiedlichen Zwecken miteinander kombinieren. Wenn Listen in Listen enthalten sind, spricht man von *mehrdimensionalen Listen*. Wir werden in diesem Abschnitt zweidimensionale (2-D) und dreidimensionale Listen (3-D) verwenden, um Strukturen in Minecraft zu bauen.

In zwei Dimensionen denken

Du weißt bereits, wie Listen geschrieben werden – konkret *eindimensionale* Listen. Diese Listen heißen so, weil sich an jeder Listenposition nur ein Element befindet.

Betrachten wir folgende Liste namens `oneDimensionalRainbowList`. Die Formatierung dieser Liste unterscheidet sich geringfügig von den bisher gezeigten, um so zu betonen, dass jede Listenposition genau ein Element enthält. Ansonsten entspricht die Struktur aber weitgehend den Listen, mit denen du bereits gearbeitet hast:

```
oneDimensionalRainbowList = [0,
                             1,
                             2,
                             3,
                             4,
                             5]
```

Die Liste enthält sechs Elemente, nämlich die Zahlen 0 bis 5. Jedes Element in der Liste hat nur genau einen Wert – es ist also eine eindimensionale Liste.

Listing 10.5 zeigt diese Liste in Minecraft in Form von Wollblöcken. Die zugehörige Programmdatei namens *rainbowStack1.py* ist Bestandteil der Ressourcen zu diesem Buch. Lade die Codedateien von <https://www.nostarch.com/pythonwithminecraft/> herunter oder schreibe das Listing einfach ab.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

oneDimensionalRainbowList = [0, 1, 2, 3, 4, 5] ❶

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z

for color in oneDimensionalRainbowList: ❷
    mc.setBlock(x, y, z, 35, color)
    y += 1
```

*rainbow
Stack1.py*

Listing 10.5: Blockstapel in Regenbogenfarben

Dieses Programm erstellt eine Liste mit Blockfarben ❶ und legt dann unter Verwendung einer `for`-Schleife einen Stapel mit Wollblöcken in den Farben an, die in der Liste aufgeführt sind ❷.

Wenn du das Programm ausführst, wird ein einzelner Stapel mit Wollblöcken erstellt (Bild 10.7). Beachte, dass der Stapel sechs Blöcke hoch und einen Block breit ist. Du hast die Variablen `x`, `y` und `z` schon einige Male in

diesem Buch verwendet. Diese Variablen lassen sich auch als *Dimensionen* bezeichnen. Das Programm erstellt einen Stapel aus sechs Blöcken in der y-Dimension. Wenn du die Variable x in der letzten Codezeile anstelle der Variablen y verwendest, kannst du einen Stapel mit Blöcken in der x-Dimension erstellen (Bild 10.8).



Bild 10.7: Mit dem Programm rainbowStack1.py erstellter Regenbogenstapel



Bild 10.8: Tauschst du nun die Variable y in der letzten Programmzeile gegen die Variable x aus, dann erstellt das Programm eine horizontale Blockreihe.

Da die Liste eindimensional ist, kannst du immer nur den Wert einer Variablen gleichzeitig ändern. Mit anderen Worten: Du kannst den Wert der Variablen *y*, der Variablen *x* oder der Variablen *z* ändern; aber es ist nicht möglich, mehrere Änderungen gleichzeitig durchzuführen.

Folglich ist es an der Zeit, sich einmal Gedanken über zwei Dimensionen zu machen. Mit eindimensionalen Listen kannst du nur eine einzelne Liste mit jeweils einem Wert an jeder Position verwenden, während zweidimensionale Listen die Verwendung vieler Werte an jeder Position in der Liste gestatten. Hierzu gibst du an jeder Position der ursprünglichen Liste eine neue Liste an. Das sieht dann so aus:

```
twoDimensionalRainbowList = [[0, 0, 0], ❶  
                             [1, 1, 1], ❷  
                             [2, 2, 2], ❸  
                             [3, 3, 3],  
                             [4, 4, 4],  
                             [5, 5, 5]] ❹
```

Wenn du genau hinsiehst, erkennst du eine öffnende rechteckige Klammer in der ersten Zeile, gefolgt von einer Liste mit Nullen und einem Komma ❶. Das ist eine Liste innerhalb einer Liste! Die übergeordnete Liste bezeichnen wir als *äußere Liste*, und diese äußere Liste enthält *verschachtelte Listen*.

An Indexposition 1 befindet sich eine Liste, die drei Einsen enthält ❷; an Indexposition 2 gibt es eine weitere Liste, die drei Zweien enthält ❸. Dieses Prinzip wiederholt sich Zeile für Zeile. In der letzten Zeile ist eine Liste mit drei Fünfen angegeben, auf die eine eckige Klammer zum Schließen der äußeren Liste folgt ❹. Insgesamt stellt der Code eine Liste mit sechs Elementen dar, die jeweils wieder Listen sind. Es handelt sich also um eine zweidimensionale Liste!

Das Verhalten zweidimensionaler Listen wirst du besser verstehen, wenn du sie in Minecraft verwendest. Betrachten wir ein Beispiel. Wir passen *rainbowStack1.py* geringfügig an, damit das Programm auch mit der zweidimensionalen Liste funktioniert. Dieses neue Programm heißt *rainbowRows.py*:

rainbowRows
.py

```

from mcpi.minecraft import Minecraft
mc = Minecraft.create()

twoDimensionalRainbowList = [[0, 0, 0],
                              [1, 1, 1],
                              [2, 2, 2],
                              [3, 3, 3],
                              [4, 4, 4],
                              [5, 5, 5]]

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z

startingX = x ❶

for row in twoDimensionalRainbowList: ❷
    for color in row: ❸
        mc.setBlock(x, y, z, 35, color) ❹
        x += 1 ❺
    y += 1 ❻
    x = startingX ❼

```

Bevor ich daran gehe, den Code zu erklären, sieh dir erst einmal Bild 10.9 an. Dort ist die Ausgabe von *rainbowRows.py* dargestellt: eine Anordnung von Blöcken, die sechs Blöcke hoch (y-Dimension) und drei Blöcke breit (x-Dimension) ist.



Bild 10.9: Mit einer zweidimensionalen Liste eine Regenbogenwand bauen

Da wir es hier mit zwei Dimensionen zu tun haben, benötigen wir zwei for-Schleifen, um die Werte in der Liste `twoDimensionalRainbowList` auszugeben. Die erste Schleife iteriert über jedes Element in der äußeren Liste ❷, die zweite Schleife ❸ ist eine *verschachtelte* Schleife, weil sie innerhalb der ersten Schleife liegt, und bearbeitet jedes Element jeder verschachtelten Liste.

Bei der ersten Ausführung der äußeren Schleife beispielsweise wird zunächst das Element abgerufen, das sich an der Indexposition 0 der Liste `twoDimensionalRainbowList` befindet, und in einer Variablen namens `row` gespeichert ❷. Der Wert von `row` ist `[0, 0, 0]`, weil es sich um das erste Element in der Liste handelt.

Als Nächstes verarbeitet die zweite Schleife jedes Element in der Liste `row` und speichert dieses in der Variablen `color` ❸. Bei diesem ersten Durchlauf ist jedes Element 0. Danach positioniert das Programm die Blöcke und legt die Farbe des jeweiligen Wollblocks auf Grundlage der Variablen `color` fest ❹. Nachdem die verschachtelte Schleife alle drei Blöcke der ersten Reihe platziert hat, endet sie, und die äußere Schleife wird aufs Neue ausgeführt. Nun wandert die äußere Schleife zur Indexposition 1 weiter und speichert den vorgefundenen Wert (`[1, 1, 1]`) in der Variablen `row`. Danach wird die verschachtelte Schleife durchlaufen, um die Blöcke der passenden Farbe zu platzieren. Dieser Vorgang wiederholt sich, bis das Ende der Liste `twoDimensionalRainbowList` erreicht wurde.

Wenn du in zwei Dimensionen arbeitest, kannst du zwei Koordinatenvariablen gleichzeitig ändern. In diesem Beispiel inkrementieren wir die Variable `y` in der vorletzten Zeile der äußeren for-Schleife ❹; auf diese Weise wird jede Blockreihe auf die vorherige Reihe gesetzt. Außerdem inkrementieren wir die Variable `x` innerhalb der verschachtelten for-Schleife ❺, damit sichergestellt ist, dass die Blöcke jeweils in einer Reihe angeordnet werden. Danach müssen wir die Variable `x` für jede Iteration der äußeren for-Schleife ❷ auf den Ursprungswert setzen (dieser ist in der Variablen `startingX` gespeichert ❶). Das Zurücksetzen der Variablen `x` hat zur Folge, dass der erste Block jeder Reihe direkt auf den ersten Block der vorangegangenen Reihe gesetzt wird usw. Das heißt, die Blockreihen sitzen korrekt aufeinander.

Werte in 2-D-Listen aufrufen

Beim Abrufen oder Festlegen eines Werts in einer eindimensionalen Liste verwendest du eckige Klammern und die Indexposition. Der folgende Code beispielsweise erstellt eine Liste namens `scores`, die Spielstände speichert und dann das Element an Indexposition 2 von 6 auf 7 setzt:

```
scores = [1, 5, 6, 1]
scores[2] = 7
```

Das Zugreifen auf einen Wert oder das Verändern eines Werts in einer zweidimensionalen Liste verläuft nicht viel anders. Auch hier verwendest du eckige Klammern und die Indexposition; weil du aber auf zwei Listen gleichzeitig zugreifst, musst du auch immer zwei Sätze mit Indizes und eckigen Klammern verwenden. Sehen wir uns das an.

Hier ist die Liste, die wir schon kennen:

```
twoDimensionalRainbowList = [[0, 0, 0],
                              [1, 1, 1],
                              [2, 2, 2],
                              [3, 3, 3],
                              [4, 4, 4],
                              [5, 5, 5]]
```

Wenn wir jetzt das zweite Element (Indexposition 1) in der ersten Liste (Indexposition 0) auf den Wert 7 setzen wollen, dann müssen wir folgenden Code verwenden:

```
twoDimensionalRainbowList[0][1] = 7
```

Damit zwei Listen verwendet werden und eine davon in der anderen verschachtelt ist, müssen wir auch zwei Sätze eckiger Klammern angeben. Der erste Klammersatz wählt die Indexposition 0 der Liste `twoDimensionalRainbowList` aus, also der ersten verschachtelten Liste. In die zweite Klammer setzen wir die Indexposition, die wir in der verschachtelten Liste bearbeiten möchten, also 1. Dann setzen wir den Wert des Elements an dieser Position mit dem Gleichheitszeichen auf 7.

Diesen Code habe ich nun zum Programm *rainbowRows.py* (Seite 274) hinzugefügt und es dann erneut ausgeführt. Bild 10.10 zeigt das Ergebnis. Beachte, dass der zweite Block der ersten Reihe sein Aussehen geändert hat, weil wir den Wert in der verschachtelten Liste auf 7 gesetzt haben.

Wenn du den Wert eines Elements in einer zweidimensionalen Liste abrufen möchtest, musst du ebenfalls zwei Sätze eckiger Klammern verwenden. Möchtest du beispielsweise den Wert an der ersten Position (Index 0) der letzten Reihe (Index 5) ausgeben, dann musst du folgenden Code dafür verwenden:

```
print(twoDimensionalRainbowList[5][0])
```

Dieser Code gibt den Wert 5 aus.



Bild 10.10: Ändern eines Werts in einer verschachtelten Liste, um ein anderes Ergebnis zu erhalten

Mission 61: Wir verkünsteln uns

Pixel sind winzige einfarbige Vierecke, aus denen sich Bilder auf deinen Computer zusammensetzen. Fasst du viele solcher Pixel in einem Raster zusammen, dann kann dein Computer Text, Bilder, Videos und alles Mögliche andere auf dem Monitor anzeigen. Alle Fotos und grafischen Elemente auf deinem Computer werden mit Pixeln dargestellt.

Pixel Art – also das Erstellen von Bildern mithilfe verschiedenfarbiger Blöcke – ist unter den Minecraft-Spielern sehr populär. Zu den beliebtesten Werken gehören etwa Bilder von Charakteren aus 2-D-Videospielen. Du kannst Pixel Art entweder manuell oder – natürlich – mithilfe eines Python-Programms in Minecraft erstellen.

Im folgenden Programm benutzen wir zu genau diesem Zweck eine 2-D-Liste und verschachtelte Schleifen. Listing 10.6 enthält den Anfang

des Programms. Kopiere es in eine neue Datei namens *pixelArt.py* und speichere sie im Ordner *forLoops*.

pixelArt.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

pos = mc.player.getTilePos()
x, y, z = pos.x, pos.y, pos.z

blocks = [[35, 35, 35, 35, 35, 35, 35, 35], ❶
          [35, 35, 35, 35, 35, 35, 35, 35],
          [35, 35, 35, 35, 35, 35, 35, 35],
          [35, 35, 35, 35, 35, 35, 35, 35]]

for row in reversed(blocks): ❷
    for block in row:
        mc.setBlock(x, y, z, block)
        x += 1
    y += 1
    x = pos.x
```

Listing 10.6: Zweidimensionale Liste zum Zeichnen eines Smileys

Das Programm erstellt eine zweidimensionale Liste namens `blocks`, die die Block-IDs enthält ❶, und verwendet dann zwei Schleifen zum Platzieren der Blöcke in der Minecraft-Welt ❷. Um dafür zu sorgen, dass die erste Reihe der Liste sich oben befindet, wenn sie in Minecraft platziert wird, während die untere Reihe ganz nach unten kommt, ist die `reversed()`-Funktion Bestandteil der ersten `for`-Schleife ❷. Andernfalls würde das Bild nämlich bezogen auf die Reihenfolge der Liste `blocks` auf dem Kopf stehen.

Zurzeit sind alle Blöcke weiße Wollblöcke, d. h., es wird kein Bild angezeigt. Um das Programm zu vervollständigen, musst du die zweidimensionale Liste mit den Blöcken so umschreiben, dass ein Smiley gezeichnet wird, wie er in Bild 10.11 dargestellt ist.

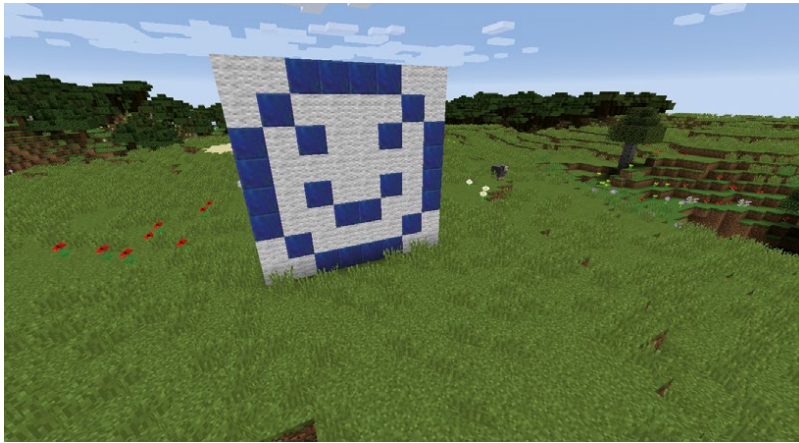


Bild 10.11: Ein Smiley aus Blöcken

Ändere die Werte in den Listen so ab, dass die Ausgabe Bild 10.11 entspricht. Zu diesem Zweck musst du einige der Werte in den Listen von Wolle (Block-ID 35) auf Lapislazuli (Block-ID 22) umstellen. Die erste Zeile beispielsweise müsste dann so aussehen:

```
blocks = [[35, 35, 22, 22, 22, 22, 35, 35],
```

Außerdem musst du einige weitere Reihen zur Liste `blocks` hinzufügen, sodass die Höhe des Bildes der Höhe in der Abbildung entspricht.

Bonus: Sei kreativ!

Versuche einmal, die Werte in der zweidimensionalen Liste in *pixel-Art.py* so abzuändern, dass ein anderes Bild angezeigt wird. Du kannst auch die Längen der Listen bearbeiten. Skizziere deine Entwürfe zunächst auf Rechenpapier. Dann wandelst du sie in zweidimensionale Listen um und baust sie in Minecraft nach.

2-D-Listen mit Schleifen erzeugen

Programme, die Zufallszahlen verwenden, machen Spaß, denn sie verhalten sich bei jeder Ausführung etwas anders. Ich habe in der Vergangenheit eine Menge Programme geschrieben, die Zufallszahlen in Verbindung mit zweidimensionalen Listen verwenden, um Bilder zu erstellen.

Jede Zufallszahl kann eine andere Farbe oder – im Fall von Minecraft – auch einen anderen Block darstellen.

Nachfolgend siehst du den Anfang eines Programms, das Zufallszahlen erzeugt und diese in einem zweidimensionalen Listensatz ablegt:

```
import random
randomNumbers = [] ❶
for outer in range(10):
    randomNumbers.append([]) ❷
    for inner in range(10):
        number = random.randint(1, 4) ❸
        randomNumbers[outer].append(number)
print(randomNumbers)
```

Das Programm beginnt mit einer leeren Liste namens `randomNumbers` ❶. Bei jeder Ausführung der äußeren `for`-Schleife wird eine neue leere Liste in die Liste `randomNumbers` eingefügt ❷. Dann generiert das Programm in der inneren Schleife eine Zufallszahl zwischen 1 und 4 und speichert sie in der inneren Liste ❸. Die innere Schleife wird zehnmal wiederholt und erzeugt dabei zehn Elemente in jeder inneren Liste.

Nachdem wir für eine bessere Lesbarkeit Zeilenumbrüche hinzugefügt haben, präsentiert sich die Ausgabe des Programms wie folgt (beachte die zehn Elemente in den zehn inneren Listen):

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

Wenn du Zufallszahlen in deinen 2-D-Kreationen verwendest, kannst du in Minecraft einige sehr coole Effekte erzeugen, die von Hand nur sehr schwierig umzusetzen wären.

Mission 62: Eine verwitterte Mauer

Wenn ich in Minecraft Mauern baue, dann verwende ich normalerweise mehrere unterschiedliche Blöcke. Indem ich einige Bruchsteinblöcke durch bemooste Bruchsteinblöcke ersetze, kann ich aus einer langweiligen glatten Wand etwas machen, das alt, wettergegerbt, organisch – mit einem Wort: cool – aussieht. Zwar macht es Spaß, eine Mauer Stein für Stein selbst aufzubauen, doch gelingt es mir nie, die Blöcke so unregelmäßig anzuordnen, dass es wirklich nach *Zufall* aussieht. Deswegen verwenden wir ein Python-Programm (du *wusstest* es!), damit alte und verwitterte Mauern wirklich realistisch aussehen.

Um eine solche Wand mit Python zu bauen, musst du dein Programm in zwei separate Schritte unterteilen:

1. Erstelle eine zweidimensionale Liste und speichere darin die Blockwerte.
2. Gib diese zweidimensionale Liste in der Minecraft-Welt aus.

Als Einstieg enthält Listing 10.7 den Code zum Auswählen eines zufälligen Blockwerts, zum Einrichten der Liste und zum Abrufen der Spielerposition. Kopiere das Listing in eine neue Datei namens *brokenWall.py* und speichere sie im Ordner *forLoops*.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import random

def brokenBlock(): ❶
    brokenBlocks = [48, 67, 4, 4, 4, 4]
    block = random.choice(brokenBlocks)
    return block

pos = mc.player.getTilePos()
x, y, z = pos.x, pos.y, pos.z

brokenWall = []
height, width = 5, 10

# Liste der beschädigten Blöcke erstellen

# Blöcke platzieren
```

brokenWall.py

Listing 10.7: Gerüst des Programms zum Erstellen einer beschädigten Mauer

Die Funktion `brokenBlock()` gibt einen zufälligen Blockwert zurück, der zum Bauen der Mauer verwendet wird ❶. Die Variablen `width` und `height` legen Breite und Höhe der Mauer fest.

Um das Programm zu vervollständigen, musst du eine zweidimensionale Liste mit Blockwerten erstellen und dann mit diesen Werten deine Mauer in Minecraft entwerfen.

Beginne mit der leeren Liste `brokenWall`. Mit einer `for`-Schleife, die in einer anderen `for`-Schleife verschachtelt ist, erzeugst du zufällige Blockwerte unter Verwendung der Funktion `brokenBlock()`. Speichere diese Blockwerte in Listen, und speichere die Listen dann in der Liste `brokenWall`. Danach platzierst du die Blöcke mit einem zweiten Satz verschachtelter Schleifen in Minecraft.

Wenn dein Programm vollständig ist, begib dich zu dem Ort, an dem du die verwitterte Mauer in deiner Minecraft-Welt bauen möchtest, und führe den Code aus. Du kannst das Programm auch verwenden, um ein Schloss auszuschnücken oder verwunschen wirkende Ruinen in einem dunklen Wald zu bauen. Experimentiere mit verschiedenen Orten und finde heraus, was dir am besten gefällt. Bild 10.12 zeigt, wie die Mauer aussieht, wenn das Programm abgelaufen ist.

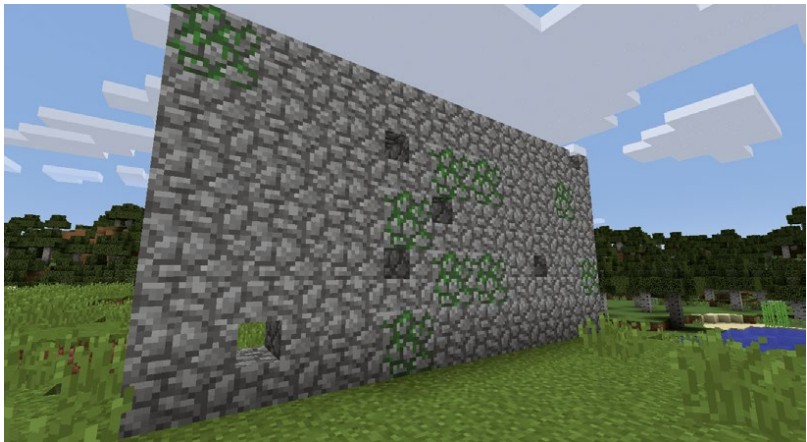


Bild 10.12: Mauer mit zufällig generierten beschädigten Blöcken. Vielleicht spukt es hier!

Bonus: Bau doch mal eine bunte Wand

Ändere im Programm *brokenWall.py* die Blockwerte in der Liste *brokenBlocks* der Funktion *brokenBlock()* so ab, dass alle möglichen Arten von Mauern gebaut werden können. Setze beispielsweise die Blockwerte auf verschiedene Wollfarben und betrachte das Ergebnis.

Und jetzt in drei Dimensionen

Selbstverständlich ist Minecraft ein Spiel, in dem drei Dimensionen verwendet werden. Und ebenso selbstverständlich hast du auch alle drei Dimensionen bereits in diesem Buch verwendet. Jede der drei Variablen *x*, *y* und *z*, die du schon in vielen Programmen benutzt hast, stellt eine Dimension dar.

Wir haben bereits gesehen, wie man einen Listensatz in einem anderen ablegt, um eine zweidimensionale Liste zu erstellen und damit coole Pixel Art und verwitterte Mauern zu bauen. Fügen wir nun einer zweidimensionalen Liste eine dritte Listengruppe hinzu, dann entsteht eine dreidimensionale Liste. Damit eröffnet sich dir buchstäblich eine ganz neue Dimension!

Dreidimensionale Listen sind in Minecraft extrem nützlich, denn mit ihrer Hilfe kannst du 3-D-Strukturen wie Gebäude, Skulpturen und viele andere Dinge duplizieren.

In der dreidimensionalen Liste, die in Listing 10.8 gezeigt ist, sind vier Listen verschachtelt. Der Clou ist aber, dass sich in jedem Index dieser verschachtelten Listen noch eine weitere Liste befindet! Im Grunde genommen ist jedes Element in dieser Liste eine 2-D-Liste. Ich habe ein paar Leerkommentare eingefügt, um die Lesbarkeit der Liste zu verbessern.

```
cube = [[ [57, 57, 57, 57],  
          [57, 0, 0, 57],  
          [57, 0, 0, 57],  
          [57, 57, 57, 57]],  
        #  
        [[57, 0, 0, 57],  
         [0, 0, 0, 0],  
         [0, 0, 0, 0],  
         [57, 0, 0, 57]],
```

```
#
[[57, 0, 0, 57],
 [0, 0, 0, 0],
 [0, 0, 0, 0],
 [57, 0, 0, 57]],
#
[[57, 57, 57, 57],
 [57, 0, 0, 57],
 [57, 0, 0, 57],
 [57, 57, 57, 57]]]
```

Listing 10.8: Dreidimensionale Liste mit verschachtelten Listen

Mit einem solchen Code lässt sich eine coole Würfelstruktur erstellen. Und genau das werden wir mit dem Programm tun, das wir nun unter die Lupe nehmen werden.

3-D-Listen ausgeben

Listen mit drei Dimensionen eignen sich hervorragend, um Daten von dreidimensionalen Objekten zu speichern, wie etwa von deinen beeindruckenden Minecraft-Gebäuden. Die Speicherung dreidimensionaler Objekte ist wichtig, und ihre korrekte Ausgabe an Minecraft ist es gleichermaßen. Weil eine dreidimensionale Liste eine Liste in einer Liste ist, die sich innerhalb einer Liste befindet, kannst du für den Zugriff auf alle Daten eine for-Schleife in einer anderen for-Schleife verwenden, die sich ihrerseits ebenfalls in einer for-Schleife befindet. Anders formuliert: Du verwendest hierzu drei ineinander verschachtelte for-Schleifen.

In Listing 10.9 habe ich die dreidimensionale Liste aus Listing 10.8 kopiert und ein Programm namens *cube.py* erstellt. Dieses Programm verwendet drei verschachtelte for-Schleifen, um alle Werte der dreidimensionalen Liste nacheinander auszugeben und so eine Würfelstruktur in der Minecraft-Welt zu bauen.

cube.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z
cube = [[[57, 57, 57, 57], [57, 0, 0, 57], [57, 0, 0, 57], [57, 57, ↵
57, 57]],
        [[57, 0, 0, 57], [0, 0, 0, 0], [0, 0, 0, 0], [57, 0, 0, 57]],
```



```

[[57, 0, 0, 57], [0, 0, 0, 0], [0, 0, 0, 0], [57, 0, 0, 57]],
[[57, 57, 57, 57], [57, 0, 0, 57], [57, 0, 0, 57], [57, 57, 57, 57]]]

startingX = x
startingY = y ❶
for depth in cube: ❷
    for height in reversed(depth):
        for block in height:
            mc.setBlock(x, y, z, block)
            x += 1
        y += 1
        x = startingX
    z += 1 ❸
    y = startingY ❹

```

Listing 10.9: Code zum Erstellen eines dreidimensionalen Diamantwürfels

Bild 10.13 stellt das Ergebnis dar.



Bild 10.13: Ein mit dem Programm `cube.py` erstellter Würfel

Der Code in `cube.py` erinnert stark an das Programm `rainbowRows.py` (Seite 274), mit dem wir die Regenbogenmauer gebaut haben. Der wesentliche Unterschied besteht darin, dass `cube.py` drei `for`-Schleifen anstelle von zweien verwendet, weil hier eben eine dreidimensionale Liste bearbeitet wird. Die zusätzliche `for`-Schleife fügt eine weitere Dimension zur Struktur hinzu: die Tiefe ❷. Nun verfügt die Struktur über Breite, Höhe und Tiefe.

Bei jeder Ausführung der äußeren Schleife für die Tiefe im Würfel wird eine zweidimensionale Liste unter Verwendung zweier verschachtel-

ter Schleifen für die Höhe in `reversed(depth)` und für die Blockhöhe erstellt. Der Code in den beiden verschachtelten Schleifen ähnelt dem im Programm *rainbowRows.py*, d. h., diese Schleifen errichten eine Mauer in Minecraft.

Sehen wir uns das Ergebnis der äußeren Schleife bei jeder Wiederholung an. So können wir nachvollziehen, wie der Würfel Schritt für Schritt gebaut wird. Bei der ersten Ausführung der äußeren Schleife gibt sie die Blöcke an der Indexposition 0 der Liste *cube* aus. Diese Liste sieht wie folgt aus:

```
[[57, 57, 57, 57],  
 [57, 0, 0, 57],  
 [57, 0, 0, 57],  
 [57, 57, 57, 57]]
```

Bild 10.14 zeigt die Ausgabe: die erste Blockmauer.

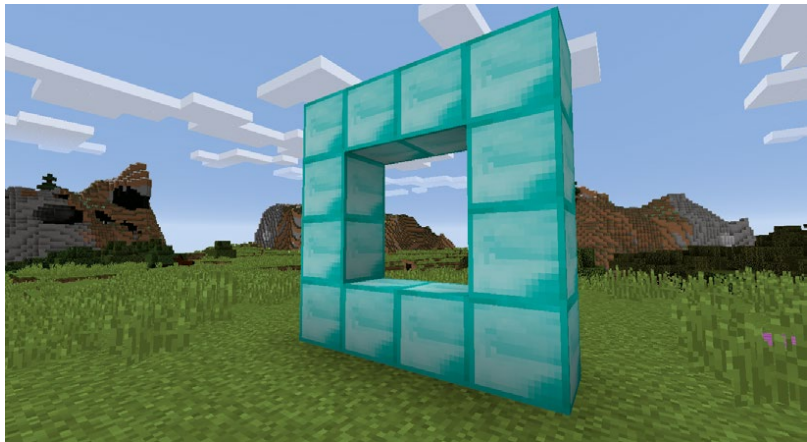


Bild 10.14: Ergebnis der ersten zweidimensionalen Schleife, Index 0 in *cube*

Nach dem Erstellen einer zweidimensionalen Liste im Spiel wird der Wert der Variablen *z* in *cube.py* ③ erhöht, um entlang der *z*-Achse einen Block weiter zu gehen. Hiermit wird die Tiefe in Blöcken angegeben, d. h., wir bauen nicht einfach eine Mauer. Außerdem müssen wir den Wert der Variablen *y* bei ④ auf den Ursprungswert ① zurücksetzen, damit die Blöcke in der untersten Reihe des Würfels bei jeder Ausführung der äußeren Schleife korrekt aneinander ausgerichtet werden. Würden wir die Variable *y* nicht zurücksetzen, dann würde sich die *y*-Koordinate jedes Block-

satzes immer stärker erhöhen, und es entstünde eine ziemlich seltsam aussehende Treppe! Bild 10.15 zeigt, wie das aussehen würde.

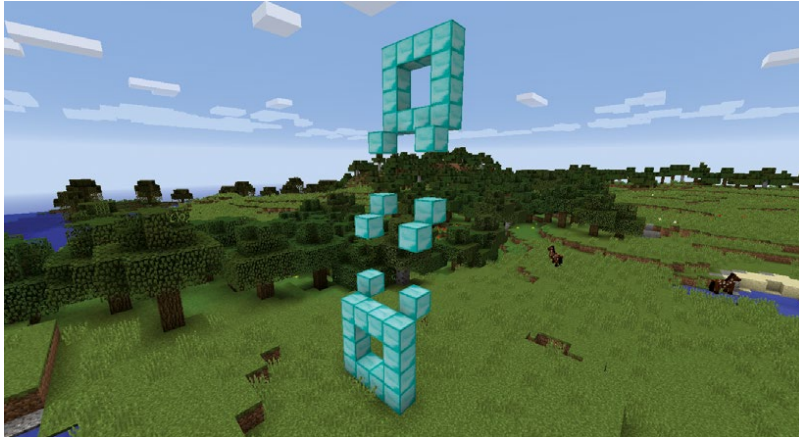


Bild 10.15: Damit das hier nicht passiert, setzen wir die Variable y zurück!

Bei der zweiten Ausführung der äußeren Schleife werden die Blöcke an der Indexposition 1 der Liste cube ausgegeben. Das sieht so aus:

```
[[57, 0, 0, 57],  
 [0, 0, 0, 0],  
 [0, 0, 0, 0],  
 [57, 0, 0, 57]]
```

Hierdurch wird der nächste Teil des Würfels hinzugefügt (Bild 10.16). Nachdem dieser Teil des Würfels erstellt ist, wird die Variable z um 1 erhöht ③ und die Variable y wieder auf ihren Ursprungswert zurückgesetzt ④.

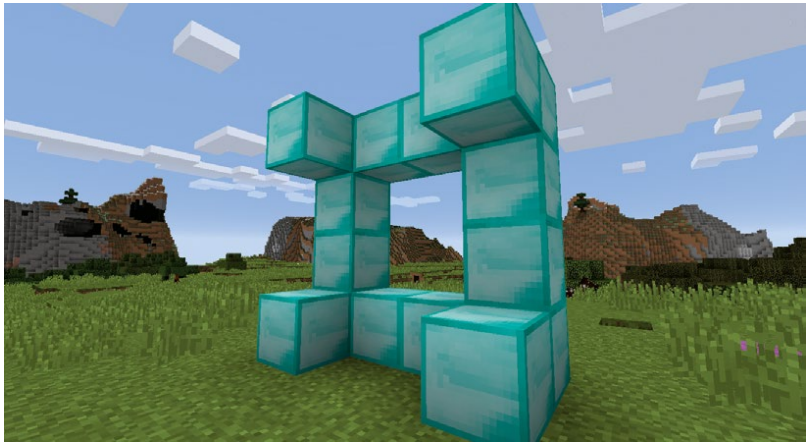


Bild 10.16: Ergebnis der zweiten zweidimensionalen Schleife, Index 1 in cube

Bei der nächsten Ausführung der Schleife gibt diese die zweidimensionale Liste an der Indexposition 2 von cube aus:

```
[[57, 0, 0, 57],  
 [0, 0, 0, 0],  
 [0, 0, 0, 0],  
 [57, 0, 0, 57]]
```

Bild 10.17 zeigt das Ergebnis. Wiederum wird der Wert von z um 1 erhöht und der Wert von y zurückgesetzt.

Danach wird die Schleife ein viertes und letztes Mal durchlaufen, wobei die Indexposition 3 von cube ausgegeben wird:

```
[[57, 57, 57, 57],  
 [57, 0, 0, 57],  
 [57, 0, 0, 57],  
 [57, 57, 57, 57]]
```

Bild 10.18 zeigt die fertige Würfelstruktur.

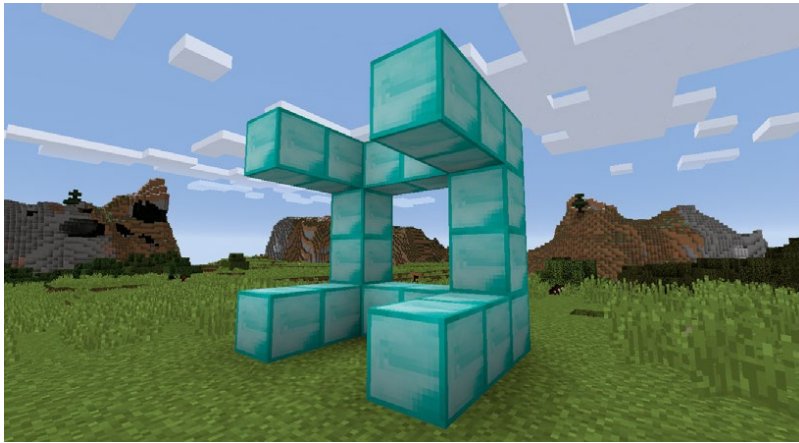


Bild 10.17: Ergebnis der dritten zweidimensionalen Schleife, Index 2 in cube

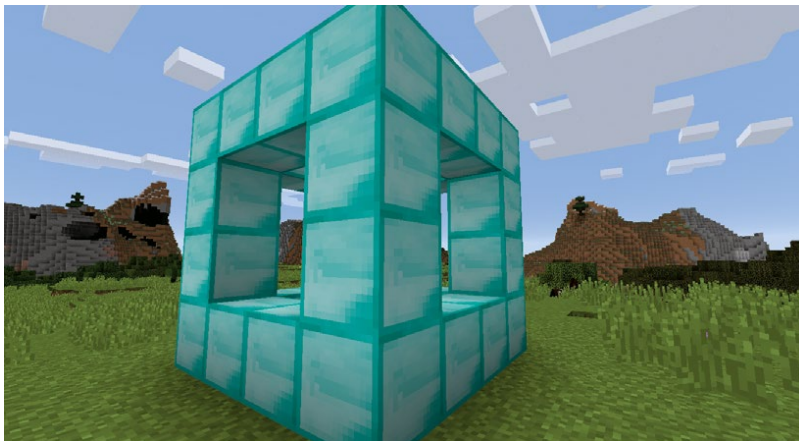


Bild 10.18: Ergebnis der letzten zweidimensionalen Schleife an der letzten Indexposition in cube

Experimentiere ein wenig mit diesem Programm: Verwende einen anderen Blocktyp, mache den Würfel größer oder tue irgendetwas damit, was dir gerade einfällt. Im nächsten Abschnitt zeige ich dir, wie du auf Werte in dreidimensionalen Listen zugreifst. So kannst du einige dieser Änderungen durchführen.

Werte in 3-D-Listen aufrufen

Die Werte in dreidimensionalen Listen können auf gleiche Weise geändert werden wie solche in ein- und zweidimensionalen Listen: mit eckigen Klammern und Indexpositionen.

Beginnen wir mit unserer dreidimensionalen Diamantwürfelloste:

```
cube = [[[57, 57, 57, 57],
          [57, 0, 0, 57],
          [57, 0, 0, 57],
          [57, 57, 57, 57]],
         #
         [[57, 0, 0, 57],
          [0, 0, 0, 0],
          [0, 0, 0, 0],
          [57, 0, 0, 57]],
         #
         [[57, 0, 0, 57],
          [0, 0, 0, 0],
          [0, 0, 0, 0],
          [57, 0, 0, 57]],
         #
         [[57, 57, 57, 57],
          [57, 0, 0, 57],
          [57, 0, 0, 57],
          [57, 57, 57, 57]]]
```

Nehmen wir nun an, ich möchte auf der Vorderseite den Block unten links in Gold verwandeln.

Hierzu muss ich zunächst den Index der Würfelloste aufrufen, die die Vorderseite des Würfels enthält; dies ist Index 0. Der erste Teil des Ausdrucks sieht also so aus:

```
cube[0]
```

Wenn ich den Wert dieses Ausdrucks ausgeben würde, sähe das wie folgt aus (ich habe die Ausgabe formatiert, damit sie besser lesbar ist):

```
[[57, 57, 57, 57],
 [57, 0, 0, 57],
 [57, 0, 0, 57],
 [57, 57, 57, 57]]
```

Diese zweidimensionale Liste stellt die Vorderseite des Würfels dar. Als Nächstes möchte ich auf die unterste Reihe zugreifen. Diese hat den Index 3. Ich füge also `[3]` zu meinem Ausdruck hinzu:

```
cube[0][3]
```

Würde ich die an dieser Position gespeicherte Liste ausgeben, dann sähe das so aus:

```
[57, 57, 57, 57]
```

Abschließend möchte ich in nämlicher Reihe auf den linken Block zugreifen. Die Reihe hat den Index 3. Deswegen sieht der endgültige Ausdruck, um den Block unten links in Gold zu ändern, wie folgt aus:

```
cube[0][3][3] = 41
```

Wenn ich das Programm *cube.py* um diese Zeile ergänzt ausführe, dann erhalte ich einen Würfel aus Diamant, der einen einzelnen goldenen Block enthält (Bild 10.19). (Ob der goldene Block vorne links oder z.B. hinten rechts zu sehen ist, hängt davon ab, in welche Richtung du schaust. X wird ja in Richtung Osten größer und Y in Richtung Süden – siehe auch <http://minecraft-de.gamepedia.com/Anleitungen/Orientierung#Koordinaten>.)

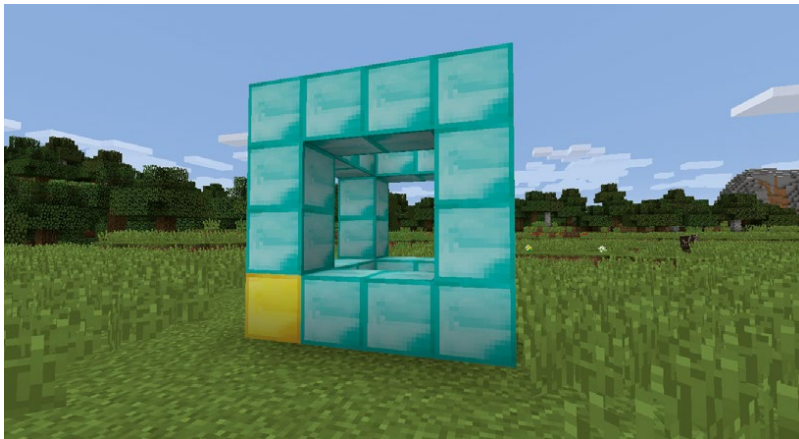


Bild 10.19: Geänderter Würfel mit einem goldenen Block

Mission 63: Wie man ein Gebäude dupliziert

Zugegeben: Man spart mit dem Bauen von Dingen in Minecraft eine Menge Zeit, wenn man dazu Python-Programme verwendet. Trotzdem treiben viele Leute – darunter auch ich – erheblichen Aufwand, um ihre Gebäude mit den vielen kleinen Details auszustatten, die den Unterschied ausmachen: Bilder, Möbel usw. Manchmal muss man hierbei eine identische Kopie eines bestimmten Objekts erstellen, und das Kopieren per Hand ist sehr zeitaufwendig. Auch das Platzieren jedes einzelnen Blocks dauert, und dann besteht noch die Gefahr, dass man den Block an der falschen Stelle ablegt. Die naheliegende Lösung besteht darin, ein Programm zu schreiben, das ein Gebäude aus Minecraft kopiert und dann im Spiel eine Kopie für dich baut.

Das fertige Programm muss dabei zweierlei erledigen: Zunächst muss es einen Bereich im Spiel kopieren und in einer dreidimensionalen Liste ablegen, und dann muss es die kopierte Struktur mithilfe dieser Liste bauen.

Nachfolgend siehst du den Anfang dieses Programms in Listing 10.10. Kopiere es in eine neue Datei und speichere sie als *duplicateArea.py* im Ordner *forLoops*.

duplicateArea
.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

def sortPair(val1, val2): ❶
    if val1 > val2:
        return val2, val1
    else:
        return val1, val2

def copyStructure(x1, y1, z1, x2, y2, z2): ❷
    # Höchste und niedrigste Werte für x, y und z sortieren
    x1, x2 = sortPair(x1, x2)
    y1, y2 = sortPair(y1, y2)
    z1, z2 = sortPair(z1, z2)

    width = x2 - x1
    height = y2 - y1
    length = z2 - z1

    structure = []

    print("Please wait...")
```

```

    # Struktur kopieren ❸

    return structure

def buildStructure(x, y, z, structure): ❹
    xStart = x
    yStart = y

    # Struktur erstellen ❺

    # Position der ersten Ecke abrufen
    input("Move to the first corner and press enter in this window") ❻
    pos = mc.player.getTilePos()
    x1, y1, z1 = pos.x, pos.y, pos.z

    # Position der zweiten Ecke abrufen
    input("Move to the opposite corner and press enter in this window") ❼
    pos = mc.player.getTilePos()
    x2, y2, z2 = pos.x, pos.y, pos.z

    # Gebäude kopieren
    structure = copyStructure(x1, y1, z1, x2, y2, z2) ❽

    # Position für Kopie festlegen
    input("Move to the position you want to create the structure ↵
        and press ENTER in this window") ❾
    pos = mc.player.getTilePos()
    x, y, z = pos.x, pos.y, pos.z
    buildStructure(x, y, z, structure)

```

Listing 10.10: Wenn das Programm vollständig ist, kann man damit Gebäude duplizieren.

Das Programm ist in verschiedene Teile gegliedert: Zuerst wird mit der Funktion `sortPair()` ❶ ein Wertepaar in ein Tupel einsortiert, wobei der niedrigste Wert an der ersten und der höchste an der zweiten Indexposition steht. Übergabe ich `sortPair()` beispielsweise die Argumente 9 und 3, dann würde die Funktion als Tupel (3, 9) zurückgeben, da 3 kleiner ist als 9. Ich verwende diese Funktion zum Sortieren von Wertpaaren für x, y und z, damit die Werte für die Variablen `width`, `length` und `depth` bei den Berechnungen stets positiv sind.

Als Nächstes kopiert die Funktion `copyStructure()` ❷ die Struktur aus der Spielwelt. Sie ist allerdings noch nicht vollständig ❸. Die Funktion `buildStructure()` ❹ baut die Struktur; auch sie ist noch nicht fertig ❺. Du musst diese beiden Funktionen in dieser Mission vervollständigen.

Ich habe einen kleinen Trick verwendet, um die Koordinaten des zu kopierenden Gebäudes und die Position im Spiel abzurufen, an der die Kopie gebaut werden soll: Bei der Funktion `input()` fordert das Programm dich zunächst auf, zu einer Ecke des Gebäudes zu gehen und dann **Enter** zu drücken ⑥. Die Funktion `input()` sorgt dafür, dass der Code wartet, bis du den Spieler an die gewünschte Position gestellt hast. Sobald du **Enter** gedrückt hast, wird die Position des Spielers mit der Funktion `getTilePos()` abgerufen. Danach wiederholen wir das Ganze an der schräg gegenüberliegenden Ecke des Gebäudes ⑦. Die beiden Koordinatensätze werden dann von der Funktion `copyStructure()` zum Kopieren des Gebäudes benutzt ⑧. (Wenn größere Anlagen kopiert werden, kann die Ausführung dieses Programmteils eine ganze Zeit lang dauern.) Abschließend begibst du dich zu dem Ort, an dem das neue Gebäude gebaut werden soll, und drückst **Enter** ⑨, um die aktuelle Spielerposition an die Funktion `buildStructure()` zu übergeben.

Zum Abschließen des Programms musst du die Funktionen `copyStructure()` und `buildStructure()` vervollständigen. Erweitere die Funktion `copyStructure()` mit drei verschachtelten Schleifen, um alle als Argument übergebenen Blöcke zwischen den Koordinaten bei ③ in eine dreidimensionale Liste zu kopieren. Um die Funktion `buildStructure()` zu vervollständigen, fügst du bei ⑤ drei verschachtelte `for`-Schleifen hinzu, die die Blockwerte aus der dreidimensionalen Liste ausgeben. Die Funktion sollte die angegebenen Koordinaten in ihren Argumenten verwenden.

Achte darauf, dass das Programm die Positionen `x`, `y` und `z` innerhalb der Struktur abarbeitet. Um die Positionen `x`, `y` und `z` zu ändern, verwendest du `for`-Schleifen.

Auch wenn *duplicateArea.py* schon ein recht langes Programm ist, ist es sehr nützlich und den Aufwand wert. Wenn du diese Mission abgeschlossen hast, kannst du ganze Städte in deiner Minecraft-Welt bauen! Ich habe mit *duplicateArea.py* eine interessante Klippe dupliziert, die ich auf der Reise durch meine Welt gefunden habe. Bild 10.20 zeigt diese Klippe, die ich kopieren wollte.



Bild 10.20: Eine schöne Klippe. Darum habe ich sie kopiert.

Wenn du Kopien mit `duplicateArea.py` erstellst, stelle dich (sofern das zu kopierende Objekt ein Gebäude ist) vor eine untere Ecke des Objekts, das du kopieren willst. Dann drückst du **Enter** in IDLE. Bild 10.21 zeigt mich an der ersten Ecke.



Bild 10.21: Zunächst bin ich zu einer Ecke der Struktur gegangen und habe dann in IDLE **Enter** gedrückt.

Danach fliegst du nach oben und zur gegenüberliegenden Ecke des zu kopierenden Objekts und drückst erneut **Enter**. In Bild 10.22 bin ich hochgefliegen und habe mich auf die andere Seite der Klippe begeben.



*Bild 10.22: Dann bin ich zur gegenüberliegenden Ecke geflogen und habe erneut **Enter** gedrückt.*

Danach wird dir eine Meldung angezeigt, die um ein bisschen Geduld bittet, während das Objekt vom Programm kopiert wird. Begib dich dann an die Position, an der du das Duplikat erstellen möchtest, und warte, bis die Frage nach dem Standort des kopierten Objekts angezeigt wird (Bild 10.23).

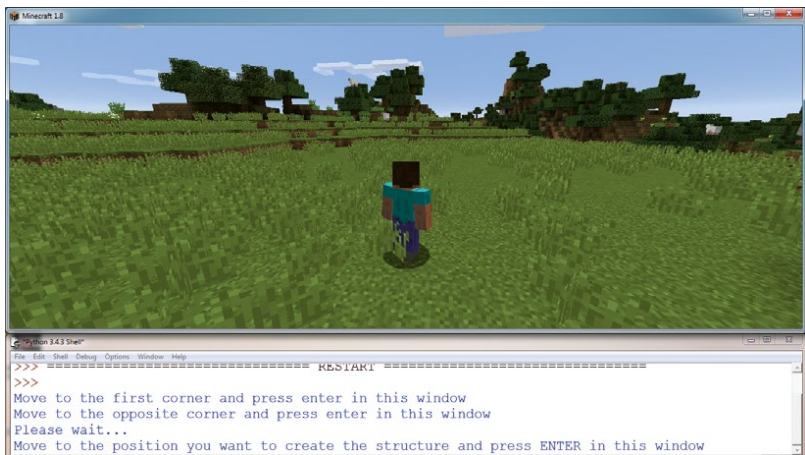


*Bild 10.23: Ich habe gewartet, bis die Struktur kopiert wurde. Dann bin ich zum vorhergesehenen Standort der Kopie gegangen und habe **Enter** gedrückt, um sie zu bauen.*

Wenn du die gewünschte Zielposition erreicht hast, drücke **Enter**. Nun entsteht vor deinen Augen die Kopie! Bild 10.24 zeigt meine kopierte Klippe.



Bild 10.24: Sieht haargenau so aus wie das Original: die kodierte Klippe.

Was du gelernt hast

In diesem Kapitel haben wir viel Stoff behandelt. Du hast erfahren, wie for-Schleifen bei Listen verwendet werden und wie du die Funktion `range()` einsetzt. Außerdem sind wir näher auf die Feinheiten von for-Schleifen und Listen eingegangen – beispielsweise auf das Umkehren von Listen, auf das Ausführen von Schleifen für Wörterbücher und auf das Unterbrechen von for-Schleifen. Wir haben zwei- und dreidimensionale Listen mit verschachtelten Schleifen erstellt, die zum Aufbau einer beeindruckenden Minecraft-Welt unentbehrlich sind.

Mit Funktionen wie dem Bau von Treppen und Pyramiden, dem Kopieren von Objekten und dem Erstellen von Bildern per Pixel Art hast du nun mehr Möglichkeiten als je zuvor, um in Minecraft kreativ zu werden. Die Programme in diesem Kapitel gehören zu meinen Lieblingsprogrammen in diesem Buch. Sie sollen dir helfen, auch sehr anspruchsvolle Projekte umzusetzen!

In diesem wie auch im vorangegangenen Kapitel lag der Schwerpunkt auf Listen und for-Schleifen, da diese eng miteinander zusammenhängen. In Kapitel 11 werden wir uns Dateien und Modulen zuwenden. Sie gehören in den Bereich der Funktionen, und auch sie hast du bereits im ganzen Buch verwendet. In deinen Missionen wirst du dann erfahren, wie du Strukturen in Dateien speicherst und daraus lädst.

Kapitel 11

Gebäude speichern und laden mit Dateien und Modulen

Dateien spielen in der IT eine wichtige Rolle. Du kannst Daten darin dauerhaft speichern und sie zu einem späteren Zeitpunkt für die Verwendung in deinen Programmen neu laden. Bisher haben wir in unseren Programmen Daten nur in Variablen abgelegt. Du hast also entweder einkodierte Daten oder aber solche verwendet, die vom Benutzer eingegeben wurden.

Natürlich kannst du mit solchen Daten bereits Beeindruckendes zaubern, aber trotzdem bist du auf einzelne Sessions und auf deine eigene Minecraft-Welt beschränkt. Wenn du aber einmal weißt, wie du Daten speicherst und abrufst, kannst du deine Minecraft-Kreationen in einer Datei ablegen und anschließend in jede beliebige Minecraft-Welt laden – sogar in die Spiele deiner Freunde!

In diesem Kapitel wirst du erfahren, wie du Eingaben aus Dateien abrufst und Daten in Dateien ausgibst. Wir werden einige in Python integrierte Funktionen verwenden, und ich zeige dir, wie du zwei Python-*Module* – nämlich `pickle` und `shelve` – zum Speichern deiner Minecraft-Kreationen verwendest.

Mit Modulen kannst du die Leistungsfähigkeit von Python erweitern. Du kannst beispielsweise Bilder auf dem Bildschirm malen oder Websites aufrufen. Module bieten außerdem Funktionen für allgemeine Aufgaben, sodass du hierfür nicht deine eigenen Lösungen entwickeln musst.

Schließlich wirst du auch *pip* kennenlernen. *pip* ist ein extrem nützliches Programm zum Installieren neuer Module. Du wirst es im Zusammenhang mit dem Flask-Modul benutzen, um eine einfache Website zu erstellen, die eine Verbindung mit Minecraft herstellt und die Position des Spielers anzeigt.

Dateien verwenden

Wenn du mit Computern arbeitest, benutzt du Dateien ständig. Jedes Mal, wenn du ein Textdokument schreibst oder Python-Code entwirfst und diesen speicherst, arbeitest du mit Dateien. Texte, Bilder, Videos Musik – all dies sind Dateien! Sogar dieses Buch wurde, während ich es schrieb, als Textdatei gespeichert. Die Dateiverwaltungsfunktionen von Python sind einfach zu erlernen und erlauben dir das Erstellen, Speichern und Auslesen von Dateien. So kannst du allerlei coole Sachen in Python machen. Beginnen wir mit den Grundlagen. Dazu wollen wir uns als Erstes ansehen, wie eine Textdatei in Python gelesen und geschrieben wird.

Eine Datei öffnen

Das Öffnen einer Datei ist der erste Schritt beim Umgang mit Dateien. Zum Öffnen von Dateien in Python verwendest du die Funktion `open()`. Diese nimmt zwei Argumente entgegen: den Speicherort der Datei und die Berechtigungen. Der *Speicherort* ist das Verzeichnis, in dem die Datei auf deinem Computer gespeichert ist. Dieses übergibst du als String an die Funktion `open()`. Mit den *Dateiberechtigungen* steuerst du, ob Python die Datei lesen oder ändern darf.

Um eine Textdatei namens *secretFile.txt* zu öffnen oder zu erstellen, würdest du in Python das Argument `"secretFile.txt"` benutzen:

```
secretFile = open("secretFile.txt", "w")
```

Das zweite Argument `"w"` ist das Berechtigungsargument. Es gibt an, was das Programm mit der Datei tun darf. In diesem Fall bedeutet `w`, dass das Programm Daten in die Datei *secretFile.txt* schreiben darf.

Ruft ein Programm die Funktion `open()` mit einem Dateinamen auf, dann überprüft Python zunächst, ob eine Datei dieses Namens bereits vorhanden ist. Ist dies der Fall, dann verwendet Python den Inhalt der betref-

fenden Datei im Programm. Gibt es die Datei hingegen noch nicht, dann erstellt Python sie und gibt ihr den Namen, den du angegeben hast.

Falls du kein *Verzeichnis* zu deinem Dateinamen angibst, sucht Python in dem Verzeichnis danach, in dem sich auch das Programm befindet. (Beachte übrigens, dass die Bezeichnungen »Verzeichnis« und »Ordner« synonym verwendet werden.) Befindet sich die Datei in einem anderen Verzeichnis, dann musst du es im Argument angeben. Wenn also beispielsweise die Datei *secretFile.txt* im Verzeichnis *secrets* abgelegt ist, muss das erste Argument `"/secrets/secretFile.txt"` lauten:

```
secretFile = open("/secrets/secretFile.txt", "w")
```

Gibst du nun ein Verzeichnis als Argument an, das nicht existiert oder in dem die betreffende Datei nicht vorhanden ist, dann wird eine Fehlermeldung angezeigt.

Für das Berechtigungsargument gibt es vier verschiedene Optionen:

- **w** Bedeutung: *write only* (Leseschutz). Diese Berechtigung erlaubt es dem Programm, neue Daten in die Datei zu schreiben und dort bereits vorhandene Inhalte zu überschreiben. Allerdings kann das Programm den Dateiinhalt nicht auslesen. Wenn eine Datei mit dem von dir als Argument angegebenen Namen nicht vorhanden ist, erstellt das Programm sie neu.
- **r** Bedeutung: *read only* (Schreibschutz). Mit dieser Berechtigung kann das Programm den Inhalt der Datei lesen, darf ihn jedoch nicht ändern. Zum Erstellen einer neuen Datei kann diese Berechtigung nicht verwendet werden.
- **r+** Bedeutung: *read and write* (Lese- und Schreibberechtigung). Diese Berechtigung gestattet dem Programm das Lesen und Ändern des Dateiinhalts. Zudem kann das Programm Dateiinhalte auch überschreiben. Ist allerdings die Datei nicht vorhanden, dann wird keine neue Datei erstellt, sondern eine Fehlermeldung ausgegeben.
- **a** Bedeutung: *append* (Anfügen). Erteilst du eine Anfügeberechtigung, dann kann das Programm neue Daten nur am Ende der Datei anfügen. Vorhandene Inhalte bleiben intakt. Zudem darf das Programm den Inhalt der Datei auch nicht auslesen. Diese Berechtigung kann zum Erstellen einer neuen Datei verwendet werden.

Für den Einsatz der verschiedenen Berechtigungen gibt es jeweils unterschiedliche Szenarios. Angenommen, du möchtest die Route zu einer ergiebigen Diamantenmine notieren, die du gefunden hast, und diese

Wegbeschreibung dann in Minecraft laden, ohne sie versehentlich zu ändern. In diesem Fall würdest du die Schreibschutzberechtigung verwenden, um dafür zu sorgen, dass dieser Teil nicht geändert kann. Alternativ würdest du, wenn jemand Daten an eine Datei anhängen, die ansonsten darin gespeicherten Daten aber nicht lesen können soll, die Anfügeberechtigung verwenden. Dies wäre etwa der Fall, wenn deinen Freunden, erlauben willst, Anmerkungen zu einem freigegebenen Reisetagebuch zu machen, ohne dass sie herausfinden können sollen, wo du deinen geheimen Schatz versteckt hast.

Im nächsten Abschnitt erfährst du, wie du Daten in eine geöffnete Datei schreibst und diese Datei dann wieder schließt, um die Daten zu einem späteren Zeitpunkt verwenden zu können.

In eine Datei schreiben und sie speichern

Die Funktion `write()` schreibt Daten in eine Datei, die zuvor vom Programm geöffnet wurde. Dies ist im Übrigen der Hauptgrund für das Arbeiten mit Dateien, denn so kannst du alle möglichen Daten speichern. Die Daten, die du in der Datei speichern möchtest, übergibst du als Argument an die Funktion `write()`.

Öffnen wir einmal eine Datei und schreiben wir dort einen einfachen String hinein:

```
secretFile = open("secretFile.txt", "w")
secretFile.write("This is a secret file. Shhh! Don't tell anyone.")
secretFile.close() ❶
```

Zunächst musst du die Datei mit der Funktion `open()` öffnen. Danach verwendest du die Punktnotation, um die Funktion `write()` aufzurufen, und schreibst deinen String in die Datei `secretFile.txt`. Abschließend musst du noch die Funktion `close()` ❶ aufrufen, denn diese führt den eigentlichen Speichervorgang durch und schließt die Datei. Du darfst diese Funktion nicht vergessen, denn sonst werden die Daten nicht in der Datei gespeichert!

Führe das Programm aus, und öffne `secretFile.txt` in einem Texteditor, um zu prüfen, ob deine Geheimbotschaft gespeichert wurde. Probiere dann einmal, durch Ändern des Strings etwas anderes in die Datei zu schreiben, und führe das Programm dann erneut aus. Was geschieht? Im Zweifelsfall sollte die alte Botschaft durch die neue ersetzt worden sein!

Ändere die Botschaft noch einmal, und übergib dann im Funktionsaufruf "a" anstelle von "w". Was passiert jetzt? Nicht schlecht, oder?

Datei lesen

Die Funktion `read()` liest den gesamten Inhalt einer Datei aus, die von einem Programm geöffnet wurde. Dies ist vor allem dann praktisch, wenn du Daten in deinem Programm verwenden, Daten ändern und dann wieder an die Datei senden oder Daten zur einfacheren Betrachtung ausgeben möchtest. In jedem dieser Fälle verbindest du die `read()`-Funktion zum Lesen der Dateien.

Möchtest du eine Datei lesen, dann musst du sie zunächst öffnen und darfst, wenn du fertig bist, auch nicht vergessen, sie wieder zu schließen. Du musst dir diese Abfolge bei der Arbeit mit Dateien zur Gewohnheit machen, um Fehler in deinen Programmen zu vermeiden!

Wir wollen einmal eine Datei lesen und ihren Inhalt dann ausgeben. Das folgende Programm *showSecretFile.py* gibt den Inhalt einer Datei mit den Funktionen `read()` und `print()` aus:

```
secretFile = open("secretFile.txt", "r")

print(secretFile.read())
secretFile.close()
```

*showSecret
File.py*

Zunächst öffnen wir die Datei und übergeben dann "r" als Berechtigungsargument, damit unser Programm Inhalte aus der Datei lesen kann. Alternativ könnte man auch "r+" angeben, aber da wir in diesem Fall gar nichts in die Datei schreiben wollen, ist "r" wohl die geeignetste Berechtigung. Um den Inhalt von *secretFile.txt* auszugeben, übergeben wir `secretFile.read()` an eine `print`-Anweisung. Und auch dann, wenn wir keine Daten in die Datei geschrieben haben, sollten wir sie am Ende in jedem Fall mit der `close()`-Funktion schließen.

Führe das Programm aus und warte ab, was passiert. Der Inhalt von *secretFile.txt* sollte auf dem Bildschirm ausgegeben werden. Jetzt kannst du die Datei lesen, ohne sie in einem Texteditor öffnen zu müssen!

Dateien zeilenweise lesen

Angenommen, du hast ein langes Textdokument, das du jedoch nur teilweise betrachten möchtest. Für solche Fälle gibt es in Python die sehr praktische Funktion `readline()`. Anders als `read()`, das den gesamten Inhalt einer Datei abrufen, liest `readline()` die Informationen Zeile für Zeile nacheinander ein.

Um die Funktion `readline()` auszuprobieren, musst du deine Datei *secretFile.txt* zunächst mit gaaanz viel Text erweitern. Hierzu kannst du entweder einen Texteditor verwenden, oder du setzt deine Python-Kenntnisse kreativ ein, um die Datei mit Daten zu füllen. Falls du mit Python in die Datei schreibst, musst du immer dort, wo ein Zeilenwechsel eingefügt werden soll, `\n` in deine Strings einsetzen. Schreibst du beispielsweise `"Cool\nDance\nParty"` in eine Datei, dann setzt Python `"Cool"` in die erste, `"Dance"` in die zweite und `"Party"` in die letzte Zeile:

```
Cool
Dance
Party
```

Nachdem du neuen Text zu deiner Datei *secretFile.txt* hinzugefügt hast, speicherst du den folgenden Code in einer Python-Datei mit dem Namen *showSecretLines.py* in einem neuen Ordner namens *files* ab:

*showSecret
Lines.py*

```
secretFile = open("secretFile.txt", "r")

print(secretFile.readline())
print(secretFile.readline())
print(secretFile.readline())

secretFile.close()
```

Erneut musst du zunächst *secretFile.txt* öffnen, um mit der `readline()`-Funktion daraus lesen zu können. Da dein Programm *showSecretLines.py* Daten aus der Datei lesen soll, musst du erneut `r` (oder `r+`) übergeben. Die nächsten drei `print`-Anweisungen sollen dann die ersten drei Zeilen aus *secretFile.txt* ausgeben. Zum Schluss schließt du die Datei wieder mit `close()`:

Die Funktion `readline()` liest nun die erste Zeile deiner Datei aus. Bei jedem Aufruf von `readline()` wird dann automatisch die nächste Zeile ausgelesen. Diese Funktion ist sehr praktisch, um mehrere Zeilen vom Dateianfang an anzugeben.

Hinweis

Die Funktion `readline()` wandelt die Datei in eine Liste mit Strings um, in der jedes Listenelement eine einzelne Zeile darstellt. Wenn du eine Zeile ausgeben möchtest, die sich irgendwo mitten in einem Textdokument befindet, dann könntest du beispielsweise eine Schleife entwerfen, um einen bestimmten String in der Liste zu suchen und auszugeben.

Mission 64: To-Do-Liste

Manchmal hast du vielleicht einfach nicht genug Zeit, um Minecraft zu spielen. Dann baust du komplexe Strukturen in vielen kurzen Sessions über mehrere Tage hinweg auf. Wenn du Programme zum Öffnen von Türen oder zum Teleportieren des Spielers an einen anderen Ort hinzufügst, werden deine Bauten immer komplexer, und es kann oft sehr lange dauern, bis sie fertiggestellt sind. Wenn du dich jedoch über mehrere Tage hinweg mit einem Projekt beschäftigst, kann es passieren, dass du nicht mehr weißt, woran du gerade gearbeitet hast und welcher Schritt nun folgen müsste. Zumindest mir passiert so etwas häufig. Zum Glück kannst du dir ein Programm erstellen, das dir diesbezüglich auf die Sprünge hilft.

Mit den Programmen in dieser Mission erstellst du eine To-Do-Liste und zeigst diese im Minecraft-Chat an. Du kannst mit so einem Programm deine Ziele in Minecraft im Blick behalten und jedes Mal, wenn du das Spiel unterbrechen musst, später dort weitermachen, wo du aufgehört hast.

Um die To-Do-Liste zu erstellen, schreibst du zwei separate Programme: eines zum Schreiben und das andere zum Anzeigen der Liste. Beginnen wollen wir mit dem Programm, mit dem du die Liste verfasst.

Teil 1: To-Do-Liste schreiben

Zunächst benötigen wir ein Programm, mit dem wir die Elemente in der To-Do-Liste erstellen. Listing 11.1 vermittelt einen Ausgangspunkt. Hier findest du eine `while`-Schleife und die `input()`-Funktion, mit der einzelne Punkte zur Liste hinzugefügt werden. Kopiere das Listing in eine neue IDLE-Datei, und speichere sie als `inputToDoList.py` im Ordner `files`.

input
ToDoList.py

```

todoFile = ❶

todoList = "" ❷

todoItem = input("Enter a to-do list item: ") ❸

while todoItem != "exit": ❹
    todoList = todoList + todoItem + "\n" ❺
    todoItem = input("Enter a to-do list item: ")

# To-Do-Liste in die Datei schreiben ❻
# Datei schließen ❼

```

Listing 11.1: Gerüst für das Programm zum Eintragen von Elementen in deine To-Do-Liste

Das Programm erstellt einen Leerstring namens `todoList` ❷, in dem alle Elemente für deine To-Do-Liste, die du eingibst, gespeichert werden. Mit der `input()`-Funktion fordert das Programm dich dann dazu auf, eine Aufgabe in deine To-Do-Liste einzugeben ❸. Daraufhin überprüft die `while`-Schleife, ob die Eingabe ungleich "exit" ist ❹. Ist dies der Fall, dann fügt das Programm die Aufgabe deiner To-Do-Liste hinzu und beendet den Eintrag mit einem Zeilenwechsel ("\n") ❺. Andernfalls – wenn du also "exit" eingibst – endet die Ausführung der Schleife, und du kannst nachfolgend keine weiteren Elemente zur To-Do-Liste hinzufügen.

Deine Aufgabe: Vervollständige das Programm. Hierzu musst du den Code schreiben, der die Datei öffnet, `todoList` in die Datei schreibt und die Datei dann wieder schließt. Mit der `open()`-Funktion öffnest du die Datei zu Beginn des Programms ❶. Das Öffnen sollte mit Schreibberechtigungen erfolgen. Nenne die Datei, die von der Funktion geöffnet wird, `todoList.txt`. Sofern die Datei noch nicht im Verzeichnis vorhanden ist, wird sie vom Programm erstellt.

Am Ende des Programms schreibst du den Inhalt der To-Do-Liste in die Datei, um ihn später wieder abrufen zu können. Mit der `write()`-Funktion schreibst du die Variable `todoList` nach `todoFile` ❻. Nachdem die Inhalte in die Datei geschrieben wurden, darfst du nicht vergessen, sie mit der Funktion `close()` in der letzten Zeile zu schließen ❼.

Bild 11.1 zeigt, wie ich mit diesem Programm eine To-Do-Liste verfasse. Zum Abschluss gebe ich `exit` ein.

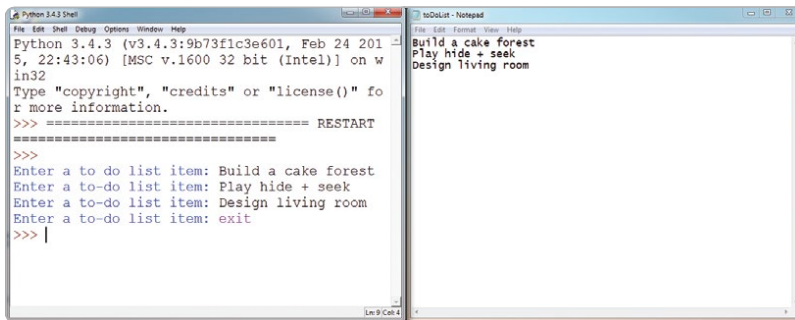


Bild 11.1: Was ich alles noch erledigen muss: einen Wald aus Kuchen bauen und ein bisschen Verstecken spielen

Teil 2: To-Do-Liste anzeigen

Nachdem wir oben ein Programm entworfen haben, mit dem eine To-Do-Liste in eine Datei geschrieben wird, brauchen wir jetzt etwas, um diese Liste Zeile für Zeile im Minecraft-Chat anzuzeigen. Das Gerüst für ein solches Programm findest du in Listing 11.2. Kopiere den Code in eine neue Datei und speichere ihn als *outputToDoList.py* im Ordner *files*.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
```

```
toDoList = ❶
```

```
for line in toDoList:
    # Zeile im Chat ausgeben ❷
```

output
ToDoList.py

Listing 11.2: Programm zur Ausgabe einer To-Do-Liste in den Minecraft-Chat

Listing 11.2 verwendet eine *for*-Schleife, um den Inhalt der Datei *toDoList.txt* zeilenweise in den Minecraft-Chat auszugeben. So wie das Programm dort steht, ist es noch unvollständig. Um es fertigzustellen, fügst du die *open()*-Funktion zum Öffnen der Datei *toDoList.txt* hinzu, die du mit *inputToDoList.py* erstellt hast ❶. Achte darauf, dass die Datei mit Leseberechtigungen geöffnet wird. Nach dem Öffnen der Datei fügst du in der *for*-Schleife Code zum Ausgeben des Strings, der in der Variablen *line* gespeichert ist, im Minecraft-Chat hinzu ❷. Die *for*-Schleife ersetzt die Funktion *readline()* und stellt in der Variablen *line* die einzelnen Zeilen zur Verfügung.

Bild 11.2 zeigt meine To-Do-Liste im Chat.



Bild 11.2: Wenn ich jetzt nach einem langen Tag Arbeit meinen Computer starte, um mich mit Minecraft zu entspannen, dann sehe ich sofort, was noch zu erledigen ist.

Module verwenden

Module sind Funktionssammlungen, die du in Python importieren kannst, um nicht jede einzelne Funktion in deinem eigenen Programm neu schreiben zu müssen. Normalerweise hat ein Modul immer einen bestimmten Zweck, beispielsweise das Durchführen wissenschaftlicher Berechnungen oder das Ausführen von Spielen. Für Python gibt es eine sehr große Zahl von Modulen. Du wirst möglicherweise überrascht sein zu hören, dass du in diesem Buch bereits von Anfang an Module eingesetzt hast. Die Minecraft-Python-API ist ein solches Modul: Jedes Mal, wenn du `from mcpi.minecraft import Minecraft` geschrieben hast, hast du ein Modul verwendet. Mit dem Minecraft-Python-API-Modul kannst du eine Verbindung deiner Python-Programme mit Minecraft herstellen. Da das Modul bereits fertig geschriebene Funktionen enthält, kannst du diese in deinen Programmen verwenden, ohne den Code selbst schreiben zu müssen.

Im Lieferumfang von Python sind eine Menge Module enthalten, die du in deinen Programmen verwenden kannst. Die Kombination aus diesen Modulen und allem, was du bislang in diesem Buch über Python gelernt hast, wird als *Python-Standardbibliothek* bezeichnet. Du kannst außerdem auch Module installieren, die nicht zur Standardbibliothek gehören; das machen wir in »Weitere Module mit pip installieren« auf Seite 327.

In diesem Abschnitt wirst du alle Möglichkeiten kennenlernen, deine Programme für die Verwendung von Modulen einzurichten. Als Beispiel werden wir das `pickle`-Modul verwenden. Dieses bietet erweiterte Möglichkeiten zum Speichern in und Laden aus Dateien, die über das einfache Schreiben und Lesen von Daten hinausgehen. Wir wollen uns das `pickle`-Modul einmal genauer ansehen.

Das pickle-Modul

Das `pickle`-Modul ist sehr nützlich, wenn es darum geht, komplexe Daten in eine Datei zu schreiben. Das Speichern und Abrufen von Wörterbüchern und mehrdimensionalen Listen ist mithilfe der Standardfunktionen, die wir bislang in diesem Kapitel verwendet haben, kein Honigschlecken. Das `pickle`-Modul hingegen bietet hierfür verschiedene praktische Optionen.

Wenn du das `pickle`-Modul verwendest, ersparst du dir viele Stunden, die du sonst damit verbringen würdest, eigene Lösungen zum Speichern komplexer Daten zu schreiben und zu debuggen. Du kannst das `pickle`-Modul natürlich auch für einfache Daten verwenden. Du kannst damit beispielsweise Zahlen speichern, ohne sie zuvor in Strings konvertieren zu müssen, was für das normale Speichern in Dateien erforderlich ist.

Anders formuliert: Mit dem `pickle`-Modul speicherst du den Wert einer Variablen in einer Datei und kannst diesen dann direkt – also ohne weitere Konvertierung – in ein anderes Programm einlesen. Dabei bleibt der Datentyp beim Speichern unangetastet – egal ob es sich um einen String, einen Integer, einen Float oder einen booleschen Wert handelt.

Im nächsten Abschnitt erfährst du, wie Module mit `pickle` importiert werden. Danach werden wir `pickle` zum Speichern komplexer Daten einsetzen. Wir werden nämlich ein ganzes Minecraft-Gebäude in einer Datei ablegen!

pickle importieren

Um die Funktionen eines Moduls verwenden zu können, musst du es mit dem Schlüsselwort `import` importieren. Tatsächlich haben wir dieses Schlüsselwort bereits für den Import von Modulen verwendet, beispielsweise des `time`-Moduls oder verschiedener Funktionen der Python-Minecraft-API.

Nach dem Import des Moduls in dein Programm kannst du seine Funktionen mithilfe der Punktnotation aufrufen: Du gibst den Modulnamen, einen Punkt und die zu verwendende Funktion an. Importieren wir also das `pickle`-Modul und verwenden wir einige seiner Funktionen:

```
import pickle ❶

locations = {'John': 'Forest', 'Phillipa': 'Mountains', 'Pete': 'City'}

secretFile= open("secretFile.txt", "wb") ❷
pickle.dump(locations, secretFile) ❸
```

Das `pickle`-Modul importieren wir bei ❶. Danach öffnen wir `secretFile.txt` mit der speziellen Dateiberechtigung `"wb"` ❷. Wenn du eine Datei mit `pickle` öffnest, musst du `b` an die Dateiberechtigung anhängen. In diesem Fall schreibt `"wb"` die Daten in die Datei, wobei es ein spezielles Format verwendet, das vom `pickle`-Modul benötigt wird.

Bei ❸ schreibt die `dump()`-Funktion in die Datei. Diese Funktion aus dem `pickle`-Modul schreibt eine Variable in eine Datei. Sie nimmt zwei Argumente entgegen: die Daten, die in die Datei geschrieben werden sollen, und die offene Datei, in die der Schreibvorgang erfolgt. In diesem Beispiel werden die Positionen von Geheimagenten in einem Wörterbuch namens `locations` abgelegt, das dann in einer Datei namens `secretFile` gespeichert wird. Da `dump()` Bestandteil des `pickle`-Moduls ist, musst du die Punktnotation verwenden und sowohl den Modul- als auch den Funktionsnamen angeben (`pickle.dump()`). Anders als bei den Standardfunktionen für Dateien in Python speichert die `dump()`-Funktion die Daten automatisch in der Datei, d. h., du musst die Datei nicht explizit mit der `close()`-Funktion schließen.

Mit dem `pickle`-Modul kannst du auch gespeicherte Daten lesen. Zum Lesen von Dateiinhalten verwendest du die `pickle`-Funktion `load()`. Diese Funktion nimmt ein Argument entgegen, nämlich die zu ladende Datei, und gibt deren Inhalt zurück. Das folgende Beispiel lädt das Wörterbuch `locations`, das wir weiter oben gespeichert haben. Füge den folgenden Code zum Programm hinzu:

```
import pickle

secretFile= open("secretFile.txt", "rb") ❶
locations = pickle.load(secretFile)
```

Zunächst öffnen wir die Datei mit der Berechtigung "rb" ❶, um dem Programm das Lesen von Daten in dem speziellen von `pickle` verwendeten Format zu erlauben. Danach laden wir das Wörterbuch.

Ist das Wörterbuch einmal geladen, dann kannst du es wie jedes andere Wörterbuch verwenden. Du kannst beispielsweise den Wert eines Schlüssels abrufen. Füge zu diesem Zweck einfach den folgenden Code nach der `pickle.load()`-Funktion ein:

```
print(locations['Phillipa'])
```

Hiermit wird 'Mountains' ausgegeben (das ist der Wert des Schlüssels 'Phillipa'). Dies ist möglich, weil das Wörterbuch in der Datei nach dem Laden in das Programm mit `pickle` unverändert ist: Es ist nach wie vor ein Wörterbuch, d. h., wir können auf seine Schlüssel und Werte zugreifen und es wie jedes andere Python-Wörterbuch verwenden. Genau das-selbe könntest du auch mit einer Liste oder einer Variablen machen.

Eine Funktion mit der from-Klausel importieren

Wenn du ein Modul importierst, hast du Zugriff auf alle Funktionen darin. Manchmal benötigst du allerdings nur eine einzige Funktion aus einem Modul. In solchen Fällen kannst du die `from`-Klausel beim Importieren des Moduls angeben. Mit dieser Klausel kannst du auf die Funktion zugreifen, ohne bei jedem Aufruf die vollständige Punktnotation mit dem Modulnamen angeben zu müssen. Du könntest dann einfach `funktionsname()` statt `modulname.funktionsname()` schreiben.

Nehmen wir einmal an, du benötigst aus dem `pickle`-Modul nur die Funktion `dump()` und keine der anderen Funktionen. Hierzu änderst du den Code von oben wie folgt ab:

```
from pickle import dump ❶

locations = {'John': 'Forest', 'Phillipa': 'Mountains', 'Pete': 'City'}

secretFile= open("secretFile", "wb")
dump(locations, secretFile) ❷
```

In der ersten Zeile wird mit der `from`-Klausel nur die `dump()`-Funktion aus dem `pickle`-Modul importiert ❶. Der Aufruf von `dump()` erfolgt dann in der letzten Zeile ❷. Beachte, dass hier keine Punktnotation eingesetzt

wird: Der Aufruf erfolgt einfach über den Funktionsnamen, d. h. ohne Angabe des Modulnamens.

Du kannst mit `from` auch mehrere Funktionen aus einem Modul importieren. Hierzu musst du nichts weiter tun, als die Funktionsnamen mit Kommata zu trennen. Wenn du beispielsweise die Funktionen `dump()` und `load()` aus `pickle` in dieselbe Datei laden möchtest, kannst du beide wie folgt importieren:

```
from pickle import dump, load ❶
locations = {'John': 'Forest', 'Phillipa': 'Mountains', 'Pete': 'City'}

secretFile= open("secretFile", "wb")
dump(locations, secretFile) ❷

locations = load(secretFile) ❸
print(locations['Phillipa'])
```

In der ersten Zeile wird die `from`-Klausel mit Kommata verwendet, um die Funktionen `dump()` und `load()` zu importieren ❶. Auf diese Weise kannst du diese Funktionen später im Programm verwenden, ohne den Modulnamen und einen Punkt angeben zu müssen (siehe ❷ und ❸).

Alle Funktionen mit `*` importieren

Natürlich kannst du auch alle Funktionen eines Moduls so importieren, dass du nicht jedes Mal den Modulnamen und die Punktnotation angeben musst. Hierzu gibst du ein Sternchen (`*`) am Ende der `import`-Anweisung an:

```
from pickle import * ❶
locations = {'John': 'Forest', 'Phillipa': 'Mountains', 'Pete': 'City'}

secretFile= open("secretFile", "wb")
dump(locations, secretFile) ❷

locations = load(secretFile) ❸
print(locations['Phillipa'])
```

Da mit diesem Code alle Funktionen im Modul durch Angabe des Sternchens ❶ importiert wurden, müssen wir beim Aufruf der Funktionen `dump()` ❷ und `load()` ❸ keine Punktnotation verwenden.

Die Option mit `*` ist sehr praktisch, birgt aber auch ein Risiko! Wenn du nämlich mit mehreren Modulen arbeitest, besteht die Gefahr, dass derselbe Funktionsname in mehreren Modulen gleichzeitig auftritt. In diesem Fall gerät Python durcheinander, was zu Fehlern führen kann. Wenn du also mit zahlreichen Modulen arbeitest, solltest du die Option `*` vielleicht besser nicht benutzen, sondern stattdessen wirklich nur diejenigen Funktionen importieren, die du tatsächlich zu verwenden beabsichtigst.

Einen Spitznamen für ein Modul erstellen

Manchmal ist es praktisch, ein Modul umzubenennen – etwa weil sein Name zu lang ist und du in deinem Programm gerne etwas Kürzeres verwenden würdest. Oder du möchtest den Modulnamen ändern, um ihn dir leichter merken zu können. Es kann auch notwendig werden, den Modulnamen zu ändern, weil ein anderes Modul denselben Namen hat und du Konflikte vermeiden möchtest.

Mit der `as`-Klausel für die `import`-Anweisung kannst du einem Modul eine Art Spitzname geben. Dieser wird gemeinhin als *Alias* bezeichnet. Der folgende Code beispielsweise importiert das `pickle`-Modul und benennt es in `p` um:

```
import pickle as p
```

Nun kannst du jedes Mal, wenn du eine Funktion aus diesem Modul verwenden möchtest, `p` anstelle von `pickle` schreiben. Hier ist ein Beispiel:

```
p.dump(locations, secretFile)
```

Wie du siehst, verwende ich `p.dump()` anstelle von `pickle.dump()`. Das spart mir Zeit, weil ich `pickle` jetzt nicht ausschreiben muss!

Mission 65: Ein Gebäude speichern

Das Bauen von Dingen gehört zu meinen Lieblingsbeschäftigungen in Minecraft. Ich verbringe Stunden damit, Häuser, Schlösser, ganze Dörfer und viele andere Dinge zu bauen. Wenn ich aber in einen anderen Teil der Karte wechsele oder eine neue Welt erstelle, muss ich meine Schöpfungen zurücklassen. Bestimmt hast auch du schon einige großartige Bauten aufgeben müssen.

Wäre es nicht cool, wenn du deine Gebäude speichern könntest, um sie bei deinem Umzug in eine andere Welt mitzunehmen? Und jetzt kommt's: Mit `pickle` und der Python-API geht das!

Wir werden in dieser Mission zwei Programme entwickeln, mit denen du Gebäude speichern und dann in eine andere Minecraft-Welt laden kannst. Ein Programm dient zur Speicherung, das andere ist zum Laden da. Beide Programme basieren übrigens auf *duplicateArea.py* aus Kapitel 10 (Seite 225).

Teil 1: Gebäude speichern

Das erste Programm speichert das Gebäude in eine Datei. Listing 11.3 enthält den hierfür erforderlichen Code. Kopiere es in eine neue IDLE-Datei und speichere diese als *saveStructure.py* im Ordner *files*.

saveStructure
.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import pickle

def sortPair(val1, val2):
    if val1 > val2:
        return val2, val1
    else:
        return val1, val2

def copyStructure(x1, y1, z1, x2, y2, z2): ❶
    x1, x2 = sortPair(x1, x2)
    y1, y2 = sortPair(y1, y2)
    z1, z2 = sortPair(z1, z2)

    width = x2 - x1
    height = y2 - y1
    length = z2 - z1

    structure = []

    print("Please wait..." )

    # Struktur kopieren
    for row in range(height):
        structure.append([])
        for column in range(width):
            structure[row].append([])
```

```

    for depth in range(length):
        block = mc.getBlock(x1 + column, y1 + row, z1 + ←
        depth) ❷
        structure[row][column].append(block)

    return structure

# Position der ersten Ecke abrufen ❸
input("Move to the first position and press ENTER in this window")
pos1 = mc.player.getTilePos()

x1 = pos1.x
y1 = pos1.y
z1 = pos1.z

# Position der zweiten Ecke abrufen ❹
input("Move to the opposite corner and press ENTER in this window")
pos2 = mc.player.getTilePos()

x2 = pos2.x
y2 = pos2.y
z2 = pos2.z

structure = copyStructure( x1, y1, z1, x2, y2, z2) ❺

# Struktur in einer Datei speichern ❻

```

Listing 11.3: Codegerüst zum Speichern eines Gebäudes in einer Datei

Die Funktion `copyStructure()` kopiert einen Bereich im Spiel als Satz dreidimensionaler Listen ❶. Sie nimmt zwei Koordinatensätze als Argumente entgegen. Ich habe im Vergleich zu *duplicateArea.py* eine geringfügige Änderung in der `copyStructure()`-Funktion vorgenommen, indem ich die Funktion `getBlockWithData()` anstelle der `getBlock()`-Funktion verwendet habe ❷. Die Funktion `getBlockWithData()` ruft nicht nur die Typ-ID eines Blocks ab, der sich an bestimmten Koordinaten befindet, sondern auch den Blockzustand. Dies ist praktisch für Blöcke wie Treppen, deren Richtung im Blockzustand gespeichert ist. Wenn die Struktur kopiert wird, dann werden Treppen und andere Blöcke mit einer bestimmten Ausrichtung korrekt aufgebaut.

Ich habe oben einige nützliche Zeilen in den Code integriert, damit du die Koordinaten eines zu speichernden Gebäudes anhand der Spielerposition festlegen kannst. Wenn du das Programm ausführst, wirst du zunächst aufgefordert, zur ersten Ecke der Struktur zu gehen und dann **Enter** in der Python-Shell zu drücken ❸. Das Programm verwendet die Spielerposition zum Abrufen der ersten Gebäudekoordinaten. Danach

wirst du aufgefordert, dich zur gegenüberliegenden Ecke der Struktur zu begeben und den Vorgang zu wiederholen ④. Du kannst dich also einfach dort hinstellen, wo der Kopiervorgang beginnen soll, statt Koordinaten zu notieren oder sie fest in dein Programm einzukodieren.

Die Werte dieser Koordinatenvariablen werden bei ⑤ an die Funktion `copyStructure()` übergeben. Der Rückgabewert wird in einer Variablen namens `structure` gespeichert.

Zur Vervollständigung des Codes musst du eine neue Datei mit `pickle` öffnen. Nenne diese neue Datei `"pickleFile"`. Danach schreibst du den Code zum Speichern des Gebäudes in der Datei. Hierzu schreibst du mithilfe des `pickle`-Moduls den Wert von `structure` in eine Datei ⑥.

Bild 11.3 zeigt einen Turm, den ich in meiner Minecraft-Welt gebaut habe.



Bild 11.3: Diesen Turm möchte ich kopieren.

Um diesen Turm mit `saveStructure.py` zu kopieren, begeben wir uns zu einer Ecke und drücken **Enter** in IDLE (Bild 11.4).

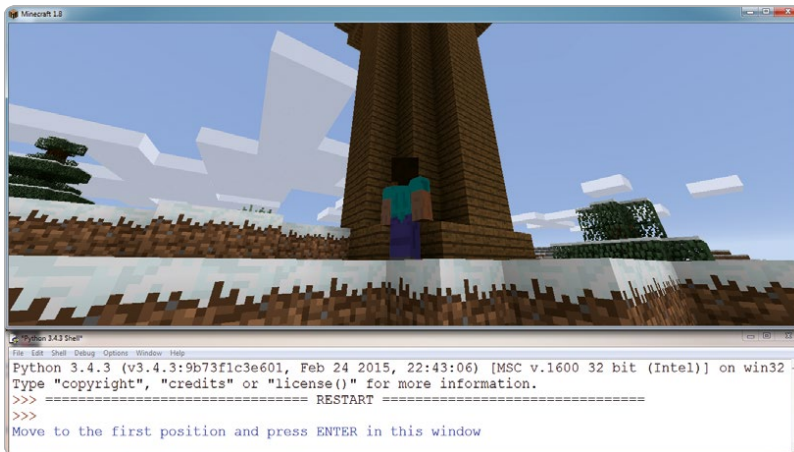


Bild 11.4: Meine Position an einer Ecke des Turms

Dann fliege ich zur gegenüberliegenden Ecke des Turms und drücke erneut **Enter** in IDLE (Bild 11.5).



Bild 11.5: Ich bin zur gegenüberliegenden Ecke des Turms geflogen.

Befolge die oben beschriebenen Schritte zur Verwendung von `saveStructure.py`, um eines deiner eigenen Gebäude zu speichern. Nun wollen wir die zweite Hälfte des Vorgangs durchführen und unsere gespeicherten Gebäude in das Spiel laden.

Teil 2: Gebäude laden

Das zweite Programm soll das Gebäude aus der Datei *pickleFile*, die wir mit *saveStructure.py* erstellt haben, in das Spiel laden. Listing 11.4 enthält Code aus dem Programm *duplicateArea.py* (Seite 225), das ein in Listen gespeichertes Gebäude platziert. Kopiere den Code in eine neue Datei in IDLE und speichere ihn als *loadStructure.py* im Ordner *files*.

loadStructure
.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import pickle

def buildStructure(x, y, z, structure): ❶
    xStart = x
    zStart = z
    for row in structure:
        for column in row:
            for block in column:
                mc.setBlock(x, y, z, block.id, block.data)
                z += 1
            x += 1
        z = zStart
    y += 1
    x = xStart

# Gebäudedatei öffnen und laden
structure = ❷

pos = mc.player.getTilePos() ❸
x = pos.x
y = pos.y
z = pos.z
buildStructure(x, y, z, structure) ❹
```

Listing 11.4: Wenn das Programm vollständig ist, kannst du damit ein Gebäude aus einer Datei erstellen.

Die `buildStructure()`-Funktion ❶ tut in diesem Programm die meiste Arbeit. Sie erstellt im Spiel die Struktur mithilfe von vier Argumenten: den Koordinaten `x`, `y` und `z` sowie einer Struktur, die in einer dreidimensionalen Liste gespeichert ist.

Importiere das `pickle`-Modul, damit du die Struktur in das Programm laden kannst, und speichere sie dann in der Variablen `structure` bei ❷. Nun öffnest du mit der Funktion `open()` die Datei *pickleFile*, in der du

die Struktur gespeichert hast. Lade sie dann in die Variable `structure`. Hierzu verwendest du die `load()`-Funktion von `pickle`. Nach dem Laden der Struktur schließt du `pickleFile` mit der `pickle`-Funktion `close()`.

Ebenfalls in Listing 11.4 enthalten ist etwas Code, der die Spielerposition abrufen, da diese als Startpunkt für die Struktur verwendet wird ③.

Nach dem Laden der Struktur und dem Festlegen der Koordinaten übergibst du die Struktur nebst der Position an die `buildStructure()`-Funktion ④. Diese Funktion baut nun die gespeicherte Struktur nach.

Bild 11.6 zeigt das Programm bei der Ausführung. Das Gebäude, das ich weiter oben gespeichert habe, wurde nun in das Spiel geladen und an einer neuen Position wieder aufgebaut. Probiere es selbst einmal aus: Du kannst deine Kreationen nun mitnehmen, wohin du willst!



Bild 11.6: Sieh mal einer an: ein Duplikat meines Turms!

Nehmen wir nun an, du möchtest ein ganzes Dorf mitnehmen, das du errichtet hast. Du könntest natürlich jedes einzelne Gebäude mit `pickle` in einer eigenen Datei speichern, aber das wäre alles andere als praktisch. Das `pickle`-Modul funktioniert zur Speicherung eines einzelnen Gebäudes einwandfrei, aber zum Speichern zahlreicher Gebäude ist es eher ungeeignet. Das ist der Punkt, an dem das `shelve`-Modul ins Spiel kommt. Und das wollen wir uns jetzt einmal genauer ansehen.

Mit dem shelve-Modul viele Daten speichern

Mit dem `pickle`-Modul lässt sich nur ein Datenelement gleichzeitig speichern. In manchen Programmen musst du jedoch mehrere Variablen speichern. Würdest du hierzu das `pickle`-Modul verwenden, dann müsstest du zu diesem Zweck mehrere Dateien erstellen, was in der Handhabung etwas schwierig sein könnte.

Dieses Problem lässt sich mit dem Python-Modul `shelve` lösen. Das Modul kann mehrere Datenelemente in einer einzelnen Datei ablegen. Es funktioniert wie ein Wörterbuch, in dem jeder Datenwert einen Schlüssel hat, mit dessen Hilfe du die Daten speichern und abrufen kannst. Du kannst dir `shelve` wie ein Bücherregal vorstellen: Jedes Regalbrett nimmt unterschiedliche Daten auf.

Datei mit shelve öffnen

Nach dem Importieren des `shelve`-Moduls verwendest du dessen `open()`-Funktion zum Öffnen einer Datei. Ist diese Datei noch nicht vorhanden, wird sie angelegt.

Mit dem folgenden Code öffnest du die Datei *locationsFile.db* und speicherst ihren Inhalt in der Variablen `shelveFile`:

```
import shelve
shelveFile = shelve.open("locationsFile.db")
```

Die `open()`-Funktion nimmt nur ein einziges Argument entgegen, nämlich den Namen der zu öffnenden Datei. Bei der Verwendung des `shelve`-Moduls musst du keine Dateiberechtigungen angeben – es werden automatisch Lese- und Schreibrechte gewährt.

Beim Angeben einer Datei für das `shelve`-Modul musst du die Dateierweiterung *.db* an das Ende des Dateinamens anhängen. Du kannst die Erweiterung *.db* am Ende meiner Datei *locationsFile.db* sehen.

Elemente mit `shelve` hinzufügen, bearbeiten und abrufen

Das `shelve`-Modul funktioniert wie ein Wörterbuch. Um Daten zur Datei hinzuzufügen, gibst du in eckigen Klammern einen Schlüsselnamen zur Aufnahme eines Werts an. Nehmen wir einmal an, eine Geheimagentin namens Beatrice befindet sich in einem Unterseeboot. Nun wollen wir ihren Standort im Wörterbuch `shelveFile` speichern:

```
import shelve
shelveFile = shelve.open("locationsFile.db")
shelveFile['Beatrice'] = 'Submarine'
shelveFile.close()
```

Zunächst öffnen wir die Datei. Dann übergeben wir an das Wörterbuch `shelveFile` den Schlüssel `'Beatrice'` und den Wert `'Submarine'`. Diese Zeile erstellt ein neues Element im Wörterbuch `shelveFile` mit dem Schlüssel `'Beatrice'` und dem Wert `'Submarine'`.

Danach fügen wir mit der `shelve`-Funktion `close()` neue Daten zur Datei hinzu und schließen sie.

Wäre ein Schlüssel dieses Namens bereits in einer solchen Datei vorhanden, dann würde mit diesem Code einfach nur der Wert aktualisiert. Nehmen wir nun an, Beatrice habe ihre Mission beendet und kehrt zum Hauptquartier zurück. Du könntest ihre Position nun wie folgt auf den aktuellen Stand bringen:

```
import shelve
shelveFile = shelve.open('locationsFile.db')
shelveFile['Beatrice'] = 'Headquarters'
shelveFile.close()
```

Jetzt lautet der entsprechende Wert des Schlüssels `'Beatrice'` `'Headquarters'`.

Auch das Abrufen eines Werts aus `shelve` funktioniert wie bei einem Wörterbuch. Du verwendest Schlüssel zum Abrufen bestimmter Werte. Um beispielsweise die Position von Beatrice auszugeben, würden wir folgenden Code verwenden:

```
import shelve
shelveFile = shelve.open('locationsFile.db')
print(shelveFile['Beatrice'])
```

Hiermit wird Beatrice' gegenwärtiger Standort (Headquarters) ausgegeben.

Wie ein ganz normales Wörterbuch kann das `shelve`-Modul beliebige Datentypen speichern: Floats, Strings, boolesche Werte, mehrdimensionale Listen, andere Wörterbücher usw. Das wollen wir einmal ausprobieren, und deswegen lernst du in der nächsten Mission das Speichern und Laden mehrdimensionaler Listen – und kannst auf diese Weise mehrere Strukturen gleichzeitig kopieren!

Mission 66: Sammlung von Strukturen speichern

Die Programme in dieser Mission speichern und laden alle deine gespeicherten Strukturen (Gebäude usw.), wobei sie nur eine einzige Datei verwenden. Auch diese Mission umfasst zwei Programme: eines für das Speichern und das andere für das Laden.

Im Wesentlichen musst du die Programme aus Mission 65 so umstellen, dass sie statt dem `pickle`-Modul das `shelve`-Modul verwenden. Außerdem werden wir ein bisschen Code hinzufügen, der eine Benutzereingabe entgegennimmt. Damit sollen die Benutzer ihre Gebäude benennen. Öffne die Dateien `saveStructure.py` und `loadStructure.py` und speichere sie als `saveCollection.py` bzw. `loadCollection.py`.

Wie in der vorherigen Mission werden wir die Änderungen an diesen Programmen in zwei Teilen vornehmen.

Teil 1: Struktur in einer Sammlung speichern

Ich habe einen Teil des ursprünglichen Programms `saveStructure.py` hier aufgeführt und kommentiert, damit du einfacher erkennen kannst, wo Änderungen erforderlich sind. Hier folgen die erste Zeile und die letzten Zeilen von `saveCollection.py`:

*save
Collection.py*

```
import pickle ❶

-- usw. --

# Struktur benennen
structureName = input("What do you want to call the structure?") ❷

# Struktur in Datei speichern
pickleFile = open("pickleFile", "wb") ❸
pickleFile.dump(structure) ❹
```

Ich habe eine zusätzliche Zeile zur Datei hinzugefügt, um zu erfragen, wie du die Struktur beim Speichern mit `pickle` nennen möchtest ❷. Meine Version des Programms fragt beispielsweise: »What do you want to call the structure?« (»Wie soll die Struktur heißen?«). Ich kann auf diese Frage Antworten wie »House« (»Haus«) oder »Cake Forest« (»Kuchenwald«) geben. Du musst dafür sorgen, dass jede neue Struktur einen anderen eindeutigen Namen erhält. Hat eine Struktur nämlich denselben Namen wie eine andere, dann wird die ältere Struktur von der neuen überschrieben.

Um das Programm für die Verwendung des `shelve`-Moduls anstelle von `pickle` anzupassen, musst du zwei Änderungen vornehmen: Erstens musst du den Modulimport von `pickle` auf `shelve` umstellen ❶ und zweitens musst du die letzten paar Codezeilen so anpassen, dass `shelve` anstelle von `pickle` verwendet wird. Öffne eine Datei namens *structures-File.db* und speichere sie in einer Variablen namens `shelveFile`, wozu du die `shelve.open()`-Funktion verwendest ❸. Danach speicherst du die Variable `structure` in einem `shelve`-Wörterbuch, wobei du die Variable `structureName` für den Namen des Wörterbuchschlüssels verwendest ❹. Dies sieht ungefähr folgendermaßen aus: `shelveFile[structureName] = structure`. Abschließend schließt du `shelveFile` in der letzten Zeile mit `close()`.

Teil 2: Struktur aus einer Sammlung laden

Wenden wir uns nun der Datei *loadCollection.py* zu und überarbeiten wir diese. Ich habe die Mitte der Datei entfernt, um an dieser Stelle etwas Platz zu sparen und die Dinge, die du verändern musst, besser veranschaulichen zu können:

```
import pickle ❶

-- usw. --

structure = pickle.load("pickleFile") ❷
structureName = input("Enter the structure's name") ❸

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z

buildStructure(x, y, z, structureDictionary[structureName]) ❹
```

*load
Collection.py*

Ich habe eine zusätzliche Codezeile hinzugefügt, die den Namen der Struktur erfragt, die wir bauen wollen ③. Ferner habe ich in der letzten Zeile ein bisschen Code ergänzt, um die Struktur aus dem `shelve`-Wörterbuch abzurufen und an die Funktion `buildStructure()` zu übergeben.

An diesem Programm musst du eine Reihe von Änderungen vornehmen. Als Erstes musst du wie in `saveCollection.py` den `import`-Teil von `pickle` in `shelve` ändern ①. Danach lädst du die Datei `shelveFile`, die du in `saveCollection.py` erstellt hast, mit `shelve.open()` ②. Speichere die von `shelve.open()` zurückgegebenen Daten in der Variablen `structureDictionary` ④. Der Code sollte nun wie folgt aussehen: `structureDictionary = shelve.load("shelveFile")`.

Alle Daten für die Strukturen einschließlich ihrer Namen und Blöcke werden in der Datei `structuresFile.db` gespeichert, was nichts anderes bedeutet, als dass du an `loadCollection.py` vor der Ausführung keine Änderungen mehr vornehmen musst. Du musst lediglich den Namen der Struktur eingeben, die bei der Ausführung des Programms verwendet werden soll.

Wir wollen uns das Programm in Aktion ansehen und verwenden dafür eine Struktur aus meiner Minecraft-Welt. Zunächst kopieren wir die Struktur mit `saveCollection.py`. Hierzu fliege ich zu einer Ecke der Struktur und drücke **Enter** (Bild 11.7).

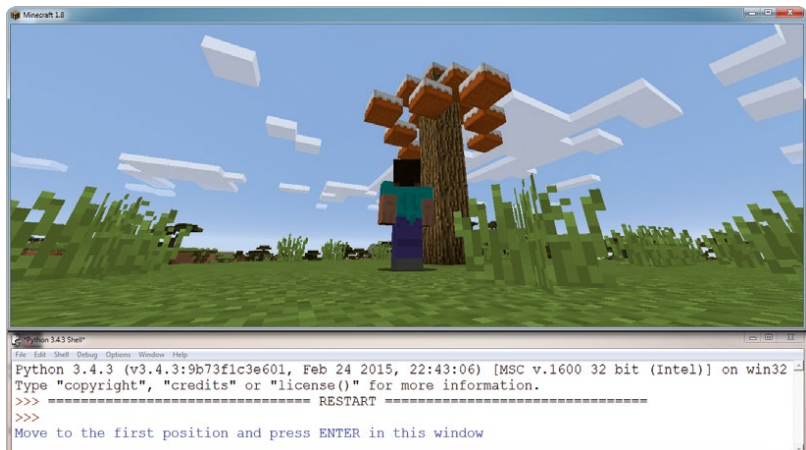


Bild 11.7: Ich begeben mich zu einer Ecke der Struktur, die ich speichern will.

Nun fliege ich zur gegenüberliegenden Ecke der Struktur und drücke erneut **Enter** (Bild 11.8).



Bild 11.8: Jetzt bin ich schräg gegenüber.

Das Programm fordert mich daraufhin auf, einen Namen für die Struktur einzugeben. Wie Bild 11.9 zeigt, habe ich meine Struktur "Cake tree" genannt.

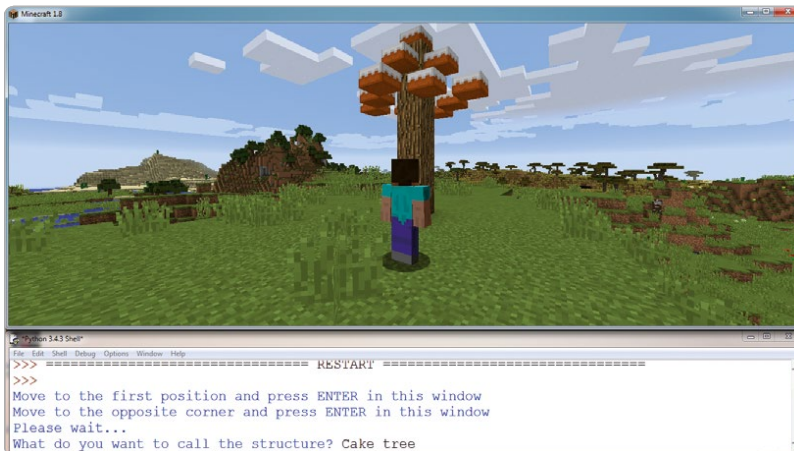


Bild 11.9: Ich gebe den Namen ein, unter dem die Struktur gespeichert werden soll.

Schließlich führe ich `loadCollection.py` aus, fliege zu der Position, an der eine Kopie meiner Struktur erstellt werden soll, und gebe den Namen der

Struktur ein, die ich erstellen will (Bild 11.10). Das Programm beginnt vor meinen Augen mit dem Bau. Das ist fast wie Zauberei!



Bild 11.10: Um die Kopie zu erstellen, muss ich nur den Namen der Struktur eingeben – und voilà: Der Bau beginnt.

Du kannst diesen Vorgang mit beliebig vielen Gebäuden oder Strukturen durchführen. In Bild 11.11 etwa habe ich eine Hütte kopiert. Nach dem erstmaligen Kopieren einer Struktur kannst du sie jederzeit laden – führe einfach nur *loadCollection.py* aus und gib den Strukturnamen ein!

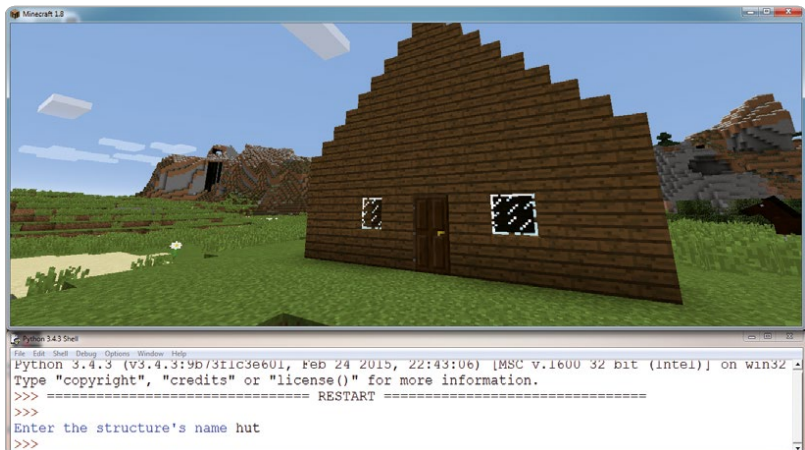


Bild 11.11: Mit diesem Programm kann ich zahllose Strukturen speichern – zum Beispiel meine Hütte.

Weitere Module mit pip installieren

Zusätzlich zu `pickle` und `shelve` kannst du auch Tausende andere Module importieren, um sie in deinen Python-Programmen zu verwenden. Weil es aber so unglaublich viele Module gibt, ist ihre korrekte Installation sehr wichtig. Damit du es bei der Modulinstallation einfacher hast, bietet Python einen Paketmanager namens `pip`. Ein *Paketmanager* ist eine Software, die eine Datenbank mit weiterer Software enthält, die du auf deinem Computer installieren kannst. Zudem bietet der Paketmanager Funktionen, die die Installation, das Upgrade und die Deinstallation der anderen Software zum Kinderspiel machen.

Mit dem Paketmanager `pip` kannst du Pakete in Python installieren, aktualisieren und entfernen. Überdies verfügt `pip` über eine umfangreiche Sammlung von Modulen, die du in Python einsetzen kannst. In diesem Abschnitt erfährst du, wie ein Paket mit `pip` installiert wird. Außerdem präsentieren wir das `Flask`-Modul, mit dem du eine Website erstellen kannst!

Wenn du die aktuellste Version von Python 3 verwendest, ist `pip` bereits vorinstalliert. Unter früheren Python-Versionen ist `pip` jedoch möglicherweise nicht installiert. Die einfachste Möglichkeit, an `pip` zu kommen, ist die Installation der aktuellen Python-Version. Informationen dazu findest du für Windows auf »Python installieren« auf Seite 4 und für den Mac auf Seite 17.

Wir wollen uns einmal ansehen, wie `pip` funktioniert. Je nach verwendetem Betriebssystem kannst du `pip` auf unterschiedliche Weise nutzen. Beachte dabei aber auf jeden Fall die für deinen Computer passenden Anweisungen!

pip unter Windows verwenden

Wenn du `pip` unter Windows verwendest, musst du die Windows-Eingabeaufforderung öffnen. Die Eingabeaufforderung ähnelt der Python-Shell. Hier kannst du jeweils einen Befehl pro Zeile eingeben. Dieser Befehl wird dann ausgeführt, sobald du **Enter** drückst.

Zum Öffnen der Eingabeaufforderung drückst du die Windows-Taste oder öffnest das Startmenü und suchst nach `cmd`. Wenn du das Programm öffnest, siehst du ein schwarzes Fenster (Bild 11.12).

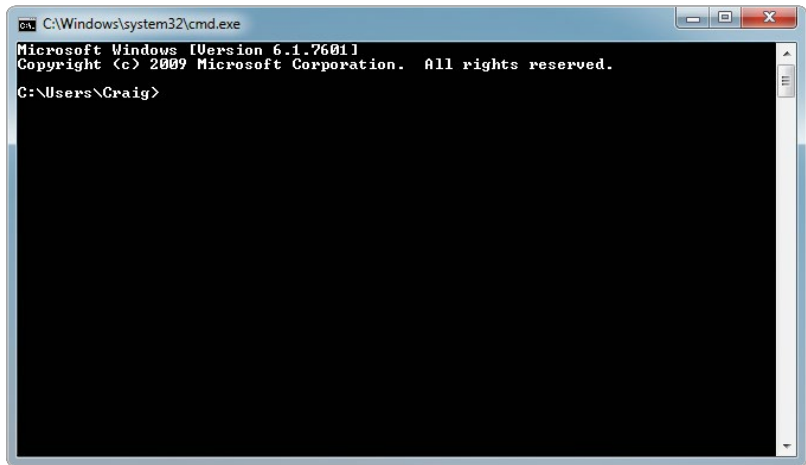


Bild 11.12: Windows-Eingabeaufforderung

Um pip in der Eingabeaufforderung zu verwenden, gibst du pip ein, gefolgt von der Aktion, die du durchführen willst. Installieren wir beispielsweise einmal das Python-Modul Flask, mit dem du in Python Websites erstellen kannst. Gib also den folgenden Befehl an der Eingabeaufforderung ein:

```
> pip install Flask
```

Auf der Python-Paketindexwebsite unter <http://pypi.python.org/> findest du viele andere Python-Module, die zur Installation bereitstehen.

pip auf einem Mac oder dem Raspberry Pi verwenden

Wenn du pip auf einem Mac oder einem Raspberry Pi verwendest, musst du, damit es funktioniert, sudo am Anfang des Befehls einfügen. Gib beispielsweise folgende Zeile ein:

```
$ sudo pip install Flask
```

Wird ein Fehler angezeigt, dann überprüfe noch einmal anhand der Installationsanleitungen in Kapitel 1, ob du Python auf dem Mac bzw. Raspberry Pi richtig installiert hast.

Auf der Python-Paketindexwebsite unter <http://pypi.python.org/> findest du viele andere Python-Module, die zur Installation bereitstehen.

Beispiel: Das pip-Modul Flask verwenden

Flask ist ein Python-Modul, mit dem du Websites entwickeln kannst. In diesem Abschnitt werde ich beschreiben, wie du eine einfache Flask-Website erstellst und diese mit Minecraft integrierst, um auf diese Weise die Position deines Spielers auf einer Website anzuzeigen!

Mit Flask benötigst du zum Erstellen und Verwalten einer Website nur einige wenige Codezeilen. Du schreibst zunächst ganz normal den Python-Code und musst dann lediglich ein paar zusätzliche Informationen für Flask ergänzen. Wenn du danach den Code ausführst, wird eine Website erstellt, die dein Computer abrufen kann. Diese Website kannst du dir dann in deinem Webbrowser ansehen.

Listing 11.5 erstellt eine einfache Flask-Website, die die wohl wichtigste Information über mich enthält: meinen Namen.

```
from flask import Flask
app = Flask(__name__) ❶

@app.route("/") ❷
def showName():
    return "Craig Richardson" ❸

app.run() ❹
```

namePage.py

Listing 11.5: Python-Programm, das mit Flask eine Website erstellt

Damit du Flask verwenden kannst, musst du zunächst Flask mit der `Flask()`-Funktion erstellen ❶. Das Argument `__name__` informiert Flask darüber, dass dein Flask-Projekt in dieser Datei enthalten ist und dass nicht woanders nach anderen Bestandteilen des Programms gesucht werden muss. Beachte die beiden Unterstriche (ja, es sind zwei!) am Anfang und am Ende des `__name__`-Arguments.

Das `@app.route()`-Tag verwendet einen Decorator. *Decorators* vermitteln Python zusätzliche Informationen über deine Funktionen. In diesem Programm beispielsweise wird Flask mit dem Decorator `@app.route()` darüber informiert, in welchem Teil der Website die Funktion verwendet werden wird. `"/` sagt Flask, dass die `showName()`-Funktion auf der Homepage verwendet werden wird ❷. Die `return`-Anweisung in der Funktion zeigt Flask dann, was auf der Seite angezeigt werden wird. In diesem Beispiel wird mein Name zurückgegeben, d. h., dieser erscheint

auf der Seite ③. Die letzte Zeile des Programms schließlich weist Python an, Flask zu starten, wenn die Datei ausgeführt wird ④.

Speichere diese Datei unter dem Namen *namePage.py* im Ordner *files*. Zum Erstellen einer eigenen Website ergänzt du natürlich deine eigenen Texte.

Klicke dann in IDLE auf **Run ▶ Run-Modul**, um die Website auszuführen. Das Programm wird gestartet und erzeugt eine Websitedatei, die du in einem Webbrowser öffnen kannst. Um herauszufinden, wo die Website gespeichert ist, musst du die Codezeile lesen, die dein Programm zu Beginn der Ausführung ausgibt. Wenn ich das Programm ausführe, dann sieht das bei mir so aus:

* Running on <http://127.0.0.1:5000/> (Press CTRL+C to quit)

Dieser Zeile kann ich entnehmen, dass die Eingabe *http://127.0.0.1:5000/* in die Adresszeile eines Webbrowsers die Flask-Website, die ich gerade erstellt habe, in eben jenem Webbrowser öffnen wird. Starte ich nun also meinen Browser und rufe diese Website auf, dann wird mein Name auf der Seite angezeigt (Bild 11.13).

Hinweis

Die Website, die du mit diesem Programm erstellt hast, ist zu diesem Zeitpunkt nur auf deinem Computer vorhanden. Das bedeutet, dass nur du darauf zugreifen kannst – ein Zugriff durch andere Benutzer über das Internet ist nicht möglich. Das nur zur Sicherheit.

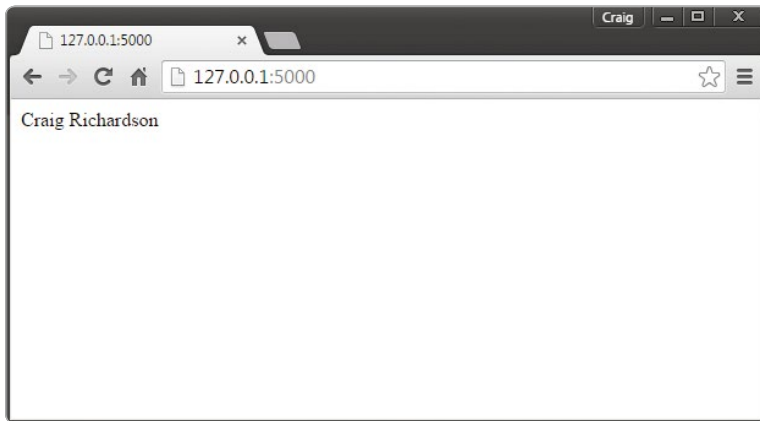


Bild 11.13: Das ist meine Website! Was hier angezeigt wird, kannst du in deinen Programmen ändern. Und? Was wirst du dort schreiben?

Um die Programmausführung zu beenden, wechselst du zu IDLE und drückst dort **STRG+C** oder klickst auf **Shell ▶ Restart Shell**.

Hinweis

In diesem Kapitel erhältst du nur eine sehr allgemeine Einführung in Flask. Es ist jedoch ausgesprochen nützlich, Flask gut zu kennen, da du hiermit in Python sehr schnell interaktive Websites erstellen kannst. Wenn du mehr über Flask erfahren möchtest, sieh dir das Tutorial auf der Flask-Website an (<http://flask.pocoo.org/docs/0.10/tutorial/>).

Mission 67: Eine Positionswebsite

Eine der besten Eigenschaften von Python besteht darin, dass es so einfach ist, Funktionen aus verschiedenen Modulen in ein einzelnes Programm zu integrieren. Wir setzen das Minecraft-Python-API-Modul bereits seit Beginn dieses Buchs ein; außerdem hast du gerade das Flask-Modul kennengelernt. Beide lassen sich mit nur ganz wenigen Schritten integrieren.

In dieser Mission kombinierst du die Minecraft-Python-API mit Flask, um die Position eines Spielers auf einer Webseite anzuzeigen.

Erstelle eine neue Datei in IDLE und speichere sie als *positionPage.py* im Ordner *files*. Hierzu musst du die Position des Spielers aus Minecraft abrufen und sie mithilfe einer Funktion mit dem Flask-Tag `@app.route("/")` auf der Webseite anzeigen. Als Ausgangspunkt für diesen Code kannst du Listing 11.5 verwenden. Schreibe den Code so, dass die Position im Format "x 10, y 110, z 12" angezeigt wird.

Führe das Programm aus und rufe die fragliche Webseite auf. Nicht schlecht, oder? Mit Flask kannst du Webseiten erstellen, auf denen Informationen aller Art angezeigt werden können. Du kannst diese Seiten sogar ins Internet hochladen, um sie mit deinen Freunden zu teilen!

Was du gelernt hast

In diesem Kapitel hast du gelernt, wie Dateien in Python verwendet werden. Du hast gesehen, wie man Dateien mithilfe der Standardbibliothek aus Python liest und schreibt. Dies ermöglicht dir die Kontrolle über Dateien beim Entwickeln deiner eigenen Programme. Außerdem hast du dich mit der Verwendung von Modulen in Python vertraut gemacht. Mit Modulen erweiterst du deine Fähigkeiten und den Funktionsumfang von Python ganz erheblich.

Wir haben uns das `pickle`-Modul, das `shelve`-Modul und den `pip`-Paketmanager genauer angesehen. Das `pickle`- und das `shelve`-Modul werden für unterschiedliche Zwecke verwendet. Mit dem `pickle`-Modul speicherst du den Wert einer einzelnen Variablen. Das ist besonders praktisch, wenn diese Variable eine mehrdimensionale Liste oder ein Wörterbuch enthält, denn diese lassen sich mit der Standardbibliothek nur schwer speichern und öffnen. Das `shelve`-Modul bietet dieselben Vorteile wie das `pickle`-Modul, erweist sich aber als noch flexibler, wenn es um die gleichzeitige Speicherung mehrerer Werte in einer wörterbuchartigen Struktur geht. Schließlich hast du auch gesehen, wie neue Module mit `pip` installiert werden. Du hast dabei das `Flask`-Modul kennengelernt, mit dem sich Websites in Python schnell und flexibel erstellen lassen.

Mit diesem Wissen hast du vier Missionen absolviert. Die Aufgabe der ersten Mission war die Erstellung von To-Do-Listen, die du in Minecraft anzeigen kannst, um jederzeit zu wissen, woran du gerade arbeitest. In der zweiten Mission hast du eigene Gebäude als Datei gespeichert und sie dann in dieselbe oder in eine andere Welt geladen. Die dritte Mission stellte eine Modifikation der zweiten dar, um zahlreiche Gebäude gemeinsam in derselben Datei zu speichern, anstatt jedes Gebäude sepa-

rat ablegen zu müssen. Die letzte Mission schließlich hat gezeigt, wie du mithilfe des Flask-Moduls eine Webseite erstellst, die die aktuelle Position des Spielers anzeigt.

Du hast bis hierher sehr gute Arbeit geleistet. Nun folgt auch schon das letzte Kapitel. Dort wirst du allerlei über Klassen und die Konzepte der objektorientierten Programmierung (OOP) lernen. Dabei handelt es sich um einen beliebten Programmieransatz, der die Wiederverwendung von Code gestattet.

Kapitel 12

Jetzt geht's ans Eingemachte: OOP

Die Möglichkeit zur Wiederverwendung von Code ist ein sehr wichtiger Aspekt bei der Programmierung. Schließlich lassen sich auf diese Weise Arbeit und Zeit einsparen. Bei Schleifen und Funktionen hast du dieses Prinzip bereits kennengelernt. Nun kommen wir darauf zurück – bei der *objektorientierten Programmierung (OOP)*.

OOP ist ein Programmierungsprinzip, bei dem Funktionen und Variablen zu *Klassen* zusammengefasst werden. Jede Klasse kann zum Erstellen von *Objekten* verwendet werden, die diese Variablen und Funktionen wie die Klasse selbst nutzen können. Du kannst zahlreiche Objekte derselben Klasse erstellen, d. h., Variablen und Funktionen der Klasse sind mehrfach verwendbar.

Ist eine Funktion Bestandteil einer Klasse, dann wird sie als *Methode* bezeichnet. Ähnlich heißt eine Variable, die zu einer Klasse gehört, *Attribut*.

In diesem Kapitel wirst du die Grundlagen der objektorientierten Programmierung kennenlernen und Code in Form von Klassen wiederverwenden. Das Beherrschen der OOP-Prinzipien und Klassen ermöglicht das Entwickeln von Programmen im Handumdrehen. Sogar Spiele kannst du per OOP programmieren. In den Missionen in diesem Kapitel wirst du mithilfe von Klassen einige einfache Programme erstellen. Wir werden zunächst nur ein einfaches Gebäude anlegen, aber schon bald wirst du eine ganze Stadt bauen!

Grundlagen der objektorientierten Programmierung

Die objektorientierte Programmierung ist sehr beliebt, denn man kann mit ihr alle möglichen coolen Softwareanwendungen schreiben. Allerdings ist der Einstieg in die OOP nicht ganz einfach. Deswegen brauchen wir eine Analogie, die du besser kennst: dich.

Du bist ein Mensch. Du hast auch eine Anzahl von Methoden: Du kannst essen, atmen, schlafen, bis zehn zählen und viele andere Dinge. Attribute fehlen bei dir auch nicht: Name, Alter, Größe, Schuhgröße usw.

Deine Freundin Marie hat dieselben Methoden wie du, denn auch sie kann essen, atmen, schlafen, bis zehn zählen und viele andere Dinge. Außerdem hat sie auch dieselben Attribute (Name, Alter usw.), doch diese enthalten andere Werte als bei dir.

Sagen wir es, wie es ist: Jeder hat diese Methoden und Attribute. Insofern kann man Menschen als Klasse beschreiben. Marie und du sind beides Menschen, d. h., man könnte auch sagen, dass ihr beide Objekte der Klasse Mensch seid.

In der objektorientierten Programmierung heißen Objekte *Instanzen* einer Klasse. Alle Objekte weisen die Methoden und Attribute der Klasse auf, aber die Werte der Attribute können für jedes Objekt anders sein.

Wenden wir uns nun wieder Python zu und erstellen wir dort eine Klasse.

Eine Klasse erstellen

Du wirst zunächst eine Klasse und dann alle deine Objekte aus dieser Klasse erstellen. Zum Erstellen einer Klasse gibst du das Schlüsselwort `class`, den Namen, den du der Klasse geben möchtest, und in Klammern die Klasse `object` an (die Klasse `object` erkläre ich in »Eine Klasse vererben« auf Seite 357:

```
class <Klassenname>(object):  
    def __init__(self):  
        # Initialisierung
```

Es gilt als empfohlene Praxis, die Namen von Klassen mit einem Großbuchstaben am Anfang zu schreiben. Dies erleichtert die Unterschei-

derung zwischen Klassen und Funktionen, denn Letztere beginnen mit einem Kleinbuchstaben.

Wenn du eine neue Klasse erstellst, musst du die Methode `__init__()` einschließen und `self` als Argument übergeben. Das Argument `self` ist für jede Methode in einer Klasse erforderlich. Es referenziert, zeigt also auf die Klasse, zu der die Methode gehört. Die Methode `__init__()` sagt Python, was mit der Klasse geschehen soll, wenn du sie zum ersten Mal in einem Programm verwendest. Dies bezeichnet man als *Initialisieren* der Klasse. (`__init__()` ist quasi die Kurzform dafür.)

Wir wollen nun exemplarisch eine Klasse namens `Cat` erstellen und dann einige Katzenobjekte anlegen. Die Klasse `Cat` speichert für jede Katze zwei Attribute: den Namen (`name`) und das Gewicht in Kilogramm (`weight`). Jedes Katzenobjekt hat dabei separate Werte für `name` und `weight`. Öffne eine neue Datei im IDLE-Texteditor und speichere sie als *catClass.py* in einem neuen Ordner namens *classes*. Dann gibst du den folgenden Code ein, um eine Klasse namens `Cat` zu erstellen:

```
class Cat(object):  
    def __init__(self, name, weight): ❶  
        self.name = name ❷  
        self.weight = weight ❸
```

catClass.py

In diesem Beispiel nimmt die Methode `__init__()` drei Argumente entgegen ❶. Das erste Argument ist `self`. Es ist wie erwähnt für jede Klassenmethode erforderlich. Das zweite Argument `name` wie auch das letzte Argument `weight` sind zusätzliche Argumente, mit denen Attribute für alle Katzen erstellt werden sollen.

In den letzten beiden Zeilen werden die Attribute `name` ❷ und `weight` ❸ erstellt und auf die Werte der Argumente `name` und `weight` gesetzt. Wenn du Attribute in einer Klasse erstellst, verwendest du die Punktnotation mit `self`. Attribute sind also immer an `self` zu erkennen, und deswegen weiß auch Python, dass ein Attribut zu der Klasse gehört.

Als Nächstes lernst du, wie du mithilfe dieser Klasse Objektinstanzen erstellst.

Ein Objekt erstellen

Wir wollen einmal mit der neu angelegten Klasse einige Katzenobjekte erstellen, d. h. Instanzen der Klasse `Cat`.

Die Initialisierung eines Objekts ähnelt dem Erstellen einer Variablen. Zur Objektinitialisierung gibst du den Namen des Objekts, ein Gleichheitszeichen (=) und den Klassennamen ein. Die an die Klasse übergebenen Argumente stehen in Klammern, ganz ähnlich wie bei einem Funktionsaufruf.

Nehmen wir an, uns sei eine Katze zugelaufen, die wir Fluff nennen wollen. Mit der Klasse `Cat` können wir ein Katzenobjekt namens `fluff` anlegen, indem wir den folgenden Code hinter der letzten Zeile von `catClass.py` einfügen (beachte, dass diese Zeile nicht eingerückt ist):

catClass.py

```
class Cat(object):
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight
```

```
fluff = Cat("Fluff", 4.5)
```

Wenn du ein Objekt erstellst, hängt die Anzahl der Argumente, die du angeben musst, von den Argumenten in der zugehörigen Funktion `__init__()` ab. Hier schließen wir zwei Argumente ein: eines für `name` ("Fluff") und ein weiteres für `weight` (4.5). Beim Erstellen eines Objekts musst du das Argument `self` nicht angeben, da es automatisch von Python hinzugefügt wird.

Ein solches Erstellen eines Objekts wird auch als das *Aufrufen eines Konstruktors* bezeichnet. »Konstruktor« ist eine häufig verwendete Bezeichnung für die Methode `__init__()`, denn diese »konstruiert« beim Aufruf eine Klasse. Bei `__init__()` handelt es sich um eine spezielle Methode, da sie nicht über ihren Namen referenziert, sondern ausgeführt wird, wenn du unter Angabe des Klassennamens ein Objekt erstellst. Hier ruft beispielsweise der Code `fluff = Cat("Fluff", 4.5)` die Methode `__init__()` auf, die ein `Cat`-Objekt namens `fluff` erstellt.

Nun wollen wir uns der Frage zuwenden, wie die Attribute des Objekts `fluff` abgerufen werden.

Auf Attribute zugreifen

Um mehr Informationen zu einem Objekt zu erhalten, kannst du auf seine Attribute zugreifen. Füge etwa in `catClass.py` den folgenden Code hinter dem Objekt `fluff` ein, um das Attribut `weight` des Objekts `fluff` auszugeben:

```
print(fluff.weight)
```

catClass.py

Der Wert, der ausgegeben wird, wenn du das Programm ausführst, sollte »4.5« sein, denn dies ist der Wert, den du dem Attribut `weight` beim Erstellen des Objekts zugewiesen hast.

Achte mal auf die Verwendung der Punktnotation zwischen dem Objekt-namen `fluff` und dem Attribut `weight`. Der Punkt besagt, dass du dieses Attribut verwenden möchtest, das zu einem bestimmten Objekt gehört. In diesem Fall gehört der Wert des Attributs `weight` zum Objekt `fluff`. Immer, wenn du den Wert eines Objektattributs abrufst oder festlegst, verwendest du die Punktnotation.

Du kannst den Wert eines Attributs genauso ändern, wie du es bei einer Variablen tun würdest: mit einem Gleichheitszeichen (=). Nehmen wir an, der Winter geht zu Ende und Fluff hat in den letzten Monaten ein bisschen zugenommen. Deswegen müssen wir den Wert des Gewichts auf 5 korrigieren. Zu diesem Zweck ändern wir das Attribut `weight` im Objekt `fluff` in 5:

```
fluff.weight = 5
```

catClass.py

Ab sofort wird, wenn du das Attribut `weight` für das Objekt `fluff` abrufst, dieses den Wert 5 haben.

Mit dem, was du nun über das Anlegen einer Klasse und das Erstellen von Instanzen weißt, kannst du in Minecraft einige wirklich coole Dinge anstellen.

Mission 68: Positionsobjekte

Wir haben im gesamten Buch bereits alle möglichen Positionen in Minecraft gespeichert: dein Haus, eine Burg, ein Schloss usw. Hierzu haben wir Variablen, Listen, Tupel und Wörterbücher auf unterschiedlichste Weise verwendet.

Aber auch mithilfe der objektorientierten Programmierung kannst du Informationen wie etwa Positionen erstellen und speichern. Es ist z. B. möglich, in Objekten die Koordinaten unterschiedlichster Positionen abzulegen.

Jede Position hat eine x-, eine y- und eine z-Koordinate, doch sind diese Werte für jeden Standort unterschiedlich. Durch Erstellen einer Positions-klasse kannst du die Koordinaten der verschiedenen Standorte abspei-

chern und später abrufen. Dies ermöglicht es dir, all die tollen Dinge, die du in Minecraft gebaut hast, im Auge zu behalten. So kannst du unkompliziert auf die Koordinaten aller deiner Minecraft-Schöpfungen zugreifen und den Spieler im Nu dorthin teleportieren lassen!

Listing 12.1 enthält den Anfang der Klasse `Location`. Wenn der Code fertig ist, kannst du ihn verwenden, um die Koordinaten einer Position in einem einzigen Objekt abzulegen. Kopiere den Code in eine neue Datei und speichere diese als `locationClass.py` im Ordner `classes`.

locationClass
.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

class Location(object): ❶
    def __init__(self, x, y, z):
        self.x = x ❷
        # Hier Attribute y und z hinzufügen ❸

bedroom = Location(64, 52, -8) ❹
mc.player.setTilePos(bedroom.x, bedroom.y, bedroom.z) ❺
```

Listing 12.1: Anfang der Klasse `Location`

Um die Klasse einzurichten, habe ich das Schlüsselwort `class` angegeben und die Klasse `Location` genannt ❶. Bei ❹ befindet sich der Code zur Initialisierung eines Objekts namens `bedroom`; er speichert die Position des Schlafzimmers in meinem Minecraft-Haus. Mit der Methode `setTilePos()` wird die Position des Spielers auf dieses Schlafzimmer festgelegt, d. h. auf die Attribute `x`, `y` und `z` des `bedroom`-Objekts ❺. Allerdings ist das Programm nicht vollständig. Du musst die Methode `__init__()` der Klasse ergänzen und die Attribute `y` und `z` auf die Werte der Argumente setzen, die an die Methode `__init__()` übergeben wurden. Den Wert des Attributs `x` habe ich bereits festgelegt ❷, die Aufgabe, dasselbe für die Attribute `y` und `z` zu tun, überlasse ich jedoch dir ❸. Vergiss aber nicht, in deiner Version des Programms die Position *deines* Schlafzimmers bei ❹ zu verwenden!

Bild 12.1 zeigt das vollständige Programm in Aktion: Der Spieler wird in mein Schlafzimmer teleportiert.



Bild 12.1: Das Programm hat meinen Spieler in mein Schlafzimmer teleportiert.

Bonus: Trautes Heim, Glück allein

Gibt es noch andere Räume in deinem Haus, in die du dich teleportieren möchtest? Erstelle weitere Objekte der Klasse `Location`, um dich standesgemäß in deinem Haus zu bewegen!

Grundsätzliches zu Methoden

Klassen können Methoden enthalten, d. h. Funktionen, die mit der Klasse verknüpft sind. Durch das Schreiben von Methoden kannst du Funktionen erstellen, die alle Instanzen der betreffenden Klasse verwenden können. Das ist super, um Zeit zu sparen und Code wiederzuverwenden, denn du brauchst eine Methode ja nur einmal zu schreiben.

Um eine Methode zu erstellen, schreibst du eine Funktion in den Klassenkörper und verwendest dabei das Schlüsselwort `def`. Nebenbei: Du hast das Schlüsselwort `def` bereits in früheren Kapiteln dieses Buchs verwendet, um Funktionen zu erstellen. Methoden werden mit dem Schlüsselwort `def` erstellt, sind aber der Klasse untergeordnet, zu der sie gehören. Wir wollen exemplarisch einmal die Klasse `Cat` in `catClass.py` abändern. Katzen müssen natürlich auch fressen, und deswegen wollen wir die Klasse `Cat` um eine Methode namens `eat()` erweitern. Gib den

folgenden Code ein und nimm dabei die erforderlichen Änderungen an *catClass.py* vor:

catClass.py

```
class Cat(object):
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

    def eat(self, food):
        self.weight = self.weight + 0.05
        print(self.name + " is eating " + food)
```

Beachte, dass die Methodendefinition und der Methodenkörper – genau wie `__init__` – um zusätzliche vier Leerzeichen eingerückt sind. Daran erkennt Python, dass diese Elemente zur Klasse gehören.

Wie Funktionen können auch Methoden Argumente entgegennehmen. Hier nimmt die Methode `eat()` ein Argument namens `food` an, das besagt, was die Katze gerade frisst. Die Methode `eat()` erhöht das Attribut `weight` der Katze um 0.05 und gibt dann eine Meldung aus, dass die Katze das Futter frisst.

Nach dem Erstellen eines Objekts kannst du alle Methoden der zugehörigen Klasse aufrufen. Folglich kannst du auch die Methode `eat()` mit dem Objekt `fluff` verwenden. Füge den folgenden Code am Ende von *catClass.py* hinzu:

catClass.py

```
fluff = Cat("Fluff", 4.5)
fluff.eat("tuna")
```

Hier sehen wir unseren früheren Code, in dem wir ein Objekt namens `fluff` erstellt haben, das zur Klasse `Cat` gehört. Dann rufen wir die Methode `eat()` auf und übergeben ihr das Argument `"tuna"`. Führt du das Programm aus, dann sieht die Ausgabe wie folgt aus:

```
Fluff is eating tuna
```

Fluff führt sich also eine Dose Thunfisch zu Gemüte. Denke daran, dass die Methode `eat()` auch das Gewicht erhöht. Nach dem Aufrufen der Methode `eat()` fügst du den Code hinzu, mit dem das neue Gewicht von Fluff ausgegeben wird.

Du kannst Methoden auch aus Klassen heraus aufrufen, indem du die Methode aus einer anderen Methode heraus aufrufst. Wir wollen eine

weitere Methode namens `eatAndSleep()` in der Klasse `Cat` erstellen. Diese Methode ruft die Methode `eat()` auf und gibt danach aus, dass die Katze nun ein Nickerchen macht. Füge folgenden Code in `catClass.py` direkt hinter der Methode `eat()` ein. (Vergiss nicht, dass die neue Methode eingerückt werden muss, damit Python weiß, dass sie zur Klasse gehört.)

```
def eatAndSleep(self, food):
    self.eat(food)
    print(self.name + " is now sleeping...")
```

catClass.py

Um eine Methode aus der zugehörigen Klasse heraus aufzurufen, fügst du `self.` am Anfang des Methodennamens ein. Hier wird `eat()` als `self.eat()` aufgerufen. Beachte, dass hier ein Unterschied zum Aufrufen einer Methode von außerhalb der Klasse besteht. Dort musst du nämlich nur den Objektnamen und die aufzurufende Methode eingeben. Der folgende Code beispielsweise ruft die neue Methode `eatAndSleep()` für das Objekt `fluff` auf. Füge sie zu deiner Datei `catClass.py` hinzu. Dies sollte die letzte Codezeile in deinem Programm sein:

```
fluff.eatAndSleep("tuna")
```

catClass.py

Die Ausgabe des Programms sollte dann so aussehen:

```
Fluff is eating tuna
Fluff is now sleeping...
```

An dieser Stelle führe ich noch einmal das vollständige Programm auf, damit klar ist, wo die einzelnen Teile hingehören:

```
class Cat(object):
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

    def eat(self, food):
        self.weight = self.weight + 0.05
        print(self.name + " is eating " + food)

    def eatAndSleep(self, food):
        self.eat(food)
        print(self.name + " is now sleeping...")
```

```
# Hier die Methoden buildDoor() und buildWindows() aufrufen ④

def clear(self): ⑤
    mc.setBlocks(self.x, self.y, self.z,
                 self.x + self.width, self.y + self.height,
                 self.z + self.depth, 0)
    # Hier Türen und Fenster wieder entfernen ⑥

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z

ghostHouse = Building(x, y, z, 10, 6, 8) ⑦
ghostHouse.build()

time.sleep(30)

ghostHouse.clear()
ghostHouse.x = 8 ⑧
```

Listing 12.2: Die Klasse `Building` erstellt ein Gebäude.

Listing 12.2 verwendet eine Klasse namens `Building` ① mit einer `__init__()`-Methode, um die Position und Größe des Hauses festzulegen ②. Dann wird ein `Building`-Objekt namens `ghostHouse` erstellt ⑦. Das Gebäude erscheint und verschwindet dann nach 30 Sekunden wieder auf mysteriöse Weise – dank den Methoden `build()` ③ und `clear()` ⑤. Allerdings gibt es ein Problem: Das sieht gar nicht wie ein Haus aus. Im Moment stellt sich das Gebäude als großer leerer Rohbau aus Bruchstein dar.

Also musst du dafür sorgen, dass das Geisterhaus eher wie ein Haus und weniger wie eine Baustelle aussieht, denn fertige Geisterhäuser sind viel gruseliger als unfertige. Damit das Gebäude eher wie ein Haus aussieht, besteht deine Aufgabe nun darin, eine Methode hinzuzufügen, mit der auf der Vorderseite eine Tür gebaut wird, und mit einer zweiten Methode Fenster zu ergänzen. Diese beiden Methoden rufst du aus der Methode `build()` auf, d. h., Fenster und Tür werden gleichzeitig gebaut ④.

Nachdem du die Methoden zum Einbauen einer Tür und von Fenstern hinzugefügt hast, musst du die Methode `clear()` so abändern, dass auch diese Elemente entfernt werden ⑥. Sie würden sonst nämlich zurückbleiben, wenn das Haus wieder verschwindet.

Nach dem Hinzufügen der zusätzlichen Methoden verschiebst du das Gebäude an eine andere Position, indem du die Attribute `x`, `y` und `z` des

Objekts `ghostHouse` änderst und weitere Aufrufe der Methoden `build()` und `clear()` hinzufügst. Ich habe damit schon mal angefangen, indem ich die x-Position für dich geändert habe ⁸.

Wenn du das Programm ausführst, sollte das Geisterhaus unvermittelt erscheinen und dann nach 30 Sekunden wieder verschwinden, nur um an anderer Stelle aufzutauchen. Das ist schon echt gruselig!

Bild 12.2 zeigt mein Geisterhaus.



Bild 12.2: Das Geisterhaus. Es taucht einfach auf und verschwindet dann wieder.

Bonus: Arbeit für den Heimwerker

Zurzeit ist dein Geisterhaus noch ziemlich schlicht gehalten. Erweitere die Funktion `build()` mit beliebigen Elementen, um dein Haus auszubauen und ansprechend zu gestalten. Dazu verwendest du die vielen mächtigen Funktionen aus Python, die du in diesem Buch kennengelernt hast.

Mit Methoden Werte zurückgeben

Wie Funktionen können auch Methoden Werte – oder die Attribute eines Objekts – mit dem Schlüsselwort `return` zurückgeben. Nehmen wir an, du möchtest die Gewichtsangabe für unsere Katze Fluff von Kilogramm in Gramm konvertieren. 1 Kilogramm entspricht 1000 Gramm, d. h., zur Konvertierung musst du das Attribut `weight` mit 1000 multiplizieren und dann zurückgeben. Füge die folgende Methode `getWeightInGrams()` zur Klasse `Cat` in `catClass.py` hinzu:

```
class Cat(object):
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

    def getWeightInGrams(self):
        return self.weight * 1000
```

catClass.py

Um den Wert auszugeben, den die Methode zurückgegeben hat, erstellst du ein Objekt und rufst die Methode dann auf. Im folgenden Code wird das `fluff`-Objekt verwendet, und die Methode zur Berechnung des Gewichts der Katze in Gramm wird innerhalb einer `print()`-Funktion aufgerufen:

```
fluff = Cat("Fluff", 4.5)
print(fluff.getWeightInGrams())
```

catClass.py

Wenn du die Datei nun ausführst, erhältst du folgende Ausgabe:

4500

In der nächsten Mission erweitern wir das Geisterhausprogramm um eine Methode, die Informationen zum Gebäude zurückgibt.

Mission 70: Die Geisterburg

Für die vielen verschiedenen Orte, die ich in meiner Minecraft-Welt gebaut habe, verwende ich alle möglichen Namen. Da gibt es das Strandhaus, das Gut, die Tierfarm, das Lager, das Schloss, das Unterwasserschloss, das unterirdische Schloss und viele andere mehr. Das Problem besteht darin, dass diese Namen nur in meinem Kopf existieren!

Mit Klassen kannst du, wie wir in Mission 69 (Seite 344) gesehen haben, Attribute wie Position und Größe für die Dinge erstellen, die du baust. Und das geht natürlich auch mit Namen!

Wir wollen also dem Geisterhaus einen Namen geben, an den Python sich für uns erinnern soll. Dazu erweitern wir die Klasse `Building` aus Mission 69 mit einer weiteren Methode, die den Namen des Gebäudes zurückgibt. Kopiere Listing 12.3 in eine neue Datei namens *ghostCastle.py* im Ordner *classes*.

ghostCastle.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

import time

class NamedBuilding(object): ❶
    def __init__(self, x, y, z, width, height, depth, name): ❷
        self.x = x
        self.y = y
        self.z = z

        self.width = width
        self.height = height
        self.depth = depth

        self.name = name ❸

    def build(self):
        mc.setBlocks(self.x, self.y, self.z,
                     self.x + self.width, self.y + self.height,
                     self.z + self.depth, 4)
```

```

mc.setBlocks(self.x + 1, self.y + 1, self.z + 1,
             self.x + self.width - 1, self.y + self.height - 1,
             self.z + self.depth - 1, 0)

def clear(self):
    mc.setBlocks(self.x, self.y, self.z,
                 self.x + self.width, self.y + self.height,
                 self.z + self.depth, 0)

def getInfo(): ④
    # Hier den Methodenkörper von getInfo() hinzufügen

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z
ghostCastle = NamedBuilding(x, y, z, 10, 16, 16, "Ghost Castle")
ghostCastle.build()
mc.postToChat(ghostCastle.getInfo()) ⑤

time.sleep(30)

ghostCastle.clear()

```

Listing 12.3: NamedBuilding ist Building sehr ähnlich, nur verfügt es über ein zusätzliches Attribut namens name und eine zusätzliche Methode, die eine Beschreibung des Gebäudes zurückgibt.

Zunächst habe ich den Namen der Klasse in NamedBuilding geändert, damit wir sie nicht mit der Klasse Building aus der vorherigen Mission verwechseln ①. Ich habe ein zusätzliches Argument und Attribut zum Konstruktor name hinzugefügt ②. Über das Argument kann ich dem Gebäude einen Namen geben, und der Konstruktor weist diesen Namen dem Attribut name zu ③.

Deine Aufgabe besteht darin, eine Methode namens getInfo() zur neuen Klasse NamedBuilding hinzuzufügen, die den Namen und die Position des Gebäudes zurückgibt. Den Anfang der Methode getInfo() findest du bei ④. Du musst also nur noch die Methodenkörper hinzufügen. Die Methode getInfo() wird bei ⑤ für das ghostCastle-Objekt aufgerufen und gibt den zurückgegebenen String im Minecraft-Chat aus. Befindet sich das Geisterhaus beispielsweise bei $x = -310$, $y = 64$, $z = 1081$, dann sollte die Methode getInfo() den String "Ghost Castle's location is at -310, 64, 1081" zurückgeben.

Bild 12.3 zeigt mein funktionsfähiges Programm. Zwar ist die Geisterburg etwas größer, aber trotzdem sieht sie wie das Haus aus Mission 69 aus.

Das liegt daran, dass die `build()`-Methode bei beiden Häusern identisch ist. Du kannst natürlich hingehen und deine Version des Codes so abändern, dass das Gebäude eher wie eine Burg aussieht.



Bild 12.3: Die Beschreibung der Geisterburg wird angezeigt.

Bonus: Eine freundliche Begrüßung

Wäre es nicht cool, wenn der Name jedes Gebäudes, das du betrittst, automatisch im Chat angezeigt würde? Das ist durchaus möglich, aber nicht ganz trivial. Wenn du es ausprobieren möchtest, kannst du das Programm `shower.py` aus Mission 32 (Seite 157) als Ausgangspunkt verwenden. Die Datei sollte sich in deinem Ordner `ifStatements` befinden. Mit diesem Programm kannst du die Koordinaten des Spielers erfassen und, falls er sich im Gebäudeinnern befindet, die Methode `getInfo()` des `Building`-Objekts aufrufen.

Mehrere Objekte erstellen

Du kannst auch mehrere Objekte aus derselben Klasse erstellen, indem du sie mithilfe desselben Klassenkonstruktors, aber mit unterschiedlichen Namen anlegst. (Du weißt ja schon, dass *Konstruktor* nur ein anderer Name für die Methode `__init__()` ist.) Nehmen wir beispielsweise an, uns ist noch eine Katze zugelaufen. Sie heißt Stella und versteht sich gut mit Fluff. Öffne `catClass.py` und ergänze den folgenden Code, um Stella hinzuzufügen:

```
class Cat(object):
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

fluff = Cat("Fluff", 4.5)
stella = Cat("Stella", 3.9)
```

catClass.py

Jetzt haben wir zwei Katzenobjekte: `fluff` und `stella`. Beide haben dieselben Attribute (`name` und `weight`), aber mit jeweils unterschiedlichen Werten.

Ergänze den folgenden Code in *catClass.py*, um die Namen der Katzen auszugeben:

```
print(fluff.name)
print(stella.name)
```

catClass.py

Wenn du die Datei ausführst, erhältst du folgende Ausgabe:

```
Fluff
Stella
```

Die beiden Katzenobjekte haben auch Zugriff auf dieselben Methoden. Beide können die Funktion `eat()` aufrufen. Füge den folgenden Code in *catClass.py* ein:

```
fluff.eat("tuna")
stella.eat("cake")
```

catClass.py

Die Ausgabe sieht dann wie folgt aus:

```
Fluff is eating tuna
Stella is eating cake
```

Das Schreiben einer Klasse macht das Erstellen zahlreicher Objekte sehr einfach. Erstellen wir also einmal eine Vielzahl von Objekten auf einen Schlag in Minecraft!

Mission 71: Die Geisterstadt

Was ist schauriger als ein Geisterhaus? Richtig: zwei Geisterhäuser. Aber drei Geisterhäuser wären noch gruseliger. Und wie wäre es mit mehr als drei Geisterhäusern? Ich muss aufhören, mir darüber Gedanken zu machen, sonst kann ich heute Nacht nicht schlafen!

In Mission 69 (Seite 344) hast du eine Klasse erstellt, die ein Haus baut, das von selbst verschwindet. Jetzt kannst du mehrere Objekte mithilfe derselben Klasse erstellen – Python merkt sich alle Attribute und Methoden der einzelnen Objekte. So kannst du so viele Häuser bauen, wie du möchtest, und sie nach Belieben auftauchen oder verschwinden lassen.

Deine Aufgabe besteht nun darin, mindestens vier Geisterhausobjekte zu erstellen und sie als Dorf zu arrangieren. Nach einer bestimmten Zeit sollen sie alle verschwinden und an anderer Stelle auf der Karte erscheinen – wie eine echte Geisterstadt eben.

Öffne *ghostHouse.py* in IDLE. Wir verwenden dieses Programm als Ausgangspunkt. Der Code zum Erstellen eines Hauses in *ghostHouse.py* müsste wie folgt aussehen:

ghostHouse
.py

```
ghostHouse = Building(17, 22, -54, 10, 6, 8)
ghostHouse.build()

time.sleep(30)

ghostHouse.clear()
```

Speichere *ghostHouse.py* unter dem neuen Namen *ghostVillage.py* und erstelle dann mindestens drei Objekte der Klasse *Building* in der Datei, um dein Dorf zu bauen. Als Ausgangspunkt habe ich in Listing 12.4 ein zweites Objekt namens *shop* erstellt. Außerdem habe ich die aktuelle Spielerposition (ermittelt mit *player.getTilePos()*) in die Variablen *x*, *y* und *z* geschrieben. Dies erleichtert den Aufbau des Dorfes um dich herum.

ghostVillage
.py

```
pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z
ghostHouse = Building(x, y, z, 10, 6, 8)
shop = Building(x + 12, y, z, 8, 12, 10)
# Hier weitere Geisterhausobjekte erstellen
```

```
ghostHouse.build()  
shop.build()  
# Hier weitere Geisterhausobjekte bauen  
  
time.sleep(30)  
  
ghostHouse.clear()  
shop.clear()
```

Listing 12.4: Erstellen mehrerer Geisterhausobjekte

Bild 12.4 zeigt mein Geisterdorf. Nach 30 Sekunden lösen sich die Häuser plötzlich in Luft auf.



Bild 12.4: Erschrick nicht: Das ist ein Geisterdorf!

Klassenattribute

Manchmal wirst du Attribute festlegen müssen, deren Wert bei jeder Objektinstanz einer Klasse identisch ist. Insofern wäre es redundant, bei jedem Erstellen eines Objekts stets dasselbe Argument an die Klasse zu übergeben. In diesem Fall kannst du besser ein gemeinsames (vordefiniertes) Attribut in der Klasse erstellen. Dieses Attribut erhalten dann alle Objekte der betreffenden Klasse.

Teilen sich mehrere Objekte dasselbe Attribut, so spricht man von einem *Klassenattribut*. Beispielsweise gehören alle Katzenobjekte, die wir bislang erstellt haben, Craig (also mir). Ich könnte die Klasse Cat in der Datei

catClass.py nun erneut öffnen, ein Klassenattribut namens `owner` erstellen und es auf "Craig" setzen:

catClass.py

```
class Cat(object):
    owner = "Craig"

    def __init__(self, name, weight):
        self.name = name
        self.weight = weight
```

Wie du siehst, verwenden Klassenattribute kein `self` vor ihrem Namen. In diesem Beispiel ist `owner` ein Klassenattribut, während `self.name` ein Attribut ist. Beachte auch, dass du Klassenattribute außerhalb der Funktion `__init__()` definierst.

Klassenattribute funktionieren auf die gleiche Weise wie andere Objektattribute. Du kannst beispielsweise den Wert eines Klassenattributs so abrufen, wie du es auch bei einem normalen Attribut tun würdest. Um etwa Fluffs Besitzer ausfindig zu machen, können wir das Attribut `owner` des Objekts `fluff` ausgeben:

catClass.py

```
fluff = Cat("Fluff", 4.5)
print(fluff.owner)
```

Der ausgegebene Wert sollte nun "Craig" sein. Hätten wir hingegen Stellas Besitzer ausgegeben, dann wäre der Wert trotzdem derselbe gewesen, weil Klassenattribute bei jedem Objekt in der betreffenden Klasse identisch sind:

catClass.py

```
stella = Cat("Stella", 3.9)
print(stella.owner)
```

Auch hier wird der Wert "Craig" ausgegeben.

Den Wert eines Klassenattributs kannst du für einzelne Objekte ändern. Wenn du dies tust, wird der Wert nur für das betreffende Objekt geändert, nicht jedoch für andere Objekte in der Klasse. Mein Kumpel Matthew hat sich in Stella verguckt und sie adoptiert. Deswegen müssen wir den Wert des Besitzerattributs für Stella in "Matthew" ändern:

catClass.py

```
stella.owner = "Matthew"
print(stella.owner)
print(fluff.owner)
```

Als Ausgabe des owner-Attributs für stella erscheint "Matthew", während als Besitzer des Objekts fluff weiterhin "Craig" angegeben wird.

Nach all den Änderungen, die wir am Programm *catClass.py* vorgenommen haben, sieht das nun fertige Programm wie nachfolgend gezeigt aus. Du findest es auch in den Ressourcen zu diesem Buch unter <https://www.nostarch.com/pythonwithminecraft/>.

```
class Cat(object):
    owner = "Craig"

    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

    def eat(self, food):
        self.weight = self.weight + 0.05
        print(self.name + " is eating " + food)

    def eatAndSleep(self, food):
        self.eat(food)
        print(self.name + " is now sleeping...")

    def getWeightInGrams(self):
        return self.weight * 1000
```

catClass.py

```
fluff = Cat("Fluff", 4.5)
print(fluff.owner)
stella = Cat("Stella", 3.9)
print(stella.owner)

print(fluff.weight)
fluff.eat("tuna")
fluff.eatAndSleep("tuna")

print(fluff.getWeightInGrams())

print(fluff.name)
print(stella.name)

fluff.eat("tuna")
stella.eat("cake")

stella.owner = "Matthew"
print(stella.owner)
print(fluff.owner)
```

Du weißt nun, wie du Objekte benutzt. Allerdings kannst du ihr Potenzial noch steigern – dank Vererbung!

Grundsätzliches zur Vererbung

Vererbung findet statt, wenn Klassen dieselben Methoden und Attribute wie andere Klassen verwenden. Nehmen wir zum Beispiel Enten. Das sind bekanntermaßen Vögel. Folglich weisen sie dieselben Methoden wie alle anderen Vögel auf (Fliegen, Fressen usw.). Gleiches gilt für die Attribute, die sie mit anderen Vögeln gemein haben: Gewicht, Spannweite usw. Man kann also sagen, dass Objekte einer Klasse *Enten* Attribute und Methoden von der Klasse *Vögel* erben. Bild 12.5 stellt diese Beziehung grafisch dar.

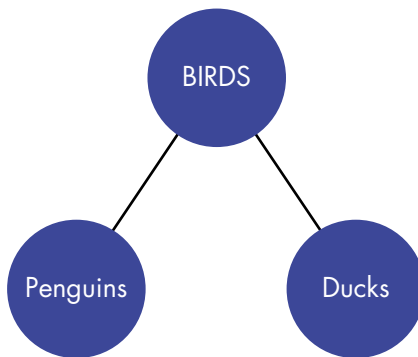


Bild 12.5: Pinguine und Enten sind Vogelarten.

Die Klasse, von der andere Klassen erben, wird als *übergeordnete Klasse* bezeichnet, während die erbende Klasse *untergeordnete Klasse* heißt.

Die Vererbung ist eine nützliche Sache, denn hiermit lassen sich subtile Unterschiede zwischen ähnlichen Objekten abbilden. Auch Pinguine sind beispielsweise Vögel, aber sie können – anders als die meisten anderen Vögel – unter Wasser schwimmen. Zur Darstellung von Pinguinen musst du eine untergeordnete Klasse erstellen, die von der Vogelklasse erbt, aber Anpassungen umfasst, die den Pinguinen das Tauchen ermöglichen. Solche Anpassungen sind der Grund für das Erstellen von untergeordneten Klassen: Du kannst die wesentlichen Merkmale der übergeordneten Klasse beibehalten, musst aber den Code nicht jedes Mal neu schreiben, sondern kannst die Methoden und Attribute, die du brauchst, in der untergeordneten Klasse hinzufügen.

Eine Klasse vererben

Eine untergeordnete Klasse, die von einer übergeordneten Klasse erbt, kann alle Methoden und Attribute der übergeordneten Klasse verwenden. Außerdem kannst du in der untergeordneten Klasse weitere Klassen und Attribute hinzufügen, ohne dass die ursprüngliche übergeordnete Klasse hierdurch geändert würde.

Wir wollen das an einem weiteren Beispiel aus der Welt der Vögel erläutern. Zunächst schreiben wir den Code für die übergeordnete Klasse `Bird`. Öffne eine neue Datei in IDLE, nenne sie *birdClass.py* und füge dann zum Erstellen der Klasse folgenden Code hinzu:

```
class Bird(object): ❶
    def __init__(self, name, wingspan): ❷
        self.name = name
        self.wingspan = wingspan

    def birdcall(self): ❸
        print("chirp")

    def fly(self): ❹
        print("flap")
```

birdClass.py

Wir erstellen hier also eine Klasse namens `Bird` ❶. Beachte jedoch, dass diese Klasse `Bird` von `object` erbt. Die Klasse `object` ist eine Basisklasse, auf der alle anderen Klassen aufbauen. Alle Klassen erben von der Klasse `object`, d. h., du verwendest diese Klasse, wenn keine übergeordneten Klassen vorhanden sind, von denen deine neue Klasse erben kann. Auch wenn es viele verschiedene Vererbungsstufen gibt und zahlreiche Klassen an- und voneinander erben, ist `object` immer die übergeordnete Klasse, die auf der höchsten Vererbungsstufe steht.

Die Methode `__init__()` der Klasse `Bird` nimmt zwei Argumente entgegen, mit denen zwei Attribute festgelegt werden: der Name des Vogels (`name`) und seine Spannweite (`wingspan`) ❷. Ferner gibt es die beiden Methoden `birdcall()` ❸ und `fly()` ❹. Im Moment gibt die Methode `birdcall()` lediglich "chirp" und die Methode `fly()` "flap" aus.

Nun erstellst du in derselben Datei ein Objekt namens `gardenBird` aus der Klasse `Bird`:

```
gardenBird = Bird("Geoffrey", 12)
gardenBird.birdcall()
gardenBird.fly()
```

birdClass.py

Der Code gibt Folgendes aus:

```
chirp
flap
```

Nachdem wir nun eine übergeordnete Klasse erstellt haben, legen wir als Nächstes eine untergeordnete Klasse an, die von der übergeordneten Klasse erbt, aber eine eigene Methode enthält. Dies tun wir im nächsten Abschnitt.

Neue Methoden zu untergeordneten Klassen hinzufügen

Nun fügen wir eine Klasse für Pinguine zu *birdClass.py* hinzu und nennen diese Penguin. Weil Pinguine unter Wasser schwimmen können, fügen wir eine zusätzliche Methode zur Klasse Penguin hinzu, die wir `swim()` nennen:

```
birdClass.py class Penguin(Bird):
               def swim(self):
                   print("swimming")
```

Wenn du eine untergeordnete Klasse definierst, die von einer anderen Klasse als object erben soll, dann setzt du den Namen der Klasse, von der geerbt werden soll, in Klammern. Wie du siehst, habe ich für die Klasse Penguin keine Methode `__init__()` angelegt. Dies ist auch gar nicht nötig, denn Penguin erbt von der Klasse Bird und verwendet folglich auch deren Methode `__init__()`. Wir wollen diese Methode `__init__()` einmal verwenden und die Funktion `swim()` ausprobieren. Dazu erstellen wir einen Penguin:

```
birdClass.py sarahThePenguin = Penguin("Sarah", 10)
sarahThePenguin.swim()
```

Der Code generiert folgende Ausgabe:

```
swimming
```

Die Klasse Penguin kann auch die Methoden `fly()` und `birdcall()` verwenden, die sie von Bird geerbt hat.

```
sarahThePenguin.fly()  
sarahThePenguin.birdcall()
```

birdClass.py

Die Ausgabe sieht dann wie folgt aus:

```
flap  
chirp
```

Allerdings sind `flap` und `chirp` für Pinguine wenig sinnvoll, denn sie können gar nicht fliegen, und ihr Ruf ist eher ein Tröten! Deswegen werden wir uns in »Methoden und Attribute überschreiben« auf Seite 362 ansehen, wie vererbte Methoden überschrieben werden. Hierdurch lässt sich dieses Problem beheben.

Zunächst jedoch wollen wir uns noch einmal Minecraft zuwenden und mithilfe der Vererbung weitere Geisterhäuser erstellen.

Mission 72: Das Geisterhotel

Häuser und Hotels gehören beide zur Kategorie der Gebäude: Sie haben Türen, Fenster, Zimmer, Treppen und Wände. Hotels sind lediglich etwas aufwendigere Häuser mit Extras wie Balkonen, vielen Zimmern oder einem repräsentativen Eingang.

Kannst du einige Geisterhotels auf Grundlage des Codes programmieren, mit dem wir bereits die Geisterhäuser angelegt haben? Die Grundstruktur der Gebäude ist dieselbe. Sagen wir also, der einzige Unterschied bestünde darin, dass Geisterhotels über zusätzliche Methoden verfügen, um die Zimmer mit Teppich auszulegen und Blumen um das Gebäude herum zu pflanzen. Die Geisterhotelklasse kann also alle Methoden der Geisterhausklasse erben. Sie benötigt außerdem zwei zusätzliche Methoden für Teppiche und Blumen.

Erstelle in IDLE eine neue Datei und speichere sie als *ghostHotel.py* im Ordner *classes*. Kopiere den Code der Klasse *Building* aus dem Programm *ghostHouse.py* über die Zwischenablage in die Datei.

Dann erstellst du eine neue Klasse namens *FancyBuilding*, die von der Klasse *Building* erbt. Die Klasse *FancyBuilding* muss über eine neue Methode namens `upgrade()` verfügen, die Teppiche im Gebäudeinnern und Blumenrabatten um das Gebäude herum ergänzt. Listing 12.5 zeigt meinen Code für die Methode `upgrade()`, den du aber beliebig nach deinen Wünschen anpassen kannst.

ghostHotel.py

```

# Hier Klasse FancyBuilding erstellen

def upgrade(self):
    # Auslegeware
    mc.setBlocks(self.x + 1, self.y, self.z + 1,
                 self.x + self.width - 1, self.y, self.z + 1,
                 self.depth - 1, 35, 6)

    # Blumen
    mc.setBlocks(self.x - 1, self.y, self.z - 1,
                 self.x - 1, self.y, self.z + self.depth + 1, 37)
    mc.setBlocks(self.x - 1, self.y, self.z - 1,
                 self.x + self.width + 1, self.y, self.z - 1, 37)
    mc.setBlocks(self.x + self.width + 1, self.y, self.z - 1,
                 self.x + self.width + 1, self.y, self.z + 1,
                 self.depth + 1, 37)
    mc.setBlocks(self.x - 1, self.y, self.z + self.depth + 1,
                 self.x + self.width + 1, self.y, self.z + 1,
                 self.depth = 1, 37)

# Eine Instanz der Klasse FancyBuilding erstellen
# Methoden build() und upgrade() aufrufen

```

Listing 12.5: Methode für die Klasse FancyBuilding, mit der das Gebäude mit Teppich und Blumen verschönert wird

Nach dem Erstellen der Klasse und dem Hinzufügen der neuen Methode erstellst du eine Instanz der Klasse FancyBuilding und nennst sie ghost-Hotel. Baue das Geisterhotel mit der Methode build() und füge dann mit upgrade() die hotelspezifischen Erweiterungen hinzu.

Bild 12.6 zeigt mein aufgehübschtes Geisterhotel.

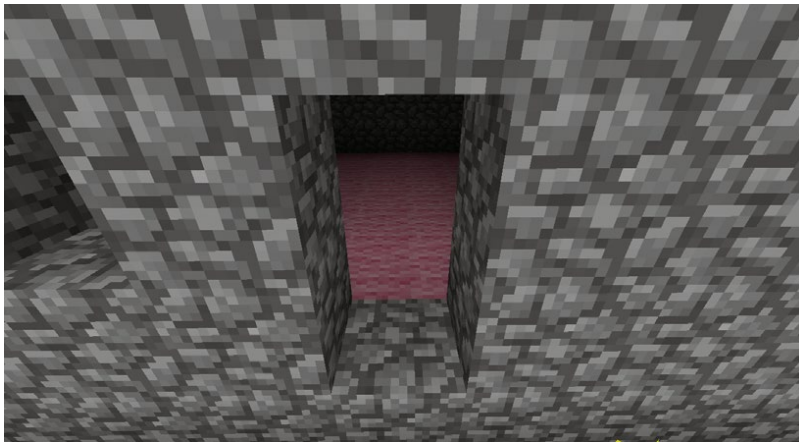
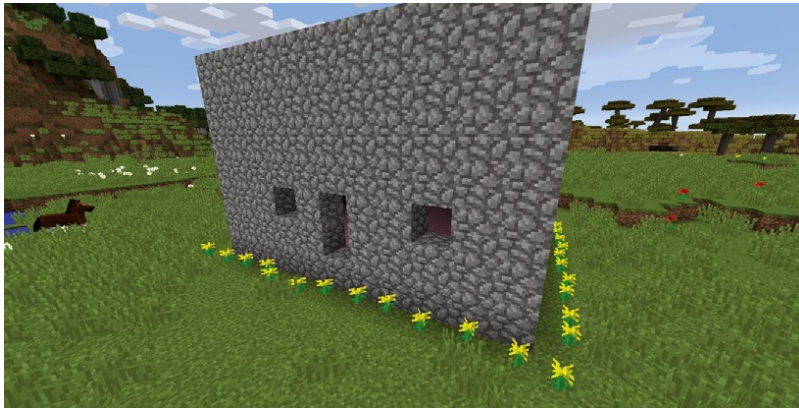


Bild 12.6: Wohlriehende Blumen. Erlesene Teppiche. Was für ein Luxus!

Bonus: Ein Dorf mit Zierrat

In Mission 71 hast du ein Geisterdorf gebaut, in dem alle Gebäude mehr oder weniger gleich aussehen. Im wirklichen Leben sehen sich die Häuser von Ortschaften allerdings nur selten sehr ähnlich. Um dieses Problem zu beheben, musst du das Programm für den Aufbau des Geisterdorfs ändern. Hierzu erstellst du verschiedene Klassen, die von der Klasse `Building` erben. Du kannst beispielsweise eine Klasse `Shop`, eine Klasse `Hospital` und eine Klasse `Restaurant` erstellen. Wenn du dann die Objekte anlegst, kannst du über die Auswahl der jeweiligen Klassen festlegen, was für Gebäude gebaut werden.

Methoden und Attribute überschreiben

Methoden und Attribute, die eine untergeordnete von einer übergeordneten Klasse geerbt hat, können neu definiert werden. Das ist nützlich, wenn man denselben Namen für eine Methode verwenden will, die sich jedoch abhängig von der jeweiligen untergeordneten Klasse unterschiedlich verhält.

In »Grundsätzliches zur Vererbung« auf Seite 356 haben wir eine Klasse `Bird` und eine Klasse `Penguin` erstellt. Die Klasse `Penguin` erbte von `Bird`, weist also die gleichen Methoden auf. Wie wir jedoch auch gemerkt haben, können Pinguine weder fliegen noch singen. Insofern sollten wir die Methoden `fly()` und `birdcall()` entsprechend anpassen. Öffne die Datei `birdClass.py` und füge folgenden Code hinzu:

birdClass.py

```
class Penguin(Bird):
    def swim(self):
        print("swimming")

    def birdcall(self): ❶
        print("sort of a quack")

    def fly(self): ❷
        print("Penguins cannot fly :(")
```

Ich habe zwei Änderungen an der Klasse `Penguin` vorgenommen: Ich habe die Methoden `birdcall()` ❶ und `fly()` ❷ hinzugefügt. Weil beide Methoden genau so benannt sind wie in der übergeordneten Klasse `Bird`, überschreiben sie die Methoden der übergeordneten Klasse.

Du rufst die Methoden auf, in dem du folgenden Code in `birdClass.py` ergänzt:

birdClass.py

```
sarahThePenguin.fly()
sarahThePenguin.birdcall()
```

Wenn du das Programm jetzt ausführst, erhältst du folgende Ausgabe:

```
Penguins cannot fly :(
sort of a quack
```

Das Überschreiben einer Methode, die aus einer übergeordneten Klasse stammt, ändert das Verhalten der Methode in der untergeordneten, nicht aber in der übergeordneten Klasse. Das bedeutet, dass Pinguine auch

weiterhin nicht fliegen werden können, dass aber andere Vögel, die von der Klasse `Bird` erben, dadurch in ihren Flugfähigkeiten nicht beeinträchtigt werden.

Du kannst auch die Methode `__init__()` in einer untergeordneten Klasse überschreiben. Das bedeutet, dass ein Objekt der untergeordneten Klasse andere Attribute oder Verhaltensweisen aufweisen kann als die übergeordnete Klasse.

Wir wollen in derselben Datei eine untergeordnete Klasse `Parrot` der Klasse `Bird` erstellen. Papageien können unterschiedlichste Farben haben. Deswegen wollen wir in der Methode `__init__()` ein zusätzliches Argument für ein Attribut namens `color` festlegen:

```
class Parrot(Bird):
    def __init__(self, name, wingspan, color): ❶
        self.name = name
        self.wingspan = wingspan
        self.color = color
```

birdClass.py

Ich habe eine neue Methode `__init__()` für die Klasse `Parrot` eingebunden, die anders als die Ursprungsklasse `Bird` über das zusätzliche Argument `color` verfügt ❶.

Wenn wir jetzt ein neues `Parrot`-Objekt erstellen, können wir auf das Attribut `color` zugreifen. Außerdem können wir weiterhin die Methoden `birdcall()` und `fly()` nutzen, weil wir diese von der übergeordneten Klasse `Bird` geerbt haben:

```
freddieTheParrot = Parrot("Freddie", 12, "blue")
print(freddieTheParrot.color)
freddieTheParrot.fly()
freddieTheParrot.birdcall()
```

birdClass.py

Der Code generiert folgende Ausgabe:

```
blue
flap
chirp
```

Denke immer daran, dass du jede Methode überschreiben kannst, die eine untergeordnete Klasse von einer übergeordneten Klasse erbt. Das gilt auch für die Methode `__init__()`. Auf diese Weise erhältst du die volle Kontrolle über Objekte und ihre zahlreichen Attribute und Methoden.

Nach all den Änderungen, die wir an *birdClass.py* vorgenommen haben, sieht das fertige Programm nun wie folgt aus. Du findest es auch in den Ressourcen zu diesem Buch unter <https://www.nostarch.com/pythonwith-minecraft/>.

birdClass.py

```
class Bird(object):
    def __init__(self, name, wingspan):
        self.name = name
        self.wingspan = wingspan

    def birdcall(self):
        print("chirp")

    def fly(self):
        print("flap")

class Penguin(Bird):
    def swim(self):
        print("swimming")

    def birdcall(self):
        print("sort of a quack")

    def fly(self):
        print("Penguins cannot fly :(")

class Parrot(Bird):
    def __init__(self, name, wingspan, color):
        self.name = name
        self.wingspan = wingspan
        self.color = color

gardenBird = Bird("Geoffrey", 12)
gardenBird.birdcall()
gardenBird.fly()

sarahThePenguin = Penguin("Sarah", 10)
sarahThePenguin.swim()
sarahThePenguin.fly()
sarahThePenguin.birdcall()

freddieTheParrot = Parrot("Freddie", 12, "blue")
print(freddieTheParrot.color)
freddieTheParrot.fly()
freddieTheParrot.birdcall()
```

In der nächsten Mission werden wir Methoden und Attribute in Minecraft überschreiben.

Mission 73: Der Geisterbaum

Wir haben mittlerweile eine ganze Reihe von Geisterdomizilen unterschiedlicher Form und Größe gebaut. Jetzt wollen wir noch einen Schritt weiter gehen und einen Geisterbaum erstellen. An sich ist das eine coole Idee, aber wie setzen wir sie um? Die Klasse `Building` ist den Gebäuden vorbehalten, die Wände und Decken haben. Bäume haben aber keine Wände und Decken. Doch keine Sorge! Wir können dieses Problem umgehen, indem wir unsere Klasse `Building` entsprechend anpassen.

Wie die Geisterhäuser erscheint und verschwindet auch der Geisterbaum mithilfe unserer Methoden `build()` und `clear()`. Allerdings müssen diese Methoden anders funktionieren, weil ein Baum nun einmal kein Haus ist. Wir müssen also eine Klasse erstellen, die von der Klasse `Building` erbt und dann die Methoden `build()` und `clear()` überschreibt.

Zum Einstieg habe ich die Funktion, mit der in der Datei *forest.py* (Seite 341) ein Baum erstellt wurde, noch einmal hervorgekramt und diese in Listing 12.6 eingefügt. Kopiere dieses Listing in eine neue Datei namens *ghostTree.py* im Ordner *classes*.

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

# Hier das Programm ghostHouse.py einfügen
# Hier eine Klasse Tree erstellen

def growTree(x, y, z): ❶
    """ Einen Baum an den angegebenen Koordinaten erstellen """
    wood = 17
    leaves = 18

    # Stamm
    mc.setBlocks(x, y, z, x, y + 5, z, wood)

    # Blätter
    mc.setBlocks(x - 2, y + 6, z - 2, x + 2, y + 6, z + 2, leaves)
    mc.setBlocks(x - 1, y + 7, z - 1, x + 1, y + 7, z + 1, leaves)

# Hier die Methoden build() und clear() für die Klasse Tree erstellen
```

ghostTree.py

Listing 12.6: Funktion zum Erstellen eines Baums

Um das Programm zu vervollständigen, kopierst du den Code für die Klasse `Building` aus *ghostHouse.py* via Zwischenablage in die neue Datei. Dann erstellst du eine neue Klasse namens `Tree`, die von der

Klasse `Building` erbt. In der Klasse `Tree` legst du dann die Methoden `build()` und `clear()` an, um die Methoden aus der Klasse `Building` zu überschreiben und einen Baum anstelle eines Hauses zu bauen. Vergiss nicht, das Argument `self` vor die Attribute der fertigen Methode `growTree()` zu setzen ❶.

Wenn du das Programm fertiggestellt hast, erstelle ein `Tree`-Objekt namens `ghostTree`. Rufe dann die Methode `build()` auf, um den Baum erscheinen zu lassen, warte ein paar Sekunden und lass ihn dann mit `clear()` wieder verschwinden.

Bild 12.7 stellt das Ergebnis des Programms dar.



Bild 12.7: Weg ist er! Das ist gruselig ...

Bonus: Ein Geisterwald

Ändere den Code in *ghostTree.py* so ab, dass ein Geisterwald errichtet wird. Stell dir nur mal vor, was für Schätze du in einem solchen Wald finden könntest!

Was du gelernt hast

Du hast in diesem Kapitel die Grundlagen eines der wichtigsten modernen Programmierkonzepte – der objektorientierten Programmierung – kennengelernt. Dabei hast du gesehen, wie eine Klasse geschrieben wird und Objekte erstellt werden und wie du mithilfe der Vererbung Klassen und das Verhalten von Objekten anpassen kannst. Diese extrem praktischen Fertigkeiten werden dir nicht nur in Minecraft nützlich sein, sondern bei allen künftigen Abenteuern, die du auf deinem Weg als Programmierer in Zukunft noch erleben wirst!

Index

Symbole

2-D-Listen. *Siehe* Listen, zweidimensionale
3-D-Listen. *Siehe* Listen, dreidimensionale
3D-Objekte
 in Listen speichern 284
= (Gleichheitszeichen)
 Werte ändern in Attributen 339
__init__() (Methode) 337
== (ist gleich-Operator) 109
 in while-Schleife 173
> (ist größer als-Operator) 109, 114
>= (ist größer/gleich-Operator) 109, 116
< (ist kleiner als-Operator) 109, 114
<= (ist kleiner/gleich-Operator) 109, 116
!= (ist ungleich-Operator) 109, 112

A

Ablaufsteuerung. *Siehe* if-Anweisungen
Abstände berechnen 186
Addition
 Operator 61
 Operator (Kurzform) 79
 Strings verknüpfen 90
and (logischer Operator). *Siehe* Logische Operatoren, and
Anführungszeichen bei Docstrings 199
Anweisungen
 break 182
 Definition 59
 elif 141
 else 138
 if 133
 try-except 97, 99
 while-else 184
append() (Funktion) 227
Argumente
 Anzahl in Objekten 338
 bei Funktionen. *Siehe* Funktionen, und Argumente
 Zeilenwechsel 199
Attribute
 an Attribut self erkennbar 337
 Definition 335

 überschreiben 362
 Werte ändern mit = (Gleichheitszeichen) 339
 zugreifen auf 338
Aufrufen
 Funktionen. *Siehe* Funktionen, aufrufen
 Konstruktor. *Siehe* Konstruktor, aufrufen
Ausdrücke, Definition 59
Ausnahmebehandlung. *Siehe* Fehlerbehandlung

B

Begrüßung bei Betreten des Gebäudes 350
Benutzereingaben annehmen 87
Berechtigungsargumente 300, 301
 a (append/Anfügen) 301
 r+ (read and write/Lese- u. Schreibrecht) 301
 r (ready only/Schreibschutz) 301
 w (write only/Leseschutz) 301
Blöcke
 an Spielerposition erstellen 95
 kreuz und quer teleportieren 81
 Lavaturm 63
 Regenbogen 63
 stapeln 61
 unter Spieler positionieren 66
 unzerstörbar machen 140
 zu Gold verwandeln 180
Blocktypen 62
Blumenkorso 161, 171
Boolesche Operatoren
 == (ist gleich-Operator) 107
 > (ist größer als-Operator) 107
 >= (ist größer/gleich-Operator) 107
 < (ist kleiner als-Operator) 107
 <= (ist kleiner/gleich-Operator) 107
 != (ist ungleich-Operator) 107
 und if-Anweisungen 155
 und while-Schleife 176
 while-Schleifen steuern mit 172
Boolesche Variablen 105
 Grundlagen 106
 Vergleichsoperatoren 108
 verketteten 108

break (Anweisung) , 182
Bugs. *Siehe* Debuggen

C

Chat
 permanenten, mit Schleife erstellen 182
 Schleife beenden 184
Chatten
 mit Strings 83
 postToChat() 85
choice() (Funktion) 236
class (Schlüsselwort) 336
close() (Funktion) 303

D

Dateiberechtigungen 300
Dateien
 arbeiten mit 300
 in Liste mit Strings umwandeln 305
 lesen 303
 lesen, zeilenweise 304
 öffnen 300
 öffnen mit shelve 320
 schreiben und speichern in 302
Daten in Variablen speichern 36
Debuggen 54, 189
Definition 356
def (Schlüsselwort) 190
 bei Methoden 341
del (Schlüsselwort)
 in Wörterbüchern 249
 mit index() (Funktion) 224
Diamantblöcke erzeugen 269
Dimensionen 272
Division
 Operator 73
 Operator (Kurzform) 79
Docstrings 198
 Anführungszeichen 199
 Definition 199
 in Funktionen 199
dump() (Funktion) 310
Dusche bauen 157

E

Editor oder Shell verwenden? 31
Einrückung bei Funktionen
 (vier Leerzeichen) 190
Einzug. *Siehe auch* Syntax, unerwarteter Einzug
Elemente mit shelve hinzufügen/bearbeiten/
 lesen 321
elif-Anweisungen
 Definition 141
 verketteten 145
else-Anweisungen, Definition 138
else-if-Anweisung. *Siehe* elif-Anweisung
Endlosschleifen. *Siehe* while-Schleife, endlose
Entfernung zu deinem Haus messen 119
Exception Handling. *Siehe* Fehlerbehandlung

F

False (Wert) 105
Fehlerbehandlung 97. *Siehe auch* try-except
 (Anweisung)
Fehlermeldungen
 bei Funktionen 192
 IndexError 219
Flask (pip-Modul)
 auf Website zugreifen 330
 Spielerposition anzeigen via 331
 verwenden 329
Fließkommazahlen. *Siehe* Floats
Floats 48
 als Schlüssel in Wörterbüchern 245
 mit Integer verketteten 92
for-else-Schleifen 267
 Syntax 267
 unterbrechen 268
for-Schleifen 253
 Definition 253
 else. *Siehe* for-else-Schleifen
 Paläste bauen mit 263
 Pyramiden bauen mit 263
 Syntax 254
 Treppen bauen mit 258
 über Wörterbücher 265
 verschachtelte 270
Friseur 105
from (Klausel), Funktionen importieren
 mit 311

Funktionen

- aufrufen 191
 - Debugging 189
 - Definition 189
 - Docstrings 199
 - Doppelpunkt 190
 - eigene definieren 190
 - Einrückung (vier Leerzeichen) 190
 - Fehlermeldungen 192
 - if-Anweisungen in 204, 205
 - importieren mit * 312
 - in Klassen 335. *Siehe auch* Methoden
 - Klammern 190
 - mit from-Klausel importieren 311
 - Modularität 190
 - Namen, innerhalb eines Programms 190
 - return (Schlüsselwort) 200
 - Rückgabewerte 200
 - Skalierbarkeit 190
 - Syntax 190
 - und Argumente 192, 193
 - Variablen inner-/außerhalb von. *Siehe*
Variablen, lokale/globale
 - Variablenwerte steuern mit 200
 - Wald pflanzen mit 194
 - while-Schleifen in 204, 208
 - Wiederverwendbarkeit 189
- Funktionskörper. *Siehe* Funktionen, Syntax

G

Gebäude

- Begrüßung bei Betreten 350
 - duplizieren 292
 - laden (Code) 299, 318
 - speichern 299, 313
 - speichern (Code) 314
- Geisterbaum 365
- Geisterburg 348
- Geisterhaus 344
- Geisterhotel 359
- Geisterstadt 352
- Geschenk auf Block ablegen 143
- getHeight() 117, 155
- getTilePos() 64
- Globale Variablen. *Siehe* Variablen, globale
- global (Schlüsselwort). *Siehe auch* Variablen,
lokale/globale

I

- id() (Funktion) 238
- IDLE
- kennenlernen 26
 - Texteditor 29
- if-Anweisungen
- Definition 133
 - einen Wertebereich testen mit 151
 - einsetzen 134
 - Elemente und 240
 - in Funktionen 204, 205
 - mit while-Schleifen verschachtelte 179
 - und boolesche Operatoren 155
 - Unterschied zur while-Schleife 161
- ImportError: No module named 'mcpip' (Fehlermeldung). *Siehe* Python, veraltete Version (Fehlermeldung)
- import (Schlüsselwort) 309
- Index in Listen 219
- IndexError (Fehlermeldung) 219
- index() (Funktion), mit del (Schlüsselwort) 224
- Initialisieren
- von Klassen 337
 - von Objekten 338
- in (Operator) 240
- input() (Funktion) 87, 94, 294
- insert() (Funktion) 227
- int() 94
- Integer
- Definition 40
 - mit Floats verketteten 92
 - Strings konvertieren in 94
- Integer (als Schlüssel in Wörterbüchern) 245

J

Java installieren

- auf Mac 18
- auf Windows-PC 5

K

Klassen

- Definition 335
 - erstellen 336
 - initialisieren 337
 - Schreibweise 336
 - vererben 357
- Klassenattribute 353

Kommentieren mit Docstrings.

Siehe Docstrings

Konstruktor aufrufen 338

Krater bauen 136

L

Lavaturm aus Blöcken 63

Leerzeichen

Einrückung bei Methoden 342

vier bei Funktionen 190

len() (Funktion) 233

Listen 217

Definition 217

dreidimensionale, ausgeben 284

dreidimensionale, zum Bauen 283

einsetzen 218

Elemente ändern 220

Elemente anhängen 223

Elemente aufrufen 219

Elemente einfügen 223

Elemente löschen 224

Elemente zufällig auswählen mit choice()
236

erster Index ist 0 219

kopieren 238

Länge ermitteln mit len() 233

manipulieren 223

mehrdimensionale 270

Schrittweite 260

Slices 239

Strings behandeln als 227

Werte aufrufen in dreidimensionalen 290

Werte aufrufen in zweidimensionalen 276

zweidimensionale 273

zweidimensionale, mit Schleifen erzeugen
279

load() (pickle-Funktion) 310

Logische Operatoren 120

and 121

Bearbeitungsreihenfolge 128

not 125

or 123

Lokale Variablen. *Siehe* Variablen, lokale

M

Mac einrichten 15

Mauer, verwittert 281

Mehrdimensionale Listen. *Siehe* Listen, mehrdimensionale

Melone, Gott der 145

Methoden

Definition 335

erstellen 341

Grundlagen 341

Syntax 342

überschreiben 362

Werte zurückgeben mit 347

zu untergeordneten Klassen hinzufügen 358

Midas 180

Minecraft mit Python verbinden 43

Minecraft.create() 33, 43

Minecraft installieren

auf Mac 15

auf Windows-PC 2

Minecraft-Python-API installieren

auf Mac 19

auf Windows-PC 7

Minecraft-Python-Setup testen 32

Minecraft Wikipedia xv

Minigame „Blöcke treffen“ 233

Missionen

1: Den Spieler teleportieren 41

2: Genau dort landen, wo man hinwill 48

3: Teleportationstour 51

4: Bug im Teleportationsprogramm
beheben 56

5: Blöcke stapeln 61

6: Der Supersprung 64

7: Die Blöcke unter dir ändern 66

8: Speed Building 69

9: Spektakuläre Säulen 74

10: Supersprung 2.0 80

11: Hallo, Minecraft-Welt! 85

12: Chatnachrichten selbst gemacht 88

13: Benutzernamen zum Chat hinzufügen 92

14: Block per Eingabe erstellen 95

15: Zahlen, nichts als Zahlen 98

16: 100 Meter in 10 Sekunden 101

17: Schluss jetzt mit der
Blockzerstörung! 107

18: Schwimme ich? 110

19: Und worin stehe ich eigentlich? 113

20: Stehe ich eigentlich auf dem Boden? 117

21: Trautes Heim, Glück allein 118

22: Der Taucher 122

23: Sitze ich auf einem Baum? 124
24: Ist dieser Block keine Melone? 126
25: Bin ich zu Hause? 130
26: Wir basteln uns einen Krater 136
27: Blöcke nicht zerschlagen –
 oder doch? 139
29: Teleportation zum gewünschten Ort 146
30: Eine Geheimtür öffnen 149
31: Teleportationsziele beschränken 153
32: Dusche gefällig? 157
33: Teleportationstour auf Zufallsbasis 163
34: Der Wasserfluch 168
35: Der Blumenkorso 171
36: Ein Tauchwettbewerb 173
37: Der König des Tanzbodens 177
38: Die Berührung des Midas 180
39: Permanenten Chat mit Schleife
 erstellen 182
40: Heiß oder kalt? 185
41: Einen ganzen Wald pflanzen 194
42: Einfach wegrationalisiert 196
43: Gedächtnisstütze für Block-IDs 202
44: Wollfärberei 206
45: Blöcke, überall Blöcke 209
46: Es bewegt sich! 213
47: Oben und unten 220
48: Fortschrittsbalken 225
49: Verschieben 229
50: Blocktreffer 233
51: Irgendein Block 237
52: Schwert mit Nachtsichtfunktion 241
53: Sightseeing 246
54: Getroffene Blöcke zählen 249
55: Der Zauberstab 254
56: Die Zaubertreppe 257
57: Säulen 262
58: Pyramide 263
59: Ergebnistafel 266
60: Der Diamantensucher 269
61: Wir verkünsteln uns 277
62: Eine verwitterte Mauer 281
63: Wie man ein Gebäude dupliziert 292
64: To-Do-Liste 305
65: Ein Gebäude speichern 313
66: Sammlung von Strukturen speichern 322
67: Eine Positionswebsite 331
68: Positionsobjekte 339
69: Das Geisterhaus 344

70: Die Geisterburg 348
71: Die Geisterstadt 352
72: Das Geisterhotel 359
73: Der Geisterbaum 365

Modularität 190

Module

 Daten speichern mit shelve 320
 Spitznamen erstellen für 313
 time 50
 verwenden 308

Multiplikation

 Operator 73
 Operator (Kurzform) 79

N

NameError: name 'x' is not defined. *Siehe* Variablen, nicht definiert (Fehlermeldung)

Neue Welt erstellen

 auf Mac 23
 auf PC 11

not (logischer Operator) 125

O

Objekte

 Definition 335
 erstellen 337, 338, 350

Objektorientierte Programmierung (OOP) 335

 Definition 335
 Grundlagen 335

Offline spielen

 auf Mac 23
 auf Windows-PC 12

Online-Ressourcen xv

open() (Funktion). *Siehe auch* Dateien, öffnen

Operatoren

 Addition 61
 Definition 60
 Division 73
 in Argumenten 67
 Kurzformen 78
 logische 120. *Siehe* Logische Operatoren
 Multiplikation 73
 Priorität. *Siehe* Punkt- vor Strichrechnung
 Subtraktion 65

or (logischer Operator) 123

P

Paketmanager
 Definition 327
 pip 327
Paläste bauen 262
Parameter. *Siehe* Argumente
pickle (Modul) 299
 importieren 309
 verwenden 309
pip (Paketmanager) 327
 auf einem Mac verwenden 328
 auf einem Raspberry Pi verwenden 328
 unter Windows verwenden 327
postToChat() 85
Potenzrechnung 76
print() 30, 84
Programm, Definition 35
Programmiersprachen
 Aufbau. *Siehe* Syntax
Prompts (Übersicht) 31
Punktnotation 339
Punkt- vor Strichrechnung 77
Pyramiden bauen mit for-Schleifen 263
Python
 mit Minecraft verbinden 43
 veraltete Version (Fehlermeldung) 33
Python installieren
 auf Mac 17
 auf Windows-PC 4
Python-Shell kennenlernen 27

Q

Quader erzeugen 70, 72

R

randint() (Funktion) 79, 195
random (Modul) 195
range() (Funktion) 256
 lustige Sachen mit 259
Raspberry Pi
 beschränkte Weltgröße 153
 einrichten 25
 Wertebeschränkung bei Variablen 44
read(). *Siehe* Dateien, lesen
readline(). *Siehe* Dateien, lesen, zeilenweise
Rechentricks 78

Refactoring

 Anzeichen für 197
 Definition 195
Regenbogen aus Blöcken 63
return (Schlüsselwort) 200
reversed() (Funktion) 260
Rückgabewerte von Funktionen 200

S

Sammlungen
 Strukturen laden aus 323
 Strukturen speichern in 322
Satz des Pythagoras 186
Säulen bauen 75
 mit Funktion 262
Schrittweite 260
self (Argument) 337
setBlock() 61
setBlocks() 70
setPos() 49
setTilePos() 44
setting() 106, 107
 world_immutable 107
Shell oder Editor verwenden? 31
shelve (Modul) 299
 Daten speichern mit 320
 Elemente hinzufügen/bearbeiten/lesen mit 321
Skalierbarkeit 190
sleep() 51, 101
Sonderzeichen, in Variablen erlaubte 39
Spielerposition 42
 ermitteln 64
 niedrigste/höchste ermitteln 221
 via Flask auf Website anzeigen 331
Spigot
 ausführen (auf Mac) 21
 ausführen (auf PC) 9
 startet nicht (Fehlermeldung) 33
Spigot installieren
 auf Mac 19
 auf Windows-PC 7
Spitznamen für Module 313
sqrt() 186
str() 92
Strecke, zurückgelegte messen 101

Strings

- als Listen behandeln 227
- Chatten mit 83
- Definition 84
- in Integer konvertieren 94
- verknüpfen 90
- Zahlen umwandeln in 91

Strukturen

- laden aus Sammlungen 323
- speichern in Sammlungen 322

Subtraktion

- Operator 65
- Operator (Kurzform) 79

Syntax 37

- Fehlermeldung 38
- unerwarteter Einzug 39

T**Tanzfläche bauen** 177**Tanztee** 161**Teleportation**

- an verschiedene Ziele 163
- Codebeispiel 44
- Codebeispiel debuggen 56
- mit Blöcken kreuz und quer 81
- mit Variablen 35
- verlangsamen mit time-Modul 50
- Ziele beschränken 153

Tippfehler. *Siehe* Debuggen**To-Do-Liste erstellen** 305**Treppen bauen**

258

True (Wert) 105**try-except (Anweisung)** 97

- einbauen 99

Tupel

- Definition 228
- Syntax 228
- Variablen festlegen mit 229
- zurückgeben 232

U**Überlebensmodus**

- umschalten in (auf Mac) 24
- umschalten in (Windows-PC) 13

Überschreiben

- Attribute 362
- Methoden 362

Umstrukturierung

- von Code. *Siehe* Refactoring
- unexpected indent (Fehlermeldung).
Siehe Syntax, unerwarteter Einzug
- uniform() (Funktion) 230

V**Variablen**

- boolesche 105
- Daten speichern in 36
- Definition 36
- erlaubte Sonderzeichen 39
- Float- 48
- globale 211
- globale (Definition) 212
- in Klassen 335. *Siehe* Attribute
- Integer- 40
- lokale 211
- lokale (Definition) 212
- mit Tupeln festlegen 229
- Namensregeln 39
- nicht definiert (Fehlermeldung) 55
- Syntaxregeln 39
- Teleportation 35
- Werte ändern 39
- Werte ändern bei Programmablauf 87
- Wertebeschränkung auf RaspberryPi 44

Variablen und Funktionen

- in Klassen 335

Variablenwerte

- über Funktionen steuern 200

Vererbung

- Grundlagen 356
- von Klassen 357

Vergleichsoperatoren. *Siehe* Boolesche Operatoren**W****Warum programmieren?** xi**Werte**

- in dreidimensionalen Listen aufrufen 290
- in zweidimensionalen Listen aufrufen 276
- zurückgeben mit Methoden 347
- while-else-Anweisungen 184
- while-Schleifen
- Definition 161
- endlose 170
- in Funktionen 204, 208

- mit break beenden 182
- mit if-Anweisungen verschachtelte 179
- mit Zählervariablen steuern 166
- steuern mit booleschen Operatoren 172
- und boolesche Operatoren 176
- und == (ist gleich-Operator) 173
- Unterschied zur if-Anweisung 161
- Wertebereich prüfen 176
- Wiederverwendung von Code 189, 335
- Windows-PC einrichten 2
- world_immutable (Option). *Siehe* setting()
- Wörterbücher 217
 - auf Elemente zugreifen 245
 - Definition 244
 - Elemente ändern/hinzufügen 248
 - Elemente löschen 249
 - for-Schleifen über 265
 - Integer/Floats als Schlüssel 245
- write(). *Siehe* Dateien, schreiben und speichern in

Z

- Zahlen in Strings umwandeln 91
- Zählervariablen, Schleifen steuern mit 166
- Zeilenwechsel in Argumenten 199
- Zufallszahlen 79