

Stephan Elter

Inkl.
Downloads

Mit Syntax-
Highlighting!

Schrödinger programmiert Python

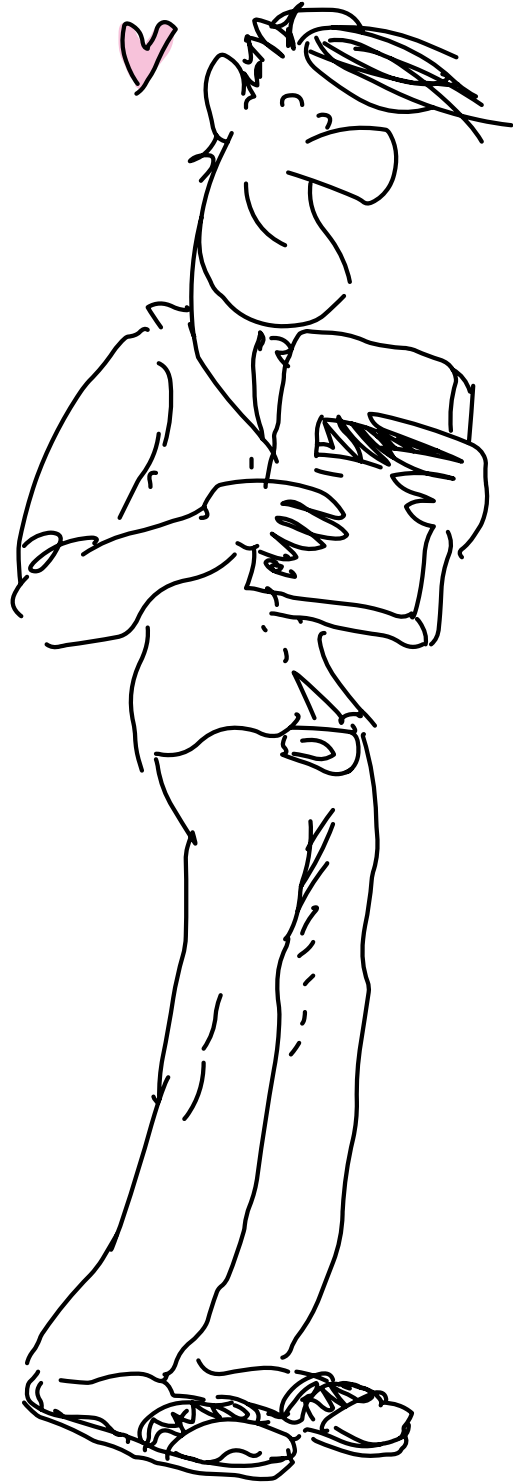
Das etwas andere Fachbuch

☞ Von „Hallo Welt“ bis
KI bis Data Science

☞ Durchblicken, mitmachen
und genießen!

☞ Mit umwerfenden Beispielen,
fantastisch illustriert

 Rheinwerk
Computing



MIT TALENT, KATZEN-
PHOBIE UND LÄSSIGEM
SCHUHWERK BESTACH
SCHRÖDINGER DIE
RHEINWERK-JURY.
FÜR IHN GEHT JETZT
EIN TRAUM IN ERFÜLLUNG.

Liebe Leserin, lieber Leser,

Sie haben gewählt:

Python!

Und Sie haben richtig gewählt.
Oder zumindest wollen wir Sie nicht vom Gegenteil überzeugen.

Python ist toll.

Ganz besonders dann, wenn man als Neuling das erste Mal mit gezückter Machete in den Programmiersdschungel vordringt. Vielleicht stecken Sie aber auch schon mittendrin, im Urwald der Programmierung, und suchen ein neues Werkzeug, mit dem Sie den lästigen Lianen und dem penetranten Ungeziefer (naja, den Bugs halt) die Stirn bieten können. Auch dann ist Python was für Sie.

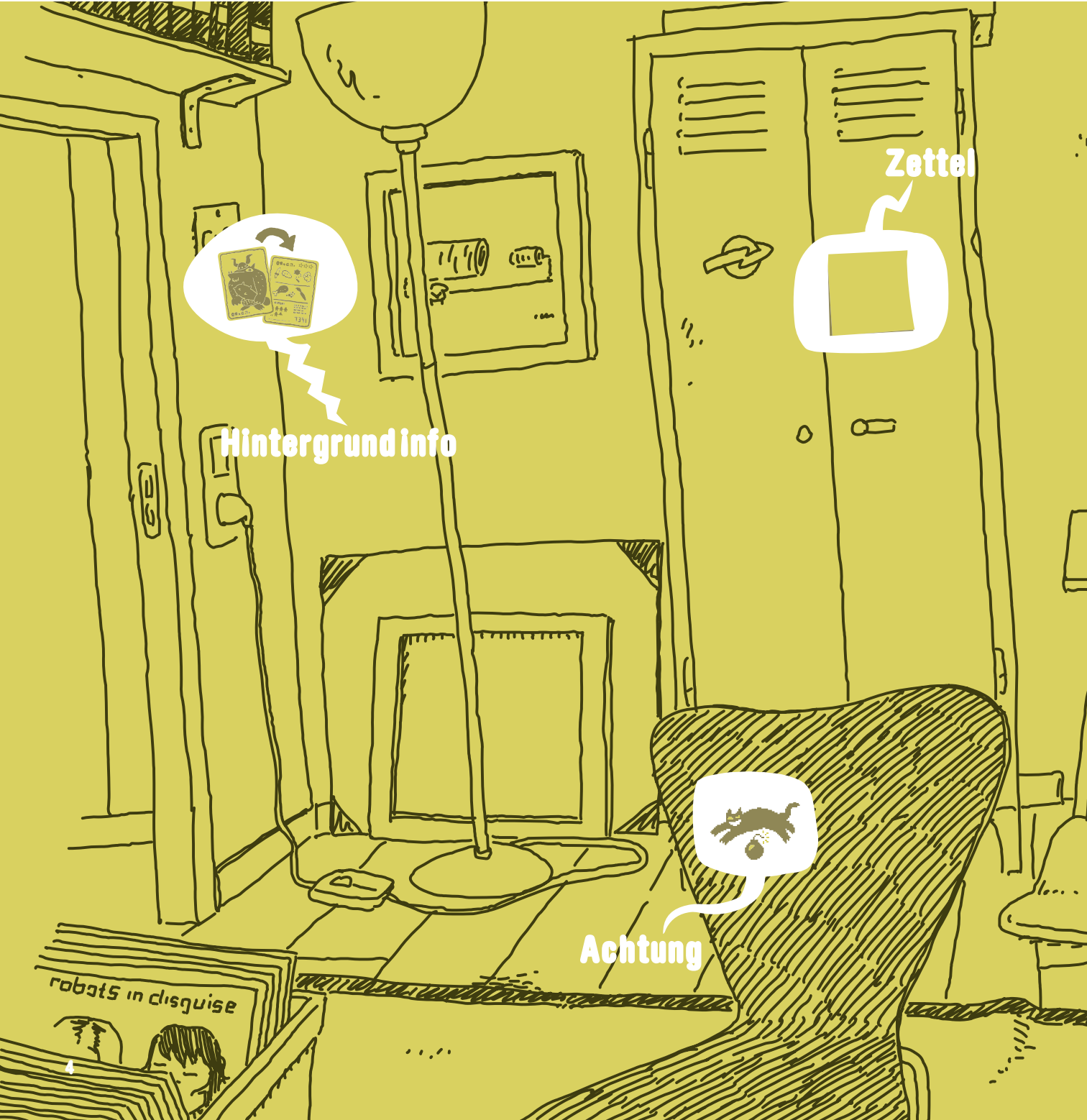
Aber jetzt schlagen Sie erstmal Ihr Lager auf. Denn bevor Sie sich Hals über Kopf in die Büsche schlagen, wollen wir Ihnen Ihre Expeditionstruppe vorstellen. Zum einen ist da **Stephan**, unser Autor und Ihr Expeditionsführer, der den Python-Dschungel wie seine Westentasche kennt. Er zeigt Ihnen alle Trampelpfade, Survival-Tipps und natürlich alle Werkzeuge, die Sie auf Ihrem Weg zum Python-Profi benötigen.

Und dann ist da natürlich noch **Schrödinger**. Schrödinger begleitet Sie auf Ihrer Expedition, aber auch wenn er selbst hin und wieder die Machete schwingt, nimmt er Ihnen das Lernen nicht ab. Ehrlich gesagt denkt er nicht einmal daran. Das wäre auch sowieso zu schade, denn Sie würden fantastische Übungen verpassen und womöglich gar die Illustrationen. Aber Spaß haben werden Sie mit Schrödinger bestimmt, und ein paar schlaue Fragen hat er obendrein parat – vielleicht sogar genau die Fragen, die Sie selbst gerade stellen wollten.

Diese Expedition wurde von einem Expertenteam möglich gemacht, das Ihnen nicht in den Dschungel folgt, aber viele nützliche Hinweise hinterlassen hat. Code wurde eingefärbt, Pfeile und Wegweiser wurden an Bäume genagelt, und hin und wieder baumeln Lösungen kopfüber in den Lianen, damit Sie nicht zu früh spinxen.

Na, dann wollen wir Sie nicht länger aufhalten. Treten Sie nicht auf Schlangen und **viel Erfolg im Dschungel!**

Schrödingers Büro



Hintergrund info



Zettel



Achtung



Die nötige Theorie, viele Hinweise und Tipps

15:20

Begriffsdefinition



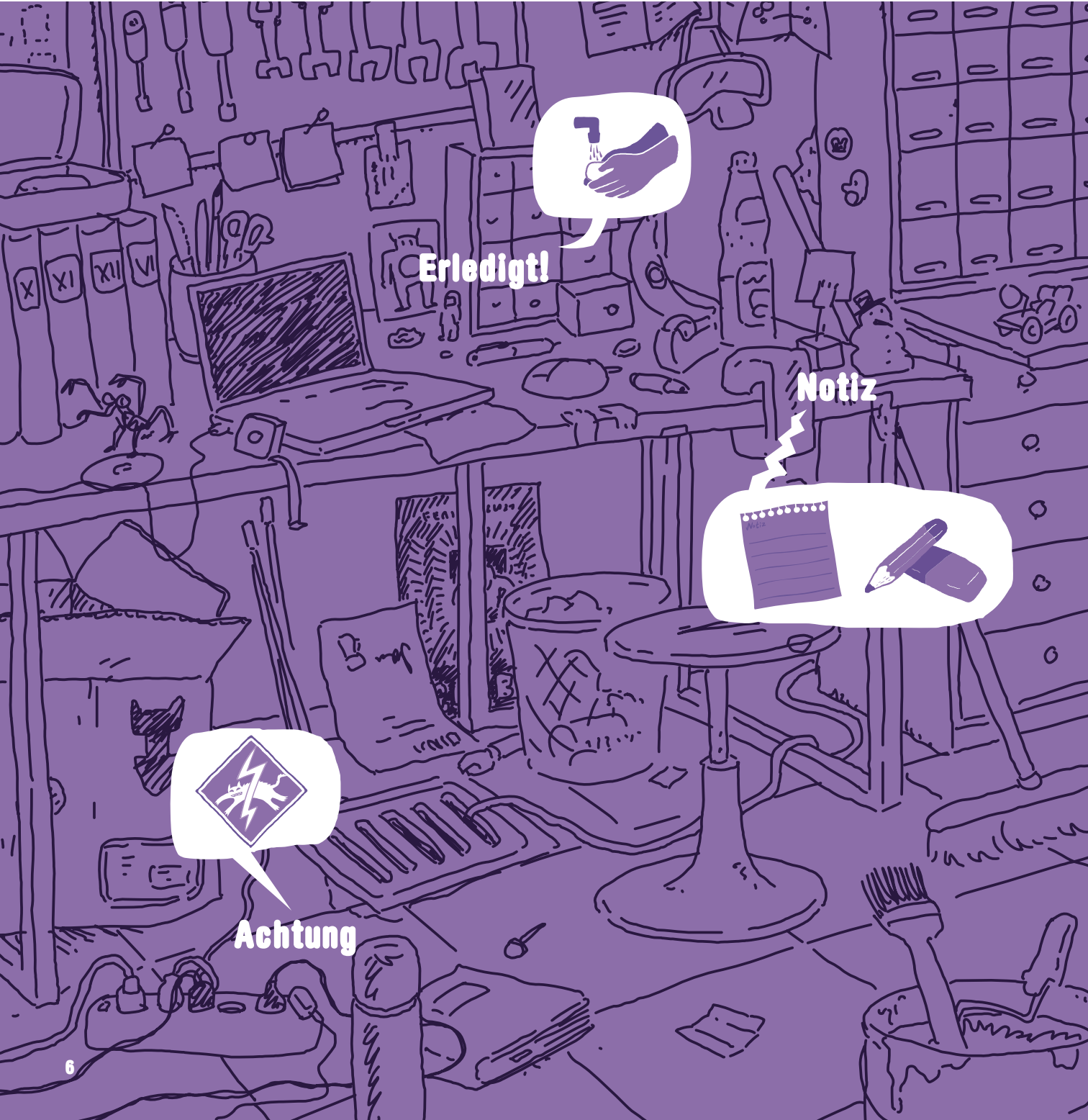
Falscher Code

X



Belohnung

Schrödingers Werkstatt



Erledigt!

Notiz



Achtung

Unmengen von Code, der ergänzt, verbessert und repariert werden will



Code bearbeiten

Einfache Aufgabe

Schwierige Aufgabe

Schrödingers Wohnzimmer

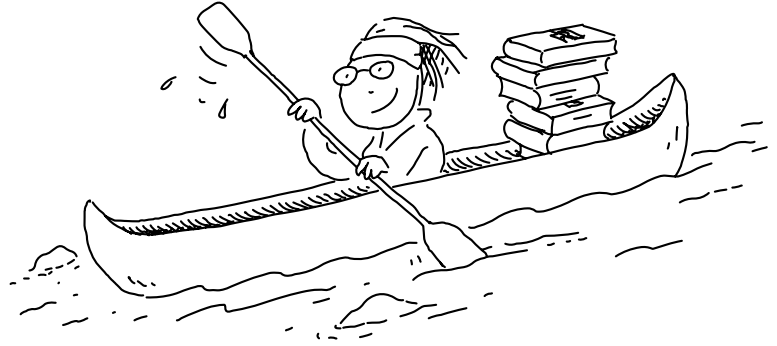


Viel Kaffee, Übungen und die verdienten Pausen



Als Kanutin umschifft Almut Stromschnellen mit notfalls nur einem Paddel. Eine Fähigkeit, die sie zur Fachbuchlektorin prädestiniert.

Almut Poll, LEKTORAT



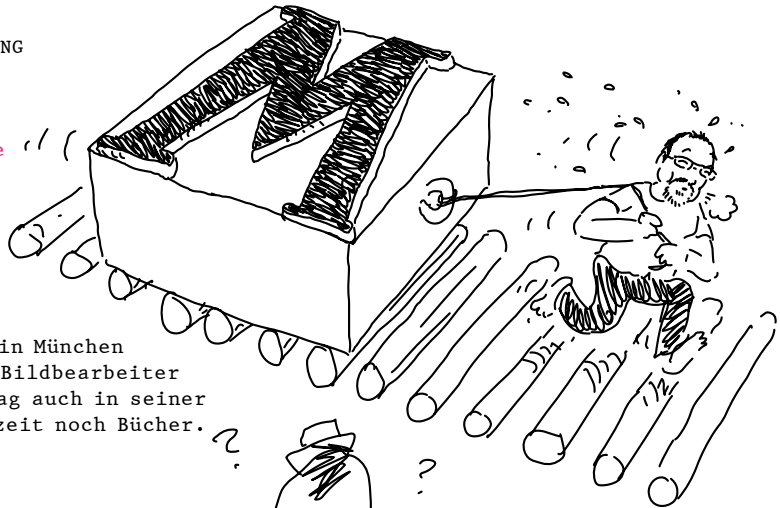
Janina »Sherlock« Brönner. Bei der Buchherstellung sind detektivische Kombinationsgabe und Finesse gefragt. Die Kollegen haben sich allerdings das Pfeiferauchen verboten.

Janina Brönner, HERSTELLUNG



Okay, die Baumstämme hatten wir schon hingelegt, bevor Markus den Satz des Buches übernahm. Und der Rest war, wie man sieht, ein Kinderspiel, gell?

Markus Miller lebt und arbeitet in München als selbstständiger Setzer, Bildbearbeiter und Reinzeichner und mag auch in seiner Freizeit noch Bücher.



Schon zu Schulzeiten zeichnete Leo am liebsten die Bücher voll, von denen er am wenigsten kapierte.

Seit er weiß, dass man das auch gegen Bezahlung machen kann, kapiert er gar nichts mehr.



Leo Leowald lebt und arbeitet in Köln als freiberuflicher Illustrator. Er veröffentlicht unter anderem in *titanic*, *jungle world* und bei *reprodukt* und zeichnet seit 2004 den Webcomic www.zwarwald.de.

Andreas' zweite Leidenschaft neben der Buchgestaltung ist kochen. Wie auch immer: Hauptsache rare – VERY RARE!

Andreas Tetzlaff ist selbstständiger Buchgestalter in Köln. Er arbeitet normalerweise für Kunstbuchverlage – dass ausgerechnet ein IT-Fachbuch ihn vor künstlerische Herausforderungen stellt, hätte er sich vorher nicht träumen lassen ...





Patricia engagiert sich ehrenamtlich bei einer Tierschutz-Hundepflegestelle. Ihr anderes Lieblingstier kann sie derzeit leider nur in RPGs und Fantasyromanen ausführen.

Patricia Schiewald, LEKTORAT

Annette ist von Haus aus Archäologin, da ist es nur ein kleiner Schritt bis zum Lektorat, und der Vorteil ist: Bei Schrödinger und Co. findet sie immer was.

Annette Lennartz ist freiberufliche Lektorin in Bonn. Für Schrödinger hat sie immer eine offene Tür. Privat schätzt sie augenzwinkernde und gruselige Geschichten oder bastelt an filigranen Schiffsmodellen.

KORRIGIERT VON: Annette Lennartz



Felix hat seine Liebe fürs Programmieren in die Wiege gelegt bekommen und sie genutzt, um den Code im Buch von vorne bis hinten durchzutesten.

CODE GETESTET VON: Felix Elter



Torsten träumt von Algorithmen, meist in C++. So ist es kein Wunder, dass er ein Diplom in Informatik besitzt. Er hat dieses Buch auf den Kopf gestellt, kräftig durchgeschüttelt und sich zuweilen mit Schrödinger angelegt. Um die Schleifen aus dem Kopf zu kriegen, verbringt er seine freie Zeit mit Fotografieren.

BEGUTACHTET VON: Torsten T. Will

Einbandgestaltung:
Andreas Tetzlaff und Leo Leowald

FÜR DIE, DIE ES GENAU WISSEN WOLLEN

Dieses Buch wurde gesetzt aus unzähligen Schriften (u.a. aus der WIMBY von Evert Ypma: Danke, Evert!), Tonnen an Illustrationen und anderen komischen Zeichen, die alle Beteiligten in den Wahnsinn trieben.

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.dnb.de> abrufbar.

ISBN 978-3-8362-6747-2

© Rheinwerk Verlag, Bonn 2021
1. Auflage 2021, 1. Nachdruck 2022



Stephan verbringt gerne Zeit mit seiner Familie am Timmendorfer Strand. Außerdem liebt er analoge Gesellschaftsspiele und seinen Retro-Mops. Zwei Leidenschaften, die sich selten verbinden lassen.

GESCHRIEBEN VON: Stephan Elter

Das vorliegende Werk ist in all seinen Teilen urheberrechtlich geschützt. Alle Rechte vorbehalten, insbesondere das Recht der Übersetzung, des Vortrags, der Reproduktion, der Vervielfältigung auf fotomechanischen oder anderen Wegen und der Speicherung in elektronischen Medien. Ungeachtet der Sorgfalt, die auf die Erstellung von Text, Abbildungen und Programmen verwendet wurde, können weder Verlag noch Autor, Herausgeber oder Übersetzer für mögliche Fehler und deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen. Die in diesem Werk wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. können auch ohne besondere Kennzeichnung Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

INHALTSVERZEICHNIS

Vorwort	22
---------------	----

Kapitel 1: Schrödinger startet durch – mit Python!

Python, erste schnelle Schritte

Seite 23

Die Programmiersprache Python!	25	Lustiges Konvertieren – was Python zu was macht	51
Das Zen of Python und die Sache mit den PEPs ...	26	Die Funktion »str()« – verwandelt alles in Text ...	51
Python, ein erstes »Hallo Welt«	27	Die Funktion »int()« – ganze Zahlen	52
Fingerübungen mit »print«	31	Die Funktion »float()« – Fließkomma mit Punkt ...	53
Hallo Welt in (einzeiligen) Variationen	32	Die Funktion »bool()« – Wahrheit oder Pflicht ...	53
Wir müssen reden: Du und deine Variablen	35	Die Funktion »bool()« – Wahrheit oder Pflicht ...	53
Variablen – was geht?		Was ist das denn für ein Typ – »type()«	54
Und was ist voll krass korrekt?	39	Probier's doch mal aus	56
Die Sache mit den (Daten-)Typen	43	Und was ist noch wichtig?	57
Diese Datentypen sind für dich da!	47	Syntax, Variablen, Datentypen und dynamische Typisierung	58
Über den richtigen Kamm scheren – Datentypen konvertieren	48		

Kapitel 2: Ein Dinkel macht noch keinen Korn

Syntax, Kommentar und guter Stil

Seite 59

Eingabe, Berechnung und eine Ausgabe	61	Ganz wichtig: Die Bedingung der Schleife	73
Zahlen mit Komma – statt mit dem Punkt	64	Ein schneller Blick auf die Vergleiche	74
Die Zeit der Abrechnung ist gekommen	66	Schleifen binden – der erste eigene Prototyp	76
Damit kannst du rechnen – die Grundrechenarten	67	Entrückt – Einrückungen statt Klammern	79
Wie wär's mit einem Rabatt?	69	Fehlerschau – nicht alles, was gefällt, ist auch erlaubt	82
Du und deine Kontrollstruktur	72	Weiter im (Kassen-)Programm – jetzt mit Schleife	84
»while« – Schleifen binden leicht gemacht	72		

... es gibt keine blöden Kommentare!	87	Einfachere Bedingungen!	93
Die »if«-Anweisung – wenn schon, denn schon ...	88	Kürzeres »else if« mit »elif«	94
Zeit für Entscheidungen – dein erstes »if«	89	»while« – The Python-Way mit »break«	97
Wenn, dann und ansonsten?	91	Was hast du gelernt? Was haben wir gemacht? ...	98
Wenn schon, denn schon und auch noch »else«	92		

Kapitel 3: Arbeitszimmer und Werkbänke

Funktionen und Rückgabewerte

Seite 99

Funktionen	100	Weiter mit der Funktion in Kastenform	118
Der Name der Funktion	101	Funktionen – die Super-Fehlervermeider	120
Falsche oder richtige Funktionsnamen?	102	The return of the living values	124
Eigene Funktionen – der erste Prototyp	103	Der »Wertekorrigierer« – Steuerung mit »return«	126
Deine Funktion im Programmablauf	105	Mehrere Werte mit »return« zurückgeben	127
Deine erste eigene Funktion	106	Du und deine Funktion – ein paar Beispiele	128
Dynamik dank Parameter?	107	Spiel's noch mal Sam – die Rekursion!	129
Zeit zum Ausprobieren! Einmal »lecker Funktion« mit extra Parametern!	109	Die dunkle Seite der Funktionen – »local« und »global«	130
Mehr Freiheit dank der Standardwerte	110	Lokal vs. global – was geht?	131
Standardwerte – Probieren geht über Studieren ...	111	»global« – die dunkle Seite der Macht	132
Schlüssel-Wert-Paare – alles richtig zugeordnet ...	113	Große Aufgabe dank Funktionen ganz klein – Kassenprogramm reloaded	133
Wie könnte so eine Funktion »spam« aussehen?	114	Was hast du gelernt? Was haben wir gemacht? ...	136
Besser als jede Doku – Docstrings	116		

Kapitel 4: Listen, Tupel, Sets und Dictionaries

... alle Daten sind schon da!

Seite 137

Du und deine Liste	138	Das Hornberger Elfmeterschießen – erst mit »for« und dann mit »range«	147
Der 1. FC Dinkel und andere Mannschaften	141	Index und Wert ausgeben mit »enumerate«	149
Teile und herrsche – Teile von Listen	143	Wenn Besuch kommt – eine Mannschaft mehr ...	150
Der neue Star auf dem Feld – die »for«-Schleife ...	144	Die Sache mit den Methoden	151
Wie für »for« gemacht – »range«	145		

Eine ganze Liste anhängen mit »extend«	152	Du, die Liste und deine Vereine	162
Einzelne Werte einfügen mit »insert«	153	Ziemlich einmalig – das Set	165
Alles wieder andersrum – mit »reverse«	154	Tupel – in Stein gemeißelte Listen	167
Geordnet und wohlsortiert, dank »sort«	154	Moment mal, wie war das mit »immutable«?	169
Sortieren nach Schrödingers Gnaden	156	Du und deine Tore – gut gespeichert im Dictionary	170
Ein Element aus der Liste holen und entfernen – »pop«	158	Werte auslesen mit »get«	171
Ja, wo is' er denn – »index«	159	Zeigt her eure Werte – alle Werte eines Dictionarys ausgeben	172
Einfach nur löschen – »remove«	159	Was hast du gelernt? Was haben wir gemacht? ...	174
Alle Werte sind schon da – »count« und »in«	160		

Kapitel 5: Text, Strings und Abenteurer

Texte verändern und bearbeiten

Seite 175

Hilfreich wie eine Machete im Dschungel – der Backslash »\«	178	Der Vollständigkeit halber – formatiert mit »%« ...	196
Texte zusammenfügen	180	Alle Wörter großgeschrieben – »title«	197
Übung macht den Meister	182	Wie oft ist das noch da – »count«	197
Die erste Aufgabe – Umbenennen von Dateinamen	183	Wo ist der Schatz – suchen mit »find« und »rfind«	198
Die richtige Ausrüstung für den Textdschungel – hilfreiche Methoden	186	So machst du aus Listen Texte – »join«	199
Alle Funktionalitäten in einer Funktion	188	Ist das eigentlich 'ne Zahl – »isnumeric«	200
Strings schöner ausgeben – mehr als nur Kosmetik	191	Buchstaben und Zahlen – »isalnum«	201
Variable im Text – praktisch und einfach mit »{}«	194	Sind da nur Buchstaben – »isalpha«	201
		Von der grauen Theorie zum praktischen Nutzen	202
		Method Chaining – Methoden in Reihe	205
		Was hast du gelernt? Was haben wir gemacht? ...	206

Kapitel 6: Von Käfern und anderen Fehlern

Nur kein Fehler ist ein guter Fehler

Seite 207

Fehlerbehandlung mit »try« und »except«	210	Kenne deine Gegner – unterschiedlichen Fehlerarten	215
Bombensichere Eingaben dank »try« und »except«	211	Fehlerbehandlungen im Eigenbau	217

Mit deinem Fehler auf du und du	218	Debuggen (nicht nur) mit Thonny	226
Schönere Fehlerbehandlung mit »else« und »finally«	222	Du, der Debugger und die Breakpoints	228
Fehler geschüttelt, nicht gerührt	223	Gezielte Fehlerjagd mit Breakpoints	232
Zu guter Letzt – »finally«	225	Was hast du gelernt? Was haben wir gemacht? ...	234

Kapitel 7: Die Module spielen verrückt

Die Standardbibliothek und noch viel mehr

Seite 235

Schnelle Infos dank Docstring und »help«	239	Module wie du und ich – Python Standard Library	256
Ein Modul namens »dateiname«	241	Was für ein Zufall – »random«	258
Modul und trotzdem Programm	243	Wo liegt der Unterschied?	259
Das doppelt gemoppelte Modul für den direkten Test	246	Import mit »from«, »import« und »as« – gezielt und direkt	260
Mächtig vielseitig – globale Variablen (nicht nur) in Modulen	247	Wie viel Zufall steckt in Zufallszahlen?	262
Die Methode »dateiname« – nur noch flexibler ...	250	Mehr als nur ein Import – zwei Importe	263
Schrecklich lange Modulnamen – »as«	252	Was hast du gelernt? Was haben wir gemacht? ...	264
Das Modul einer Variablen zuweisen	253		

Kapitel 8: Von Klassen, Objekten und alten Griechen

Objektorientierte Programmierung

Seite 265

Die gute, alte Softwarekrise	266	Die erste Klasse am Stück – gleich mal etwas reloaded	280
Retter gesucht? Retter gefunden: OOP!	266	Das erste eigene Objekt	281
Ganz konkret – die Sache mit Klassen und Objekten	269	Orakel reloaded – das Attribut ändern	285
Von der ersten Klasse zum ersten Objekt	270	Vorsicht beim Zugriff auf Attribute!	286
Alles auf Anfang – die Methode »__init__«	271	Die Sache mit den Parametern	288
Dein erstes Attribut	272	Vertrauen ist gut, Kontrolle besser	289
Es gibt auch ganz schnöde Variablen	274	Übergebene Werte sind gefährlich!	290
Mehr Infos dank Docstring	275	Das ist die Stunde der Methode »property«!	291
Das Orakel von Delphi	277	Besser als nur Setter und Getter	293

»property« und Orakel – das passt!	297	Besser als recyceln – neue Klasse aus alter Klasse	308
... der durchaus seltsame klingende, aber ziemlich coole »@property«-Dekorator	300	Super Sache dieses »super«	311
»private« und »protected« – aber gar nicht so ganz	302	Statische Attribute und Methoden	313
Wiederverwendbarkeit und Vererbung	306	Was hast du gelernt? Was haben wir gemacht? ...	317

Kapitel 9: Höchste Zeit für Datum, Zeit und Zeitangaben

Schrödingers Zeitmaschine

Seite 319

Du und deine Zeitmaschine	320	Gestatten, »datetime«, aus dem Hause »datetime«	338
Welcher Tag ist heute? Welches Jahr!? – »date«	320	Besser als jedes Orakel – Zeit lesen mit »strptime«	339
Bastel mal ein schickes Datum	322	Datum und Zeit finden – so ganz in der Praxis ...	341
Tag, Monat, Jahr mit Platzhaltern in Form bringen	323	Ganz großes Kino – Unixtime und The Epoch ...	343
Einmal Datum, geschüttelt – nicht gerührt	325	Wann war die letzte Änderung?	345
Es wird Zeit, die Zeit zu ändern	327	Noch ein Wort zu »timedelta« – rechne mit der Zeit	347
Was von der ganzen Zeit noch übrig bleibt	329	... rette Weihnachten mit »timedelta«	348
Stunden, Minuten und Sekunden mit »time«	331	Was hast du gelernt? Was haben wir gemacht? ...	350
Ist noch Zeit für einen Dinkelkaffee?	333		
Es ist an der Zeit, die Zeit zu formatieren!	335		

Kapitel 10: Vom wichtigen Umgang mit Daten, Dateien und Ordern

Endlich in Stein gemeißelt

Seite 351

Wohin mit all den Daten?	352	Du und dein Regal – ein paar hilfreiche Informationen	357
Das Regal auf der Gurke	353	Mit Netz und doppeltem Boden	358
Zeit zum Lesen	355	Du und deine Textdatei – schreiben und lesen ...	360
Zeit für ein bisschen Serialisierung	356		

Und es geht noch kürzer – mit »with«!	362	Kopieren oder nicht kopieren,	
Du und deine Textdatei	363	das ist hier die Frage	375
Listen und Zeilenumbrüche schreiben	365	Eindeutige Hash-Werte für Vergleiche	377
Zeilenweises Lesen	367	Und nicht vergessen: Verschieben und	
Im Dschungel der Ordner und Dateien	368	Löschen	379
Halt mal die Machete – Überleben im		Was hast du gelernt? Was haben wir gemacht? ...	380
Ordnerdschungel	372		

Kapitel 11: Zufallszahlen, Matrizen und Arrays

Ein klein bisschen Mathematik, die du wirklich gebrauchen kannst

Seite 381

Du, die Zufallszahlen und NumPy	383	Bau mal ein Array	398
Auf dem Weg zum Millionär –		Weniger selbst arbeiten – Arrays mit »arange« ...	399
ein Lottoprogramm	386	Die Sache mit den mehrdimensionalen Arrays ...	400
Andere Verteilungen bei den Zufallszahlen	388	Bastelarbeiten mit Arrays	402
Ganz normalverteilte Werte	391	Rechnen mit Arrays	404
Die Sache mit den Arrays	393	Was hast du gelernt? Was haben wir gemacht? ...	406
Ein paar schnelle Berechnungen	395		

Kapitel 12: Grafische Oberflächen

Buttons, GUI und Layout-Manager

Seite 407

Der Layout-Manager »pack« und die		Rechnen mit dem Schrödinator und die Sache	
Sache mit den Frames	411	mit den »tkinter«-Variablen	427
Mit deinem Fenster auf du und du	414	Ganz kurz noch schönere Schriften	429
Ein bisschen Kosmetik mit schöneren Elementen	419	Ereignisse im objektorientierten Fenster	430
Die Sache mit dem Lambda – nicht nur für GUIs	421	Das Schrödinger-Zeichenprogramm	438
Button mit Parametern – ganz einfach dank		Was hast du gelernt? Was haben wir gemacht? ...	440
Lambda	423		

Kapitel 13: Von Daten, Datenbanken und SQL

Das relationale Datenbankmodell

Seite 441

Retter und Held gesucht: Datenbankprofi	443	Weiter im Programm mit der	
Daten braucht das Land!	451	kontrollierten Eingabe	471
Keine doppelten Sachen – die Sache		Eine Funktion, alles zu speichern	473
mit dem Primärschlüssel	454	Zeit, die Viren zu stoppen –	
Stopp die Viren und Trojaner!	456	die Auswertung der Daten	475
»fetchall«, »fetchmany«, »fetchone« –		Höchste Zeit für schnelle Auswertungen	479
alle, viele, einer	462	Finale – Ändern mit UPDATE	486
Finde die richtige Abwehrstrategie!	465	Die Sache mit den Normalformen	491
Die Sache mit dem WHERE	467	Mehr Leistung mit dem Index	496
Schönere Datenbankverbindung mit »with«	470	Was hast du gelernt? Was haben wir gemacht? ...	498

Kapitel 14: Hast du mal einen Chart für mich?

Zahlen und Daten im Überfluss

Seite 499

Drei Kurven sollt ihr sein	505	Sahne, Frucht und Dinkel – was ist	
Zeit für die erste Auswertung	507	am beliebtesten?	520
Nicht nur für Charts: Schlaue Listen		Mehr als nur ein Fenster – die Sache mit	
mit List Comprehension	509	den »subplots«	523
Kleine, schlaue Listen selbst gemacht	512	Noch mehr Torten – das Kuchendiagramm	526
Da geht noch was – noch ein »if« und		Du und deine Normalverteilung – von ziemlich	
auch ein »else«	518	eindimensional bis schick in 3D	529
		Was hast du gelernt? Was haben wir gemacht? ...	532



Kapitel 15: Daten, Statistik, Data Science und künstliche Intelligenz

Wenn der eigene Kopf schon raucht

Seite 533

Richtige Ergebnisse – mal ganz ohne Formel	537	Zeit für noch mehr Lernen lassen	544
Und jetzt alles mit echter KI	540	Virenerkennung mit dem	
(Trainings-)Daten braucht das Land	541	RandomForestClassifier	547
Nicht für die Schule lernt die KI	541	Daten polieren – mit der richtigen Strategie!	550
Zeit, das Orakel zu befragen	542	Mittelwert und Median als Strategie	554
Die zweite Zahlenreihe	543	Was hast du gelernt? Was haben wir gemacht?	556

Kapitel 16: Datenaustausch mit CSV und JSON

Daten schreiben – Daten lesen

Seite 557

Ein paar Einstellungen und der richtige Dialekt ...	563	Einmal JSON und zurück	571
Da geht auch was mit Dictionaries	565	Auch JSON will gelesen sein	573
Zeit zu lesen	567	Was hast du gelernt? Was haben wir gemacht?	574
Die Sache mit JSON	570		

Kapitel 17: Reguläre Ausdrücke

Das Schweizer Messer der Textverarbeitung

Seite 575

Eine Suche – ganz klassisch	577	Besser als jedes Orakel – das	
Mit regulären Ausdrücken ist das		Matchobjekt auslesen	596
kein Problem!	580	Finde die Kennung – probieren geht über	
Nur mal kurz: Ein paar Flaggen	585	studieren	597
Eine Funktion zur passgenauen Wortsuche	587	Selber basteln – eigene Zeichenklassen	600
Vordefinierte Zeichenklassen, ein Punkt und		Schöneres Datum dank eigener Zeichenklassen	603
viele Beispiele	591	Quantifier – wie oft oder vielleicht auch	
Ein Beispiel – die Suche nach Datum und Zeit	594	gar nicht?	606

Mehr als ein Treffer – »findall« und »finditer«	609	Ein Anfang, ein Ende und ein paar	
Mach mal was mit Quantifiern	610	Empfehlungen	616
Ganz kurz: Quantifier mit »?«, »*« und »+«	613	Suchgruppen – mehr als nur ein Zeichen	619
Ein (fast) gültiges Kennwort	614	Die Sache mit »compile«	621
		Was hast du gelernt? Was haben wir gemacht?	622

Anhang: Die verlorenen Kapitel

Für alles, was (noch) nicht passend ist

Seite 623

Anhang A: Installation von Python unter Windows – Ein neues Zuhause für Python	625	Anhang D: Die All-in-one-Entwicklungsumgebung – Wundertüte Thonny	643
Anhang B: Die Python-Shell – Schnell mal machen, und das voll interaktiv	631	Anhang E: DB Browser for SQLite – Datenbanken leicht gemacht	653
Anhang C: PEP 20 –The Zen of Python	639		

Index	663
-------	-----



**Für
Andrea, Alva
und Felix**



Vorwort

Hallo Schrödinger! Ich darf dich doch Schrödinger nennen, oder? Ich habe gehört, dass du die Programmiersprache Python lernen möchtest.



Das ist eine tolle Idee!

Python ist nämlich eine ganz besondere Sprache. Sie ist sehr gut für den Einstieg in die Programmierung geeignet. Allein die Syntax ist einfach, aber dennoch sehr mächtig und ausdrucksstark (nennen wir es mal so).

Wenn du möchtest, kann ich dir gerne zeigen, wie du mit Python programmieren kannst – und zwar so richtig programmieren, mit allem Drum und Dran. Ich werde dir nicht alles zeigen (können), was es gibt, denn Python ist tatsächlich sehr umfangreich und bietet auch dem erfahrenen Entwickler noch sehr viel.

Stattdessen werde ich versuchen, dir **das** zu zeigen, was du brauchst, um richtige, echte Programme zu schreiben. Dabei werde ich den Schwerpunkt auf praktische Dinge legen, weniger auf (scheinbar) vollständige Auflistungen von Funktionen und Methoden, die du in der Praxis dann doch nicht brauchst.

Aber vielleicht sollte ich mich kurz vorstellen. Mein Name ist Stephan und ich bin Entwickler. Vor nun (unglaublichen) 40 Jahren habe ich angefangen zu programmieren, als mir mein Onkel einen der ersten programmierbaren Taschenrechner in die Hand gedrückt hat, mit den Worten: »Ja, mach doch mal was damit!« Genauso möchte ich dir Python an die Hand geben, damit du damit die seltsamsten, interessantesten und spannendsten Dinge programmieren kannst.

Ach, ja:

Ich lebe mit meiner Familie – samt unserer Mopsdame – in einem kleinen Städtchen vor den Toren Hamburgs und arbeite als Entwickler bei einem großen Verlag mit dem (aller-)besten Team und mit PHP, Java, JavaScript und natürlich auch mit Python.

—EINS—

Python,
erste schnelle
Schritte

Schrödinger startet durch — mit Python!

Die ersten Schritte in Python sind schnell gemacht, dabei geht es nicht nur um die Syntax — du wirst auch gleich den Zen of Python kennenlernen, in dem (nicht ganz bierernst) ein paar Regeln und Empfehlungen festgelegt sind. Und mit ein paar Variablen und dem Befehl »print« legst du gleich schon so richtig los.

Sag mal, Schrödinger, ...



Hast du nicht deiner Freundin versprochen, ihr bei der Buchhaltung zu helfen? Für den Kohl- und Dinkelversand? Wolltest du ihr nicht ein Kassensystem für den Flohmarkt schreiben? Der Flohmarkt findet nächstes Wochenende statt!

Jep, da war was ...

Und was ist mit deinen Kumpels vom Fußballverein, deren Vereinsdaten auf Vordermann gebracht werden sollten? In ein paar Wochen ist das große Fußballturnier mit dem beliebten Hornberger Elfmeterschießen.

Du wolltest doch ein Auswertungsprogramm schreiben? Mit einer Verwaltung der Schützen und einer aktuellen Bestenliste ...

Da war definitiv was ...

Und sag mal, ... hattest du nicht deinem Bruder versprochen, ihm bei der Aufbereitung der Bilder und Notizen seiner letzten Amazonas-Expedition zu helfen? Du weißt schon, um die Funde zu erfassen und als Katalog aufzubereiten. Wollte er nicht in zwei Wochen wieder hier sein und loslegen?

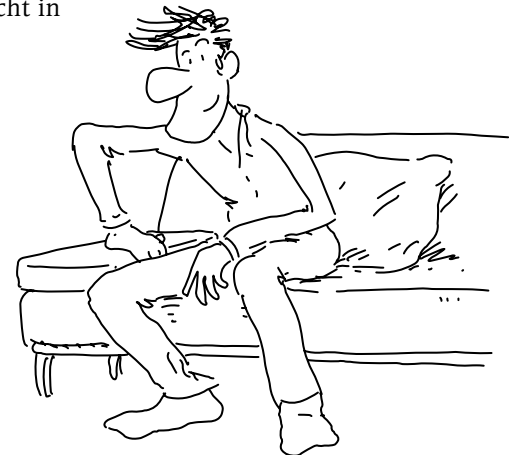
Okay, okay, ich fang ja schon an!

Ähm, Stephan?

Ja?

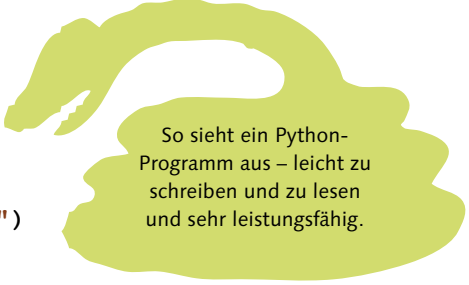
Die Programmiersprache ...

Schon gut. Ich zeige dir ...



Die Programmiersprache Python!

```
eingabe = input("Dein Name?")
if eingabe == "Schrödinger":
    print("Schön, dass du da bist! Ich bin Python.")
```



So sieht ein Python-Programm aus – leicht zu schreiben und zu lesen und sehr leistungsfähig.


Es gibt viele Programmiersprachen, gute, ausgereifte Programmiersprachen, die sich seit vielen Jahren und auf vielen Gebieten bewährt haben.

Aber ... ich zeige dir mit Python eine Sprache, die du **rasch** erlernen kannst. Sie ist **einfach** zu schreiben und genauso einfach zu lesen, sodass du schon als Anfänger gut lesbare Programme schreibst, die nicht in einem riesigen Tohuwabohu enden. Es ist eine Sprache, in der sich auch Entwickler anderer Sprachen nach kurzer Einarbeitungszeit zurechtfinden, ohne lange die syntaktische Schulbank drücken zu müssen.

Es ist eine Programmiersprache, die **universell** einsetzbar ist und die nicht nur für das Web oder ganz bestimmte Anwendungsfälle gedacht ist – **plattformübergreifend** für Windows, Linux und Mac verfügbar.

Objektorientiert und funktional ist Python auch. Und außerdem kommt Python ohne die ganzen Klammern (besonders die geschweiften, die so schlecht zu tippen sind) und ohne Semikolon am Ende jeder Anweisung aus.

Klingt gut.



Python hat alles Wichtige dabei, also quasi »Batteries included«. Python ist leicht zu **erweitern** – durch Module, die du selbst schreiben kannst, genauso wie durch professionelle Frameworks und fertige Bibliotheken, und das für alle erdenklichen Bereiche wie Bildbearbeitung, numerische Analysen, künstliche Intelligenz oder Datenvisualisierung – eben voll und ganz **wissenschaftlich**.

Mit Python kannst du **schnell** entwickeln – sogar sehr schnell.

Trotzdem ist Python nicht etwa auf kleine, einfache Programme beschränkt.



[Zettel]

Python ist eine Sprache, die auch die **Big Player** verwenden, wie Google, Microsoft oder Amazon. Und Python findet in **Wissenschaft** und **Forschung** Anerkennung.

[Hintergrundinfo]

Python wurde Anfang der 1990er Jahre von dem niederländischen Entwickler **Guido van Rossum** erfunden. Er arbeitete damals an einem Forschungsinstitut für Mathematik und Informatik. Vor Weihnachten hatte er ein paar Tage Zeit und suchte nach einem Nachfolger der dortigen Lehrsprache namens ABC. Diese Programmiersprache war für den Unterricht und fürs Prototyping vorgesehen, hatte aber zahlreiche Einschränkungen und ließ sich kaum erweitern. Da er keine geeignete Sprache fand, schrieb er kurzerhand die erste Version der Programmiersprache Python. Das war der Beginn der steilen Karriere einer bemerkenswerten Programmiersprache.

Das Zen of Python und die Sache mit den PEPs

Was macht Python denn nun so besonders?

Nun, folgende Gründe:

- Python hat das »Zen of Python«, 20 fast magische Regeln, die eine große Rolle in Python spielen und den »Geist« (und den Humor) von Python widerspiegeln – alles festgehalten im Python Developers Guide, in 20 Regeln namens PEP 20.

PEP was?



[Hintergrundinfo]

PEP steht für **Python Enhancement Proposals**, also Verbesserungsvorschläge zu Python. Die bekanntesten sind eben **PEP 20**, das Zen of Python, und **PEP 8**, der Styleguide, mit dem dein Code noch besser, stylicher und professioneller wird. Die PEP 20 findest du übrigens in Anhang C dieses Buches aufgelistet.

- Python ist **weitverbreitet** und hat eine große, aktive Community, von der Python weiterentwickelt wird.
- Python ist leicht zu **erlernen**, da es nur wenige Befehle kennt. Aber keine Sorge, du kannst **alles** damit machen, oft sogar schneller, als du es vielleicht mit anderen Sprachen machen könntest.
- Python ist leicht zu **lesen**, da es auf einige Zeichen verzichtet, die »typisch« für Programmiersprachen, aber eigentlich doch unpraktisch sind: Python kennt zum Beispiel für normale Anweisungen keine geschweiften Klammern (für die man sich auf einer deutschen Tastatur die Finger verbiegen muss) und auch kein »schickes« Semikolon am Ende jeder Anweisung.
- Der Python-Code von Anfängern und Profis ist gleichermaßen gut zu lesen, denn Einrückungen von zusammengehörigen Codeteilen sind fester Bestandteil der Sprache – und korrekt eingerückter Code ist immer besser lesbar.
- Bei der Verwendung von Variablen muss kein Typ (Text, Zahl oder was auch immer) angegeben werden. Python nutzt dafür die sogenannte **dynamische Typisierung** und kümmert sich um alles Notwendige. Ganz praktisch: Eine 1 sieht aus wie eine Zahl, also wird es wohl auch eine Zahl sein, und du sparst dir einiges an Schreibarbeit und Festlegungen. Das ist in Python weniger gefährlich als in manchen anderen Sprachen, denn Python passt genau auf, dass die Typen unterschiedlicher Variablen zueinanderpassen – oder ausdrücklich passend gemacht werden.
- Python hat viele interessante Pakete, Erweiterungen und Bibliotheken.
Für fast alle Bereiche gibt es fertigen Code, den du ganz einfach verwenden kannst.

Und das sind nur ein paar Punkte, die Python ausmachen.

Mehr – und vor allem die oben genannten 19 Punkte des 20 Punkte umfassenden Zen of Python – wirst du in diesem Buch kennenlernen. Jetzt gleich wirst du deine ersten Schritte mit Python machen ...



Moment: 19? 20?!!!

Es gibt laut PEP 20 tatsächlich 20 Regeln (bzw. Empfehlungen). Niedergeschrieben sind aber nur 19. Es gibt zahlreiche Diskussionen im Internet, welches wohl der 20. Grundsatz ist und warum er in dieser Auflistung fehlt. Vermutlich ist das dem schrägen Humor geschuldet, der von der namengebenden Komikertruppe Monty Python auf die Programmiersprache abgefärbt hat.



Ich dachte Python heißt so nach dieser riesigen Schlange?

Okay, lass uns anfangen ...

[Hintergrundinfo]

Auch wenn zwei Schlangen das Logo zieren (was ja wirklich sehr gut aussieht), kommt der Name tatsächlich von der legendären britischen Comedy-Truppe Monty Python, die in den 1970er Jahren das Fernsehprogramm und sogar die Kinos mit ihrem unnachahmlichen britischen Humor heimsuchte.



Python, ein erstes »Hallo Welt«

Wie wäre es mit »Hallo Welt«? Das ist der Klassiker, wenn du eine neue Sprache lernst: Du schreibst als Erstes ein Programm, das nichts anderes tut, als den Text "Hallo Welt" auszugeben.

Nichts leichter als das:

```
mein_text = "Hallo Welt"  
print(mein_text)
```

Natürlich, um es gleich vorwegzunehmen: Ein noch einfacheres

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

würde genauso funktionieren. Aber ich möchte ja, dass du gleich etwas lernst. Also benutze eine **Variable** namens `mein_text` und weise ihr einen Wert zu – natürlich den Text "Hallo Welt". Und diese Variable gibst du dann mit dem Befehl **print** aus.





[Achtung]

Python-Code beginnt **ganz links** am Rand.
Es sind **keine Leerzeichen** zu Beginn der Zeilen erlaubt! Einrückungen werden **ausschließlich** verwendet, um zusammengehörende Programmteile zu kennzeichnen (indem sie eben gleich eingerückt werden).



Ah, und wie oder wo fühle ich das jetzt aus?

Es gibt **zwei Möglichkeiten**, Python-Code auszuführen:

- **Kurzen Code** und sogar einfache **Berechnungen** kannst du direkt auf der **Kommandozeile** bzw. in der **Python-Shell** ausführen. Das ist ganz praktisch, um mal eben schnell etwas auszuprobieren – oder wenn du etwas berechnen möchtest, aber keinen Taschenrechner zur Hand hast.
- **Mehrzeilige Programme** solltest du in einer **Datei speichern** und dann als richtiges Programm ausführen. Das ist die Arbeitsweise, die wir in erster Linie verwenden wollen. Schließlich willst du ja richtige Programme schreiben, die du speichern und verändern kannst.

Du schreibst den Programmcode also in die Kommandozeile oder mit dem **Editor** deiner Wahl und **speicherst** ihn unter einem beliebigen Namen, aber **mit der Endung .py** ab.

```
HalloWelt.py - C:\Desktop - Geany
Datei Bearbeiten Suchen Ansicht Dokument Projekt Erstellen Werkzeuge Hilfe
Neu Öffnen Speichern Alle speichern Zurück Vor Ausführen
HalloWelt.py x
1 mein_text = "Hallo Welt"
2 print(mein_text)
3
Zeile: 2 / 3 Spa: 16 Aus: 0 EINFG Tab mode: CRLF Kodierung: UTF-8 Dateityp: Pyt...
```

Nur noch schnell in einen Editor eingeben und unter einem fast beliebigen Namen mit der Endung ».py« speichern

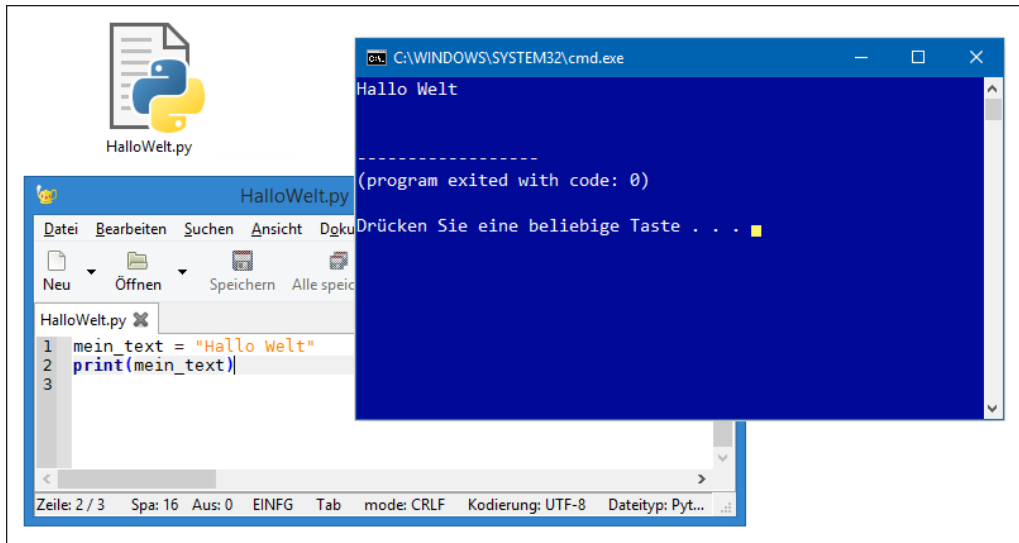
Dann noch ein **Doppelklick** auf diese neue Datei, und schon öffnet sich ein Programmfenster, in dem dein Text ausgegeben wird. Wenn du einen speziellen **Editor** wie Thonny oder Mu hast (oder eine Entwicklungsumgebung), kannst du dein Programm auch direkt darüber starten. Bei dem Editor Geany beispielsweise genügt es, auf den Button **Ausführen** zu klicken. Verwendest du IDLE, die Integrated Development and Learning Environment, dann genügt es, die Taste **F5** zu drücken, um dein Programm zu starten.

Aber ich habe noch gar kein Python ...

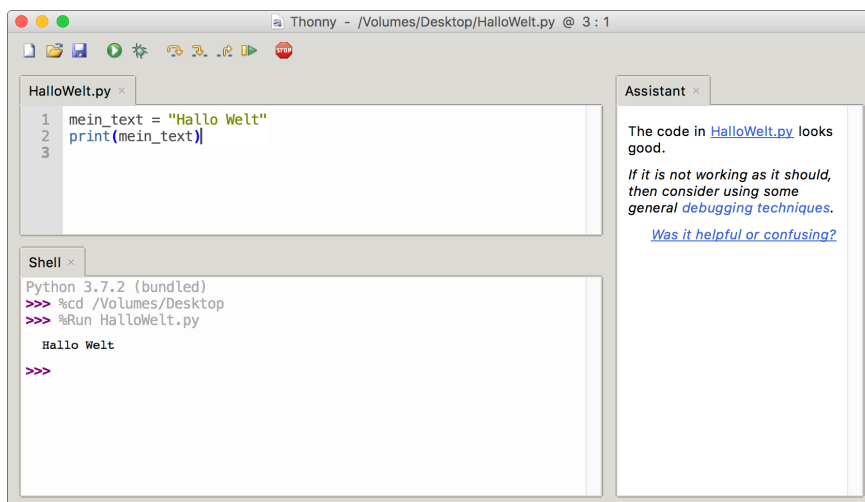
Du hast noch keinen Editor?

Oder hast du kein Python, um die Programmdatei zu starten oder für die Kommandozeile? Dann ist **jetzt** der richtige Zeitpunkt, genau das **in den verlorenen Kapiteln im Anhang** nachzulesen. Dort erfährst du mehr darüber, woher du Python bekommst, wie du es installierst und wie du deine Programme schreiben und starten kannst.

Wir sehen uns hier wieder.
Alles klar und bereit? Dann weiter ...



So kann dann das erste »Hallo Welt« aussehen, hier auf der Kommandozeile von Windows.
»code: 0« heißt so viel wie »kein Fehler«.



Und so sieht unser »Hallo Welt« mit dem Editor Thonny aus. Auch nicht schlecht ...

Schauen wir uns mal genauer an, was da gemacht wurde:

Ein Python-Programm (also geschriebener Quelltext im Editor) **fängt einfach an**. Leerzeilen spielen dabei erst mal überhaupt **keine Rolle**. Du schreibst die Anweisungen und Befehle **Zeile für Zeile** in eine Datei. Es gibt keine Klassen oder andere Konstrukte, die du zuerst als »Rahmen« für das Programm definieren müsstest. Auch der **Name** der Datei, in der du dein Programm speicherst, folgt nur den Regeln des (hoffentlich) gesunden Menschenverstandes. Wichtig ist nur, dass dein Betriebssystem den Dateinamen akzeptiert.

*1 Ein **gültiger Name** einer Variablen beginnt mit einem Buchstaben oder einem Unterstrich. Danach sind Buchstaben, Zahlen und auch wieder Unterstriche erlaubt. Auch Umlaute kannst du problemlos verwenden.

*2 Eine **Zuweisung** geschieht mit einem einfachen **=**. Rechts von dem **=** kann ein **Literal** (also ein echter, fester Wert), eine andere **Variable** mit einem Wert oder eine ganze **Operation** stehen, die einen Wert als Ergebnis erzeugt. Dieser Wert wird dann in der Variablen gespeichert.

```
mein_text*1 =*2 "Hallo Welt"*3*4
```

*3 Das ist ein **Literal**, hier in Form eines Textes, in der Programmierung **String** genannt. Gut zu erkennen ist ein String, da einfache **oder** doppelte Anführungszeichen den Text umschließen.

*4 Das ist das **Ende** dieser Anweisung – natürlich noch nicht das Ende des Programms. In der nächsten Zeile geht es ja weiter. Sicher ist dir aufgefallen, dass hier am Ende **gar nichts** steht, kein Semikolon und kein anderes Zeichen. Python verzichtet auf ein abschließendes Zeichen.

Was ist denn der Unterschied zwischen einfachen und doppelten Anführungszeichen?

Es gibt keinen, beide sind **gleichberechtigt**! Der Vorteil ist, dass du das jeweils andere Anführungszeichen dann ohne Probleme **innerhalb** des Textes verwenden kannst: **"Hallo 'liebe' Welt"** oder **'Hallo "liebe" Welt'** kannst du so problemlos schreiben.

*1 Das klassische **print** sorgt dafür, dass etwas ausgegeben wird. Was? Das steht nicht in den Sternchen, sondern innerhalb der runden Klammern, ...

*2 ... die du hier siehst. Die Klammern werden geöffnet und (klar) wieder geschlossen.

```
print*1(*2mein_text*3)*4
```

*4 Auch hier wieder am Ende: kein Semikolon, keine End-Anweisung, keine geschweifte Klammer, weder am Ende der Zeile noch in der Zeile darunter.

*3 Alles, was in den Klammern steht, wird ausgegeben. Das könnte ein Literal, eine Operation oder wie hier eine Variable sein.



Fertig! Dein (aller-)erstes Python-Programm



Fingerübungen mit »print«

Machen wir einfach ein paar Übungen dazu. So bekommst du etwas Praxis im Umgang mit Variablen und wirst sehen, wie flexibel der Befehl **print** ist.



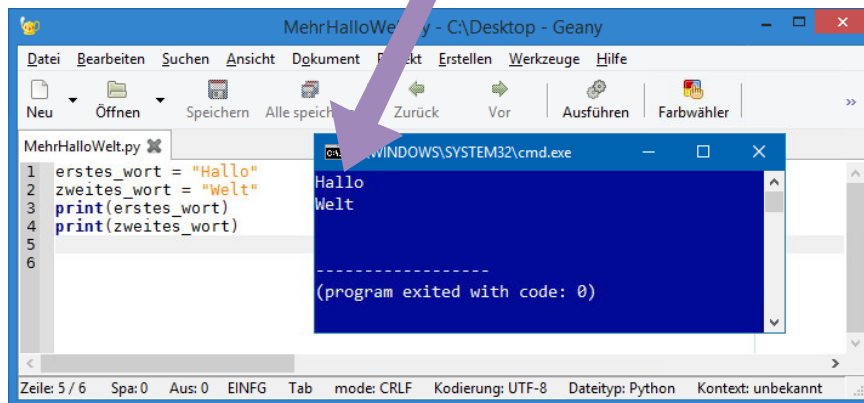
[Einfache Aufgabe]

Verwende bitte für jedes Wort von **"Hallo Welt"** eine eigene Variable und gib sie dann mit **print** aus.



Lösung:

```
erstes_wort = "Hallo"
zweites_wort = "Welt"
print(erstes_wort)
print(zweites_wort)
```



Ausgegeben wird alles – aber gleich in zwei Zeilen!?

Jede Ausgabe steht in einer eigenen Zeile. Das könnten wir natürlich zur gewünschten Lösung erklären, aber so einfach wollen wir es uns doch nicht machen. **Zumal der Befehl print einiges zu bieten hat!**

[Notiz]

Jedes **print** schreibt seinen Inhalt in eine neue Zeile.



Hallo Welt in (einzeiligen) Variationen

1. Du fügst die beiden Wörter in einer neuen Variablen zusammen und gibst den Inhalt dieser neuen Variablen dann mit einem **print**-Befehl aus:

*2 Unsere neue Variable namens **ergebnis** gilt für die beiden zusammengeführten Wörter.

*1 Du kannst **mehrere Zuweisungen** in **einer Zeile** zusammenfassen. Du hast auf der linken Seite des = mehrere Variablen und genauso viele Werte auf der rechten Seite. Die Variablen und die Werte sind jeweils durch ein Komma getrennt.

```
erstes_wort, zweites_wort*1 = "Hallo", "Welt"  
ergebnis*2 = erstes_wort +*3 " "*4 + zweites_wort  
print(ergebnis)
```

*3 Strings werden in Python ganz einfach mit einem + aneinandergehängt.

*4 Das **Leerzeichen** solltest du auch einfügen: Ohne Leerzeichen käme **"HalloWelt"** heraus.

Python weiß, dass ein + zwischen Strings kein mathematisches + ist, sondern Texte zu einem langen Text **zusammenfügt**. Dabei spielt es keine Rolle, ob die Texte als Literale oder Variablen angegeben sind. Wie du in unserem Beispiel siehst, können Variablen und Literale (hier unser einsames Leerzeichen) problemlos miteinander gemischt werden.

2. Wir geben beide Variablen zusammen aus. Gemeinsam im Befehl **print** – ohne alles erst in einer Variablen zu speichern. Auch das ist eine durchaus gängige Praxis:

```
erstes_wort, zweites_wort = "Hallo", "Welt"  
print(erstes_wort + " " + zweites_wort)
```

3. Oder wir machen es nach Python-Art:

Python ist eine sehr **praktische** Programmiersprache. Die **print**-Funktion ist so **flexibel**, dass du es schneller und einfacher machen kannst als mit dem **+**. Eben auf the Python way: Du kannst **print** mehrere Werte übergeben, die du einfach durch ein Komma trennst.

```
print(erstes_wort,zweites_wort)
```

Ja, aber hast du nicht das Leerzeichen vergessen?

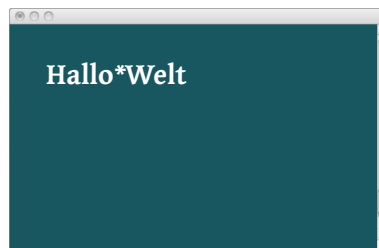
Nein, **print** fügt **automatisch** zwischen alle angegebenen Werte ein Leerzeichen ein. Verantwortlich dafür ist ein **optionaler Parameter** namens **sep**. Das steht für **Separator**, also Trennzeichen. Diesem Parameter kannst du einen beliebigen String zuweisen, der dann zwischen alle angegebenen Werte eingefügt wird. Verwendest du **sep** nicht (denn dieser Parameter ist ja optional), dann wird automatisch ein Leerzeichen zwischen alle Elemente gesetzt. Probier es mal mit einem ***** aus:

***1** Leerzeichen in den Anweisungen und **zwischen** den auszugebenden Elementen dienen nur der **Übersichtlichkeit** und haben **keinen** weiteren Einfluss.

```
print(erstes_wort, *1zweites_wort, *1sep='*'*2)
```

***2** Was **sep** zugewiesen wurde, wird zwischen jedes Element bzw. jeden Wert geschrieben, den du im **print** angegeben hast. Du kannst **sep** einen beliebigen (und auch beliebig langen) Text zuweisen, sogar einen leeren String, also **""**.

Die Ausgabe:



[Code bearbeiten]

Probiere es einfach mal mit verschiedenen Werten anstelle des Sternchens selbst aus. Nun noch alles in einer Python-Datei deiner Wahl **speichern** und **ausführen** lassen.

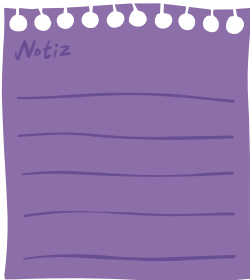
4. Das ist aber noch gar nicht alles. Die **print**-Funktion kennt noch einen weiteren optionalen **Parameter** namens **end**. Normalerweise schreibst du mit jedem **print** eine neue Zeile. Mithilfe von **end** kannst du das ändern und anstelle des Zeilenumbruchs ein anderes Zeichen angeben:

```
erstes_wort, zweites_wort = "Hallo", "Welt"  
print(erstes_wort, end=" "*1)  
print(zweites_wort*2)
```

*1 Hier wird der Zeilenumbruch »umgebogen« zu einem **Leerzeichen**. Genau das, was wir brauchen.

*2 Hier müssen wir nichts ändern. Der standardmäßige Zeilenumbruch darf hier bleiben. Anders wäre es, wenn wir weitere Worte mit einem eigenen **print** in die gleiche Zeile schreiben wollten.

Die Parameter **sep** und **end** müssen für jedes **print** explizit angegeben werden – zumindest wenn du nicht das normale Verhalten von **print** haben möchtest: Ohne diese Angaben hat das jeweilige **print** nämlich wieder die (unsichtbaren) Standardwerte **sep=' '** und **end='\n'**.



[Notiz]

Das **\n** steht für eine neue Zeile. Auf diese etwas seltsame Art, beginnend mit einem ****, können sogenannte **Steuerzeichen** geschrieben werden. **\n** steht dabei für **new line**, also für einen Zeilenumbruch. **\t** ist ein Tabulator, **\b** ein Backspace (es geht also wieder eine Stelle nach links).

*Puh, wieviele Möglichkeiten es für »Hallo Welt«!
Und welche soll ich jetzt nehmen?*

Nun, alle Möglichkeiten machen Sinn. Meist ergibt es sich aus der konkreten Aufgabe, welche Methode am sinnvollsten ist. Wichtig ist, dass du weißt, dass es (eigentlich immer) unterschiedliche Möglichkeiten gibt, ans Ziel zu kommen. Such dir einfach das Passendste heraus.



Wir müssen reden: Du und deine Variablen

Variablen spielen in der Programmierung – und damit natürlich auch in Python – eine große Rolle. Sie sind so etwas wie Speicher, die einen Wert aufnehmen können. Wie du bereits gesehen hast, ist das Arbeiten mit Variablen recht einfach: Variablen **entstehen**, indem du ihnen einen Wert **zuweist**. Woher dieser Wert kommt – ein Literal (also ein fester Wert wie ein Text oder eine Zahl), eine Berechnung oder irgendeine andere Operation –, spielt erst mal keine Rolle.

```
neue_variable = "Ich bin ein Literal"  
andere_variable = 42 * 2 / ( 0.5 * 4 )
```



[Achtung]
Keine Variable kann **ohne Zuweisung** existieren!



Du darfst **keine** Variable verwenden, **ohne** ihr zuvor einen Wert zugewiesen zu haben. **Also, was ist hier richtig, und was ist falsch?**

```
spam = eggs * 2 X
```

1 Genau! Python mag die (noch) unbekannte und leere Variable namens **eggs** überhaupt nicht.

In diesem Fall wäre **spam** eine gültige Variable, **nur eggs** ist leider noch unbekannt (und hat auch keinen Wert) und so kommt es zu einem **Fehler**.

```
C:\WINDOWS\SYSTEM32\cmd.exe  
Traceback (most recent call last):  
  File "meinProgramm.py", line 1, in <module>  
    spam = eggs * 2  
NameError: name 'eggs' is not defined  
-----  
(program exited with code: 1)
```

Die Fehlermeldung mag erst etwas kryptisch erscheinen, gibt aber einen eindeutigen Hinweis, nur leider auf Englisch.

Erst wenn du vorher **eggs** einen Wert zuweist, wird Python dein Programm akzeptieren:



```
eggs = 1  
spam = eggs * 2
```

Genauso wenig funktioniert übrigens ein **print** mit einer Variablen, der noch kein Wert zugewiesen wurde:

```
print( eggs*1 )X
```

*1 Nein! Auch das mag Python gar nicht.

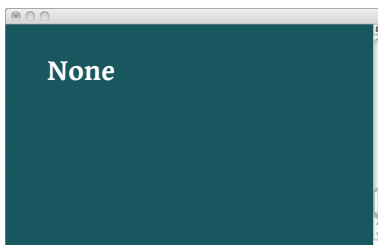
Erst **nach** einer Zuweisung eines Wertes funktioniert das:

```
eggs = "So wird das was"  
print(eggs)
```

Oder anders gesagt: Bevor eine Variable in einer Funktion wie **print** oder einer Operation wie **eggs * spam** verwendet werden darf, muss sie erst mal einen Wert erhalten haben. Dabei sind auch **0** oder **' '** (zwei Anführungszeichen **ohne** Inhalt), also Texte ohne Inhalt, gültige Werte. Es gibt übrigens auch einen speziellen Wert **None**, der für ein voll pythonisch korrektes »**gar nichts**« steht:

```
eggs = None*1  
print(eggs)
```

*1 None wird tatsächlich ohne Anführungszeichen geschrieben, denn es ist kein String, sondern ein in Python an sich gültiger Wert.



Was hat es eigentlich mit **spam** und **eggs** auf sich?



[Hintergrundinfo]

In Python ist es guter Ton, **spam** und **eggs** zu verwenden, wenn man für ein **Codebeispiel** beliebige Variablennamen braucht – sogenannte metasyntaktische Variablen.

spam und **eggs** gehen auf einen Sketch der englischen Comedy-Gruppe Monty Python zurück: Darin versucht ein Gast in einem Restaurant ein Gericht ohne Spam zu bestellen. Spam ist salziges Frühstücksfleisch in eckigen Dosen, die sich kaum ohne Verletzung öffnen lassen.

Wie zu erwarten, gibt es nur Gerichte mit Spam – und zufällig anwesende Wikinger (!) singen in dem Sketch regelmäßig einen Lobgesang auf Spam. Und jetzt rate mal, wo das Wort Spam für unerwünschte Mails herkommt ...

[Achtung]

Variablen **entstehen**, indem ihnen ein Wert zugewiesen wird. Solange eine Variable keinen Wert hat, kannst du sie auch nicht verwenden. Das würde einen **Fehler** verursachen.



[Hintergrundinfo]

Durch die erste Zuweisung eines Wertes, die sogenannte **Initialisierung**, wird eine Variable **deklariert** – sie wird vereinfacht ausgedrückt dem System bekannt gemacht. Ab diesem Zeitpunkt kannst du sie verwenden. Die Initialisierung und die Deklaration sind wichtige Konzepte in praktisch allen Programmiersprachen.



Und du hast ja schon gesehen, dass du sogar mehrere Zuweisungen **zusammenfassen** kannst:

```
spam, eggs, brot = "lecker", "gekocht", "geschnitten"  
kleine_zahl, grosse_zahl = 100, 1
```

Jede Variable bekommt der Reihenfolge nach einen Wert zugewiesen. Die Anzahl der Variablen **links** und die Anzahl der zugewiesenen Werte auf der **rechten** Seite müssen natürlich genau passen, sonst gibt es einen Fehler.

Moment, du hast die Werte in der zweiten Zeile vertauscht!

Du hast recht! Aber mit Python sind die Werte zweier Variablen **schnell getauscht**:

```
kleine_zahl, grosse_zahl = 100, 1  
kleine_zahl, grosse_zahl = grosse_zahl, kleine_zahl
```

Das geht ganz einfach, denn du kannst einer Variablen problemlos den Wert einer anderen Variablen zuweisen.



In der Programmierung müssen oft die Werte zwischen Variablen getauscht werden, zum Beispiel wenn große und kleine Werte verglichen und dann zum Sortieren getauscht werden müssen. Mit Python geht das schnell und einfach mit einer einzigen Zeile Code!

Übrigens... Variablennamen werden nach **Groß- und Kleinschreibung** unterschieden, **mein_Text** ist also eine ganz andere Variable als **Mein_text** oder **mEiN_tExT**. **eggs** ist eine andere Variable als **Eggs**. Wenn du dich beim Schreiben von Variablennamen mal vertippen solltest, wird dich Python bereits beim Programmstart mit einer Fehlermeldung darauf hinweisen, dass da eine unbekannte oder falsch geschriebene Variable im Quelltext sitzt.



[Hintergrundinfo]

Auch wenn Python eher niederländisch-locker ist, es gibt eine Namenskonvention nach PEP 8: Alle Namen von Variablen und Funktionen (die kommen noch) sollten **klein mit Unterstrich(en)** geschrieben werden. Einzelne Worte in den Namen werden dabei zur besseren Lesbarkeit mit Unterstrichen getrennt. Das ist **lower_case_with_underscores**.

[Begriffsdefinition]

Namenskonventionen sind Regeln bzw. Vorschläge, wie du Namen von Variablen oder Funktionen schreiben solltest. Konventionen deshalb, weil es nur Empfehlungen sind. Wenn du dich (vielleicht aus gutem Grund) nicht daran hältst, gibt es keinen Fehler und auch keine Warnung.



Variablen – was geht? Und was ist voll krass korrekt?



Hier sind erst mal ein paar **gültige** Variablennamen. Schau sie dir bitte mal an:

```
_wert = "Hallo"*1  
wort1 = "Schrödinger"*2  
wörtlich = "wie"*3  
wort_drei*1 = "geht's"
```

*1 Der Unterstrich ist das **einzigste, erlaubte Sonderzeichen**, egal, ob am Anfang, innerhalb des Namens oder am Ende.

*2 **Zahlen** sind natürlich auch im Namen erlaubt (nur eben nicht an erster Stelle).

*3 **Umlaute** sind auch kein Problem.



[Einfache Aufgabe]

Lass dir diese Variablen mit einem **print** ausgeben, sodass zwischen jedem Wort ein Bindestrich »-« steht. Am Ende soll ein Fragezeichen stehen und trotzdem noch ein Zeilenumbruch gemacht werden. Spicken ist wie immer erlaubt ...

Lösung:

```
print(_wert, wort1, wörtlich, wort_drei, sep="-", end="?\n")
```

geforderten **Zeilenumbruch** zuweist:

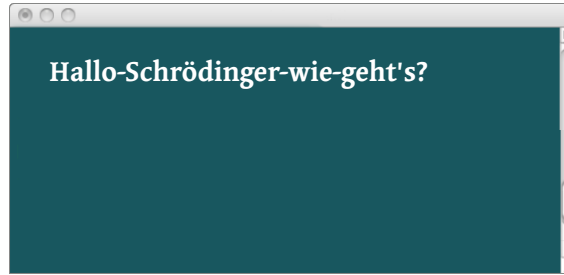
Und zum Schluss noch ein passendes **end** dazu, dem du ein **Fragezeichen** mit dem

```
print(_wert, wort1, wörtlich, wort_drei, sep="-")
```

Danach »baust« du mit dem Parameter **sep** noch den Bindestrich zwischen den Wörtern ein:

```
print(_wert, wort1, wörtlich, wort_drei)
```

Erst mal ganz einfach: die Ausgabe an sich.



Super, das war doch gar nicht so schwer.

Und hier auf besonderen Wunsch der Käferfraktion ein paar Fehler bzw. Variablennamen, die du **nicht** nehmen darfst, weil sie **falsch** sind!



[Notiz]

Fehler werden in der Programmierung gerne als **Bugs** bezeichnet. Ein Bug ist im Englischen ein **Käfer**. Angeblich rührt das von der Zeit der ersten, damals riesigen Computer her. Eine kleine Motte hatte in einem Computer einen Schalter außer Gefecht gesetzt, was zu unerklärlichen Fehlern im Programm führte.



Agent Schrödinger,
übernehmen Sie!

Klaro!



*1 Das Dollarzeichen ist ein unerlaubtes Sonderzeichen, und zwar egal, ob am Anfang wie hier oder innerhalb des Namens.

*2 Am Anfang darf keine Zahl stehen.

*1 \$wort = "Vorsicht" X

*2 lwort = "Schrödinger" X

*3 ...? Moment! Der Name der Variablen ist richtig – nur das **Ist-Zeichen** bei der Zuordnung fehlt!

*3 wort_zwei "so geht" X

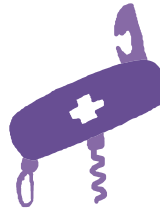
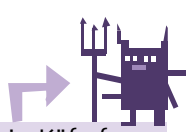
*4 Wort-drei = "es NICHT" X

*4 Der Bindestrich ist ein unerlaubtes Sonderzeichen im Namen. Außerdem entspräche der Name nicht der Namenskonvention alles kleinzuschreiben (was aber kein Fehler wäre).

Schrödinger, schau dir mal folgende Aufgaben an:

[Einfache Aufgabe]

Da der Fehlerteufel in Käferform an vielen Stellen zuschlagen kann: Was ist **richtig**? Was ist **falsch**? Und warum? Jeder Zettel steht übrigens für sich allein.



1.

```
spam = "Irgendwas mit " + egg
```

2.

```
spam = spam + 'Brot'
```

3.

```
spam = 'Ist das "etwa" richtig?'
```

4.

```
egg = "Die Fälschung ist 'echt' falsch"
```

5.

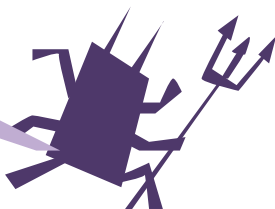
```
egg = 'Ich bin ein Literal'
```

6.

```
egg = "Schrödinger"  
spam = 'Hallo' + Egg
```

7.

```
spam = "Ja" * 3
```





Lösung:

1. Falsch. Die Variable **egg** ist unbekannt, sie hat noch keinen Wert.
2. Falsch. **spam** ist noch unbekannt und darf noch nicht Teil einer Operation (auf der rechten Seite des **=**) sein. Das geht natürlich nicht.
3. Stimmt! Wenn der String durch einfache Anführungszeichen umschlossen ist, kannst du die doppelten Anführungszeichen wie jedes andere Zeichen im String verwenden – übrigens als Zeichen im Text nicht nur paarweise, sondern beliebig oft.
4. Stimmt auch! Der String ist von **doppelten** Anführungszeichen umschlossen. Also kannst du im String die **einfachen** Anführungszeichen verwenden – wieder beliebig oft oder auch nur einmal.
5. Falsch, denn ein String muss immer von der **gleichen Art von Anführungszeichen** umschlossen werden. Hier beginnt der String mit einem einfachen und endet mit einem doppelten Anführungszeichen. Umgekehrt wäre es natürlich genauso falsch.
6. Falsch! **egg** ist eine ganz andere Variable als **Egg** – Python unterscheidet ja zwischen Groß- und Kleinschreibung.
7. Das funktioniert tatsächlich! Das Ergebnis ist dreimal der angegebene String aneinandergesetzt: »JaJaJa«. Strings können ja mit einer Integer-Zahl (und **nur** damit) multipliziert werden!

Das war doch gar nicht so schwer!

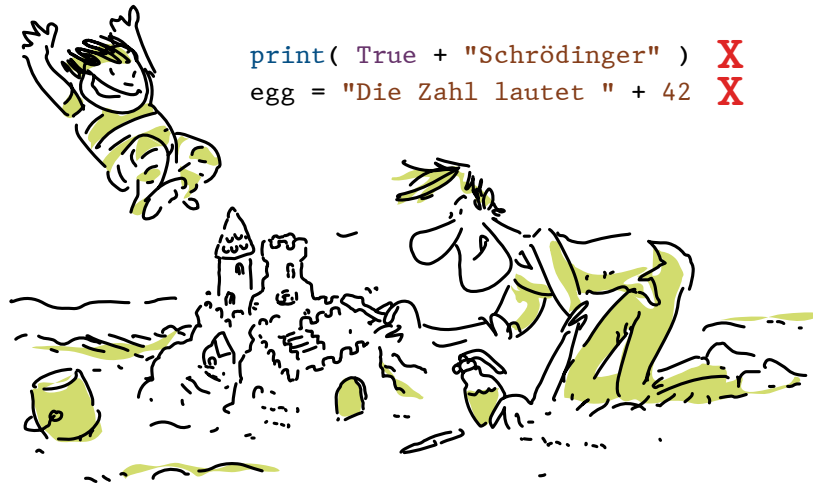


Ach, i wo!

Ach, Schrödinger. Bevor ich es vergesse, ist hier noch eine wichtige Sache.

Die Sache mit den (Daten-)Typen

Schau dir bitte mal das hier an:



```
print( True + "Schrödinger" ) X  
egg = "Die Zahl lautet " + 42 X
```

Wenn es sich nicht ausschließlich um Zahlen handelt, darfst du unterschiedliche Arten von Werten **nicht in Berechnungen bzw. Operationen vermischen**. In der Programmierung werden nämlich Daten nach ihrem **Typ** unterschieden: **Zahlen** und **Strings** kennst du ja schon. Und Zahlen werden sogar noch weiter unterschieden:

- **ganzzahlige Zahlen** wie 1 oder 42 oder 2021, ohne Stellen nach dem Komma
- **Fließkommazahlen** wie 1.9 oder 3.141
- **komplexe Zahlen**, wie du sie vielleicht aus dem Matheunterricht oder dem Studium kennst

Außerdem gibt es noch **boolesche Werte** für wahr, **True**, oder falsch, **False**, in Python. Und dann wirst du noch ein paar andere, gar nicht mal so komische Typen kennenlernen.

[Achtung]

In Python wird, wie in fast allen Programmiersprachen, das **Komma** bei Fließkommazahlen (wie bei 3,14159) als **Punkt** geschrieben: **3.14159**



[Hintergrundinfo]

Nicht umsonst ist eine Fließkommazahl im Englischen eine »floating point number«. Und da Englisch die Sprache der Programmierung ist, richten sich Programmiersprachen hier nach ihr.

Für den Computer ist die Frage des Datentyps wichtig: Ein String muss intern anders behandelt und gespeichert werden als ein Wahrheitswert: Ein String kann fast beliebig lang (oder kurz) sein, während ein boolescher Wert nur **True** oder **False** (also wahr oder falsch bzw. intern **0** oder **1**) darstellt.

Dabei können boolesche Werte, Integer (also ganzzahlige Werte) und Fließkommazahlen recht **gut miteinander** in Berechnungen verwendet werden. In Berechnungen mit Zahlen werden **True** zu **1** und **False** zu **0** umgewandelt. Integer wie **1** oder **42** werden in Berechnungen mit Fließkommazahlen wie **3.141** automatisch in Fließkommazahlen **1.0** oder **42.0** umgewandelt. Hier macht Python die Arbeit für dich ganz automatisch – denn hier ist es ziemlich **eindeutig** wie die Werte zu verstehen sind, und es besteht **keine Gefahr**, dass Werte durch eine automatische Umwandlung verändert oder falsch verstanden werden.

Schau dir bitte mal die Berechnung in dem **print** an und sag mir, was dabei herauskommt:

```
print( 42 + 12.0 * False )
```

Also Schrödinger, Python arbeitet nach den normalen Regeln der Mathematik: **Punktrechnung vor Strichrechnung.**

Somit beginnt die Berechnung rechts mit der Multiplikation. Das **False** wird zu einer Fließkommazahl **0.0** umgewandelt, damit es passend zur Fließkommazahl **12.0** ist. Das ergibt **12.0 * 0.0** und als (Teil-)Ergebnis **0.0**. Das Ergebnis **0** bleibt übrigens eine Fließkommazahl als **0.0**.

Dann berechnet Python **42 + 0.0**. Damit es passend wird, macht Python aus der **42** die Fließkommazahl **42.0** und rechnet die beiden Zahlen zusammen.

Das Ergebnis ist 42.0!

Wenn das mit den Zahlen doch so gut funktioniert, warum dürfen dann ein Text und eine Zahl nicht zusammengefügt werden?

Nun, so lange es **eindeutig** ist, wie Werte zu interpretieren (und damit umzuwandeln) sind, so lange übernimmt Python das gerne für dich – ganz **automatisch**. Aus einer **42** wird in einer Operation mit einer Fließkommazahl immer eine **42.0**. Wichtig für Python ist auch, dass keine Informationen dabei verloren gehen. Deshalb findet immer eine automatische Konvertierung zu dem **höherwertigen** Datentyp statt.

Boolean → Integer → Fließkommazahl

Andersherum geht es **nicht**, zumindest nicht automatisch. Denn wie sollte anders (beispielsweise) aus einer 17.7 eine Integer-Zahl werden? Runden? Aufrunden? Abrunden? **Nein, das ist zu unsicher!** Und wie sollte es erst recht funktionieren, wenn Zahlen und Texte miteinander verknüpft werden sollen?





Wie sagt das Zen of Python?

»In the face of ambiguity,
refuse the temptation to guess. –
Wenn etwas mehrdeutig ist,
widerstehe der Versuchung zu raten.«

Es gibt Programmiersprachen, die **versuchen**, selbst zu erkennen, was sinnvoll ist und wohl gemeint sein könnte: Ist also eine "42" ein String? Oder ist die Zahl gemeint? Gerade bei einer Addition:

```
text_oder_zahl = "42" + 23X
```

Das lässt Python gar nicht zu! Es kommt zu einem Fehler!

Was würdest du hier als Ergebnis erwarten? Die Zahl **65** als Ergebnis einer Addition oder den zusammengeführten Text "**4223**"? Was ist sinnvoll?

Python hat hier eine ganz eindeutige Antwort: **Das kann niemand wissen – nur der Programmierer, also du!**

Was? Ich?

Schrödinger, das Zen of Python sagt dir:


»Explicit is better than implicit. –
Explizit ist besser als implizit.«

Also sagt Python, es ist das Beste, **du** legst ausdrücklich (**explizit**) fest, wie die Werte zu verstehen sind. **Du** musst dich darum kümmern, wie dein Programm mit unterschiedlichen Daten bzw. Datentypen umgeht. **Du** musst explizit festlegen, wie die einzelnen Werte in einer Operation zu verstehen sind!



[Begriffsdefinition]

Explizit bedeutet, du musst es **ausdrücklich** angeben, also in Form eines Befehls oder einer Anweisung wirklich schreiben.


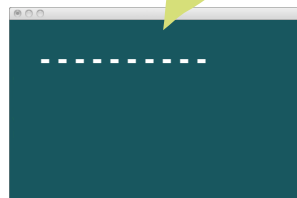


Python **passt genau auf**, dass unpassende Datentypen nicht einfach addiert oder mit anderen Operationen einfach »mal so« zu neuen Werten verarbeitet werden – es sei denn, es ist eindeutig und es gehen keine Informationen dabei verloren. Und von einigen, genau festgelegten **Sonderfällen** einmal abgesehen.

Sonderfälle?

Zum Beispiel die Multiplikation von Strings mit Integer-Zahlen:

```
zeichenkette = "-" * 10  
print( zeichenkette )
```



Du kannst tatsächlich einen String mit einer Zahl multiplizieren. Der String wird dann **vervielfacht**, das ist durchaus ganz praktisch – **willkommen in der wunderbaren Welt von Python.**

Ansonsten gilt aber: Erst wenn Daten explizit – also auf deine Anweisung hin – zu einem **gleichen Datentyp** umgewandelt sind, kannst du sie gefahrlos vermischen, addieren oder sonst etwas damit machen.

[Achtung]

Für **Berechnungen** musst du die beteiligten Daten zu **einem gleichen Datentyp** umwandeln ...



... **dafür ist es natürlich wichtig zu wissen, welche (Daten-)Typen es so gibt!**

Werfen wir einen raschen Blick auf die (erst mal) **wichtigsten Datentypen**, die du zum Programmieren benötigst.

Diese Datentypen sind für dich da!

- **Strings bzw. Texte**

'Hallo' "42"

'10 Vorne' "Schrödinger"

Strings kennst du ja. Gekennzeichnet sind sie durch Anführungszeichen am Anfang und am Ende. Wichtig ist, dass eine **Zahl**, die sich **innerhalb** solcher Anführungszeichen befindet, auch ein **String** ist: "42" ist also etwas ganz anderes als die Zahl 42.

- **Integer-Zahlen**

42 451 -100 1233457898073278928330420382903

Integer-Zahlen, also Ganzzahlen, können positiv oder negativ sein, haben aber eben keinen Nachkommateil. Eine **Besonderheit** von Python ist, dass Integer-Zahlen praktisch **beliebig groß** werden können.

- **Fließkommazahlen**

42.0 -191.0 3.14159 .25 1e2 134.

Fließkommazahlen werden mit einem Punkt geschrieben. Aber auch Zahlen in der **Exponentialschreibweise** (wie eben **1e2**, also **100.0**) sind Fließkommazahlen.

- **Boolean, Wahrheitswerte**

True False

Ist etwas **wahr** oder **falsch**? Das sagen dir die **booleschen** Werte. **True** und **False** müssen übrigens exakt so geschrieben werden: Der erste Buchstabe ist **groß**, der Rest wird **kleingeschrieben**. Sie dürfen auch **nicht** in Anführungszeichen geschrieben werden – dann wären sie nämlich normale Strings.

Und was mache ich jetzt damit?



Über den richtigen Kamm scheren – Datentypen konvertieren

Python hat Funktionen, mit denen du Werte von **einem Datentyp** in einen **anderen Typ** umwandeln kannst. Vorausgesetzt, das geht von den Werten her überhaupt: Denn man kann nicht jeden Wert sinnvoll in einen anderen Datentyp konvertieren. **Man kann eben nicht aus jeder Mücke einen Elefanten machen.**

Um beispielsweise für eine Berechnung unpassende Werte in **Zahlen** zu verwandeln, kannst du die Funktion **int()** verwenden: Damit wird jeder Wert in einen Integer, also eine ganze Zahl, umgewandelt – zumindest soweit das möglich ist. Aus **"Haus"** könnte eben keine Zahl erzeugt werden. Eine Fließkommazahl würde damit in eine Integer-Zahl verwandelt, indem einfach alles nach dem Komma (bzw. Punkt) abgeschnitten würde.

Willst du eine Fließkommazahl erzeugen, hilft dir dabei die Funktion **float()**. Um etwas in einen String zu verwandeln, gibt es dafür die Funktion **str()**.



[Einfache Aufgabe]

Addiere den String **"42"** als Zahl zu der Zahl 23 und weise das einer Variablen zu.

```
eine_zahl = int("42")*1 + 23
```

[Einfache Aufgabe]

Und jetzt füge die gleichen Werten als Strings zusammen und weise das Ergebnis einer Variablen zu.



^{*1} Mit der Funktion **int()** wird ein darin angegebener Wert in eine Integer-Zahl umgewandelt, zumindest wenn der Wert in die andere »Form« als Zahl passt.

```
ein_text = "42" + str(23)*2
```

^{*2} Die Funktion **str()** macht aus dem angegebenen Inhalt einen String. Das Ergebnis ist **"4223"** – zwei Strings wurden zusammengefügt und das Ergebnis dann der Variablen zugewiesen.

So klappt es mit dem Nachbarn, der in diesem Fall eben einen anderen Datentyp hat. Allerdings hat das Grenzen, denn aus "10 Vorne" oder "Hausnummer 13" kannst du auch mit `int()` keine Zahl machen.

Die Konvertierung der Werte funktioniert natürlich nicht nur mit Literalen (wie oben), sondern genauso mit Variablen oder Werten, die aus Operationen entstehen.



[Einfache Aufgabe]

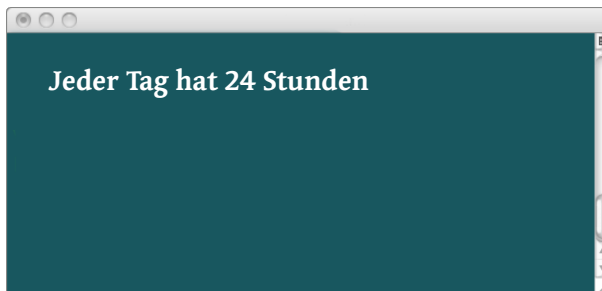
Was wird hier als Ergebnis von `print` ausgegeben?

```
eggs = 20
spam = "Jeder Tag hat *1" + str(4 + eggs) + " *1Stunden"
print(spam)*2
```

*1 Im Gegensatz zum `print`-Befehl musst du beim Zusammenfügen von Strings selbst an notwendige Leerzeichen zwischen den einzelnen Elementen denken.

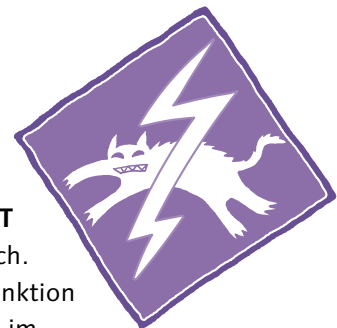
*2 `print` zeigt dir dann, was aus deinem `spam` geworden ist.

Richtig:



[Achtung]

`str(eggs)` verändert übrigens **NICHT** die Variable `eggs`! Ihr Inhalt bleibt gleich. Der umgewandelte Wert wird von der Funktion `str` zurückgegeben und an dieser Stelle im Programm verwendet.



Natürlich kannst du auch ganze Herden von **Operationen** (also Berechnungen) innerhalb solcher Funktionen wie **int()**, **float()** oder **str()** durchführen.



[Einfache Aufgabe]

Welchen Wert hat die Variable **spam** am Ende?

*1 Zuerst wird der Wert innerhalb der Klammern »berechnet«. Somit werden erst die beiden Strings zu dem String **"24"** zusammengefügt. Aus diesem String macht **int** dann die Zahl **24**, die der Variablen **eggs** zugewiesen wird.

```
eggs = int( "2" + "4" *1)
spam = "Jeder Tag hat " + str(eggs + 12*2)*3 + " Stunden"
print( spam )
```

*2 Wieder wird die Operation in der Klammer ausgeführt, bevor konvertiert wird. Die Zahl **24** aus **eggs** wird mit der Zahl **12** zusammengerechnet – zu einer gewagten **36**.

*3 Aus der berechneten Zahl **36** wird ein **String** gemacht, der zusammen mit den beiden anderen Strings zusammengefügt wird.

Das Ergebnis sieht dann so aus:



Endlich hat der Tag
mal genug Zeit,
um so richtig zu chillen!

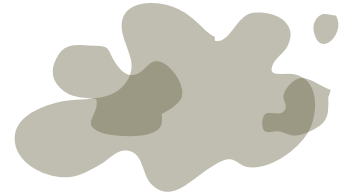


Lustiges Konvertieren – was Python zu was macht



Es ist ganz wichtig, zumindest einmal gesehen zu haben, was Python aus unterschiedlichen Werten bei der Konvertierung macht. Schauen wir uns das mal etwas genauer mit den Funktionen `str()`, `int()` und `float()` an.

Die Funktion »str()« – verwandelt alles in Text



```
str() → ''  
str(0) → '0'  
str(1) → '1'  
str(12.94) → '12.94'  
str(False) → 'False'  
str(True) → 'True'  
str(None) → 'None'  
str('eins') → 'eins'
```

Hier kannst du sehen, welche Strings von der Funktion `str()` aus unterschiedlichen Werten »gebastelt« werden. Sogar leere oder gar keine Werte sind erlaubt und erzeugen eben einen **leeren** String.



Eine Variable ist ja auch mit einem leeren String korrekt initialisiert.

Versuchst du, eine Fließkommazahl **mit Komma** zu schreiben, führt das zu einem Fehler:

```
str(3,141)X
```

Das liegt nicht mal an der Funktion `str()`. Python akzeptiert eben **kein Komma** als Dezimalzeichen – nur der im Englischen übliche **Punkt** ist erlaubt.



Die Funktion »int()« – ganze Zahlen

Mithilfe von **int()** werden Werte in Ganzzahlen, also Integer, umgewandelt.

```
int() → 0
int(0) → 0
int(1) → 1
int(12.94) → 12
int("42") → 42
int(False) → 0
int(True) → 1
```

»Leere Werte« sind auch hier erlaubt. Auch aus einem Text **"42"** wird korrekt eine Zahl **42**. Und Fließkommazahlen werden ja umgewandelt, indem der Teil nach dem Komma einfach **abgeschnitten** wird.



Bei einem **String** wie **"42.123"** versucht Python aber erst gar nicht, eine **42** zu raten – es ist wieder deine Aufgabe als Programmierer, dich darum zu kümmern. Nur aus direkt passenden Strings wie **"42"** erzeugt **int()** eine Zahl. Und auch ein Text, der eine Zahl als Wort darstellt, wie **"one"** oder **"eins"** wird von Python nicht als Zahl zurechtgeraten, sondern mit einem Fehler quittiert – genauso wie der Versuch, **None** in eine Zahl zu verwandeln.

```
int("42.123") X
int("one") X
int("eins") X
int(None) X
int("10vorne") X
```

Was bei **int()** mit dem String **"42.123"** nicht funktioniert, funktioniert natürlich bei **float()**, da die Funktion ja Zahlen mit Dezimalpunkt kennt.

Die Funktion »float()« – Fließkomma mit Punkt

Die Funktion `float()` ist der große Bruder von `int()` und für Fließkommazahlen zuständig – nur eben mit dem Punkt als Dezimalzeichen.

```
float() → 0.0
float(0) → 0.0
float(1) → 1.0
float(3.141) → 3.141
float("42") → 42.0
float("42.123") → 42.123
float(True) → 1.0
float(False) → 0.0
```

Und auch hier gibt es Werte, die nicht in eine Fließkommazahl umgewandelt werden können:

```
float("eins") X
float(None) X
float(3,141) X
```



Die Funktion »bool()« – Wahrheit oder Pflicht



Die Funktion `bool()` ist von solch schlichter Eleganz und Einfachheit, dass sie mit einem Absatz abgehandelt werden kann: `bool()` macht aus gar keinem Wert, `None`, einem leeren String `' '` oder einer `0` (oder `0.0`) ein `False` – alles andere ist `True`. Selbst ein `"False"` oder ein Leerzeichen `' '` als Text wird zu einem `True`.

Fehlt jetzt noch was? Klar!

Du musst ja auch feststellen können, von welchem Typ ein bestimmter Wert ist. So lange du **Literale** hast, ist das natürlich banal – du siehst ja direkt, um was für einen Wert es sich handelt. Kniffliger wird es bei **Operationen** oder wenn die Daten eben nicht direkt unter deiner Kontrolle sind, wenn Daten zum Beispiel aus einer Datenbank, dem Internet oder einer Benutzereingabe kommen.

Was ist das denn für ein Typ – »type()«

Mit der Funktion **type()** kannst feststellen, um **welchen Datentyp** es sich bei einem Wert handelt. Dabei ist es egal, ob der Wert als Literal oder in Form einer Variablen vorliegt.

Du musst nur den Wert (bzw. die Variable) in die Klammer der Funktion schreiben. **type()** überprüft den Typ – todsicher und schnell.

Hier ist einmal vereinfacht das Ergebnis solcher Prüfungen dargestellt. Willst du das Ergebnis sehen bzw. ausgeben, dann gib **type()** am einfachsten mit einem **print()** aus:

```
print( type(spam) )
```



Und praktisch ausprobiert sieht das so aus:

***1** Innerhalb der Klammer einer Funktion – also auch bei **type()** – kannst du Operationen bzw. Berechnungen verwenden. Es wird das **Ergebnis** der gesamten Operation untersucht: Da hier 10 unzweifelhaft größer als 1 ist, ergibt dieser Vergleich im Ergebnis wahr, also **True**. Und **type** stellt somit ganz richtig fest, dass wir einen **booleschen Wert** haben.

Code	Ausgabe
<code>type(42)</code>	<code><class 'int'></code>
<code>type(42.123)</code>	<code><class 'float'></code>
<code>type("Hallo")</code>	<code><class 'str'></code>
<code>type(False)</code>	<code><class 'bool'></code>
<code>type(10 > 1*1)</code>	<code><class 'bool'></code>
<code>type()*2</code>	FEHLER!

***2** Die Funktion **type()** verträgt es nicht, wenn sie leer, also **ohne Inhalt**, aufgerufen wird. Dann kommt es zu einem Fehler.

Die Ausgabe in der Form **<class 'IRGENDWAS'>** erscheint auf den ersten Blick vielleicht etwas seltsam. Du musst dazu wissen, dass so ziemlich alles in Python als ein **Objekt** angesehen wird. Daher rührt die Angabe **class**. Bei diesen Klassen, **class**, handelt es sich um so etwas wie **Vorlagen**, was aus der objektorientierten Programmierung kommt. Hinter jedem Datentyp steht also eine entsprechende Klasse. Und genau diese Klasse beschreibt, was die jeweiligen Datentypen ausmacht.

Uii, das sieht aber kompliziert aus!

Keine Sorge: Für einen **Vergleich** in einem Programm, also um zu testen, ob ein bestimmter Typ vorliegt, ist der Aufruf viel leichter. So prüfst du beispielsweise, ob der Wert einer Variablen vom Typ her ein String ist:

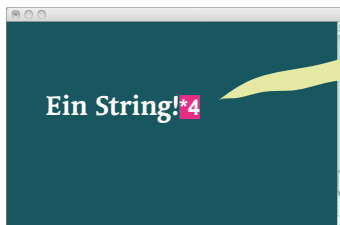
*1 Das hier ist ein **if**. Damit kannst du Entscheidungen treffen, die von einem **Vergleich** abhängen. Das **if** wirst du später noch genauer kennenlernen.

*2 Hier fragst du den **Typ** ab, den der Wert der Variablen gerade hat, ...

```
eingabe = "Hallo"  
if *1 type(eingabe) *2 == str *3:  
    print("Ein String!") *4
```

*3 ... und vergleichst ihn mit dem **Typ String** – der ganz einfach als **str** (ohne Anführungszeichen) geschrieben wird.

*4 Siehe da: Es ist ein String!



Auch auf andere Typen kannst du so testen. **bool** steht für einen Wahrheitswert, **int** für einen Integer und **float** für eine Fließkommazahl.

Alle Typen werden **ohne Anführungszeichen** geschrieben.

Aber im Moment genügt, dass du feststellen kannst, um welchen Datentyp es sich handelt.



Probier's doch mal aus



[Einfache Aufgabe]
Finde heraus, um welchen Datentyp es sich jeweils handelt.

```
spam = "10" * 421  
spam = int("42") > 302  
egg = 12.0 * 33  
ergebnis = spam * egg4
```

² Klar, das Ergebnis aus einer **Vergleichsoperation** zweier Zahlen mit $>$ ist (in diesem Fall) wahr, also **True**, und damit ein boolescher Wert, also **bool**.

⁴ In einer Berechnung mit einer Zahl wird unser boolescher Wert in **spam** zu **0** oder **1**. Unser **True** wird also zu **1** bzw. in der Operation zu **1.0** und das Ergebnis ist **36.0** – also wieder **Float**.

³ Einmal **Float** und einmal **int**? Klar, das Ergebnis ist in dem Fall praktisch immer **Float**, hier **36.0**

¹ Das Ergebnis ist ein ziemlich langer **String** in der Art "10101010..." (wobei wir uns den Rest des Strings sparen).

Mit `print(type(name_der_variable))` kannst du dir den Typ jeweils ausgeben lassen.

Und was ist noch wichtig?

Eine Variable selbst ist nicht dauerhaft auf eine bestimmte Art von Wert bzw. einen Datentyp festgelegt. Im Gegenteil darf jede Variable lustig ein »Bäumchen wechsel dich« spielen.

```
spam = 42  
spam = "Schrödinger"  
spam = True
```



Auch wenn Variablen unterschiedliche Datentypen haben und **wiederverwendet** werden können, gilt folgender Hinweis:

[Achtung]

Es ist **keine gute Idee**, Variablen »recyclen« zu wollen. Nimm lieber eine **neue Variable**.

Python kümmert sich um Variablen, die nicht mehr benötigt werden. Es wird also kein Speicher durch alte Variablen belegt. Die Anzahl der Fehler, die durch recycelte Variablen entstanden sind, ist hingegen legendär. Viele Entwickler können davon ein leidvolles Lied singen, und es gibt kaum bessere Möglichkeiten, sich bei Kollegen unbeliebt zu machen.



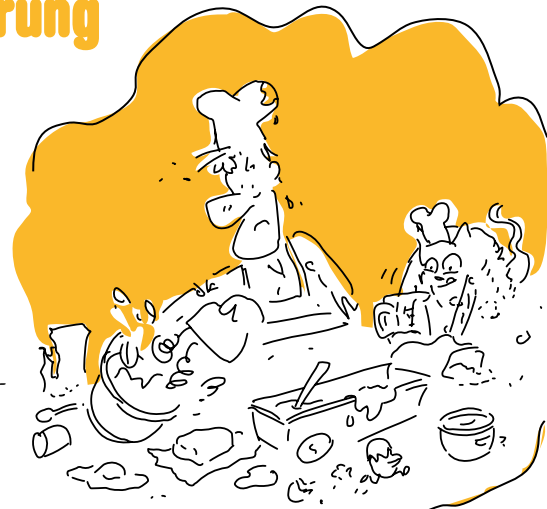
**Schrödinger, das Zen of Python sagt dir:
»Readability counts. – Die Lesbarkeit zählt.«**



Syntax, Variablen, Datentypen und dynamische Typisierung

Lass uns kurz rekapitulieren ...

- Python ist nach einer (nein, **der**) britischen **Komikertruppe Monty Python** benannt. Wichtige Empfehlungen und Regeln sind in den sogenannten PEPs festgelegt.
- Die **Syntax** ist »leichtgewichtig«. Gerade auf die schlecht zu tippenden geschweiften Klammern oder das »schicke« Semikolon wurde weitgehend verzichtet. Jeder Befehl steht in einer eigenen Zeile.
- Ein Python-Programm schreibst du einfach so. Keine Klassenkonstrukte, keine Importe oder andere Vorbereitungen sind nötig: einfach schreiben, speichern, ausführen.
- Variablen** sind Speicher, die unterschiedlichste Werte aufnehmen können. Du kannst ihnen Literale, also feste Werte, oder die Ergebnisse von Berechnungen und Operationen zuweisen.
- Jeder Variablen musst du vor der Verwendung einen Wert zuweisen. Den Datentyp erkennt Python selbst, dank **dynamischer Typisierung**.
- Unterschiedliche **Datentypen** dürfen nicht ohne Weiteres in Berechnungen vermischt werden. Es ist aber problemlos möglich, den Typ eines Wertes festzustellen und für eine Berechnung passend zu machen.
- Mit **print** kannst du Werte komfortabel ausgeben, getrennt durch ein Komma verstehen sich hier sogar unterschiedliche Datentypen wunderbar.



Zeit für eine Pause ...?
... keine Pause!

—ZWEI—

Syntax,
Kommentar und
guter Stil

Ein Dinkel macht noch keinen Korn

**Ein einfaches Kassenprogramm schreiben?
Eingaben machen, das Wechselgeld und einen Rabatt
berechnen? Mit Python ist das überhaupt kein Problem –
das kriegst du schneller hin, als du denkst!**

Also, es gilt! Noch drei Tage bis zum Flohmarkt. Ein **einfaches Kassensystem** hattest du versprochen. Was ist also zu tun? Es muss ein Verkaufspreis eingegeben werden und der Betrag, den der Kunde zahlt. Und das Kassensystem berechnet dann das Rückgeld ...

... ganz übersichtliche Anforderungen für den Anfang:

1. Wir brauchen eine **Eingabe** für den (Verkaufs-)Preis. Der sollte sinnvollerweise in einer Variablen **gespeichert** werden.
2. Dann muss eine **zweite Eingabe** erfolgen – für das vom Kunden gezahlte Geld. Und auch diese Eingabe sollte in einer Variablen **gespeichert** werden.
3. Schließlich muss aus den Eingaben das **Rückgeld berechnet** und (auf dem Bildschirm) **ausgegeben** werden.

Reicht das denn schon für ein Kassensystem?

Das sind erst einmal die ([aller-]aller-)wichtigsten **Grundfunktionen**. Natürlich macht ein professionelles Kassensystem viel mehr, aber für den ersten Schritt, für einen einfachen Prototyp, reicht das. Zumal du gerade hier einige wichtige Grundlagen von Python kennlernst: die ersten **Kontrollstrukturen**.

[Begriffsdefinition]

Ein **Prototyp** ist Software, die durchaus erst mal mit der »heißen Nadel« gestrickt wird, um zunächst eine grundlegende Funktionalität zu haben.

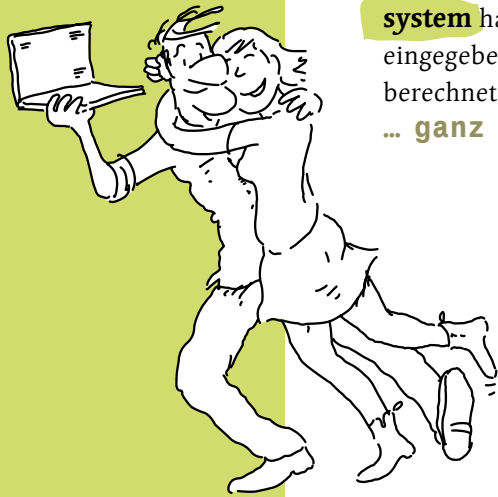
Man verwendet Prototypen gerne in der professionellen Entwicklung, um zusammen mit dem Kunden Schritt für Schritt zum fertigen, ausgereiften Programm zu kommen.

[Begriffsdefinition]

Kontrollstrukturen sind Elemente in der Programmierung, mit denen der **Ablauf** eines Programms gesteuert werden kann. Ansonsten würde jedes Programm immer gleich linear ablaufen – ohne Unterscheidungen vom Anfang zu einem immer gleichen Ende.

[Achtung]

Das heißt **nicht**, dass schlampig oder schlecht programmiert wird. Es wird aus Zeitgründen nur wenig Wert auf Feinheiten und Details gelegt. Die **Funktionalität** steht im Vordergrund.





Eingabe, Berechnung und eine Ausgabe

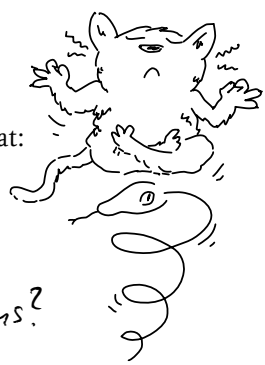
Was du erst mal brauchst, ist die Möglichkeit, einen Wert **einzugeben**. Der Befehl dafür lautet **input()**.



Bei diesem Befehl stoppt das Programm, gibt einen optionalen Text aus und erwartet dann eine Eingabe. Mit der **Return-** bzw. **Enter**-Taste beendest du die Eingabe. Diese **Eingabe** kann einer Variablen zugewiesen werden (was in den meisten Fällen gemacht wird) oder auch direkt in einer Operation verwendet werden. Letzteres wird aber eher selten gemacht, da das doch eher **unübersichtlichen Code** erzeugt.

Denk dran, Schrödinger, das Zen of Python weiß immer Rat:

»Beautiful is better than ugly. –
Schön ist besser als hässlich.«



Gut, wie sieht das nun mit dem **input** richtig aus?

```
spam*1 = input*2("Ein Hinweistext"*3)
```

***1** Unserer beispielhaften (also ziemlich metasyntaktischen) Variablen **spam** wird die Eingabe zugewiesen – also alles, was du tippst, bis du die **Return-** oder **Enter**-Taste drückst. Mit der Variablen und ihrem neuen Inhalt kannst du dann problemlos im Programm weiterarbeiten.

***2** Der Befehl **input** nimmt einen Wert entgegen und gibt ihn sofort weiter – in unserem Fall direkt an die Variable **spam**.

***3** In der Klammer kannst du einen Hinweis angeben, welche Art von Eingabe erwartet wird. Sinnvoll ist ein kurzer, erklärender Text, aber auch Zahlen oder boolesche Werte wären hier erlaubt.



Fangen wir an. Erst einmal unsere Eingabe(n):

*1 Eine Variable namens **preis** bekommt den Wert der Eingabe zugewiesen. Der Name ist beliebig, sollte aber natürlich zur Verwendung passen. **preis** (oder **Preis**) ist in jedem Fall besser als **prsl** oder nur **p**!

*2 Und hier kommt unser **input** mit einem kurzen, erklärenden Text zur erwarteten Eingabe. Natürlich reicht als Text auch **"Preis?"**.

```
preis*1 = input*2('Gib den Preis ein: *3')  
zahlung = input("Der Kunde zahlt: ")
```

*3 Sicher hast du das Leerzeichen bemerkt!? Das Leerzeichen sorgt dafür, dass deine Eingabe etwas Abstand zum Hinweistext hat. Das ist nicht notwendig – wirkt aber etwas schicker.

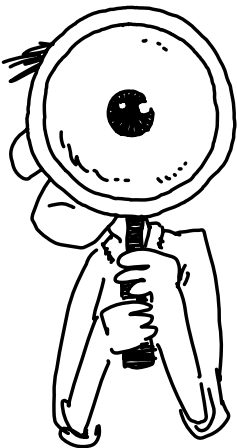
*Warum hast du den Text innerhalb des Beiden **input()** einmal mit einfachen und einmal mit doppelten Anführungszeichen geschrieben?*



So siehst du, dass du die unterschiedlichen Anführungszeichen ' ' und " " tatsächlich beliebig verwenden kannst. Du musst nur beachten, dass die **zusammengehörigen** Anführungszeichen gleich sind. Mischen ist da nicht erlaubt.

Bevor wir jetzt anfangen zu rechnen, ist es wichtig zu wissen, dass Python jede Eingabe, die über **input** gemacht wird, als **String** bereitstellt.

Als String, also als Text? Auch die eingegebenen Zahlen?



[Achtung]
Jede Eingabe, die mit **input** gemacht wird, behandelt Python als **String**!



Es ist wie bei der Addition von Variablen mit unterschiedlichen Datentypen: Welchen Typ sollen die jeweiligen Werte haben? Ist eine 42 nun eine Zahl 42 oder ein String **"42"**?

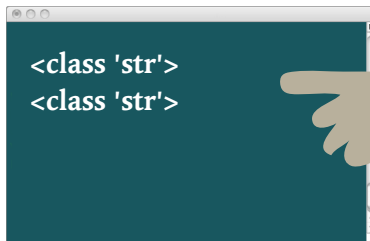
Python geht auch hier davon aus, dass **nur du**, der Entwickler, das wissen kannst. Deshalb werden alle Werte als **String** bereitgestellt, String ist schließlich der Datentyp, der erst mal alles problemlos aufnehmen kann. Und nur **du** kannst (und musst) dich dann darum kümmern, dass die Werte passend gemacht werden.

Ach, deshalb warhin das Ganze mit diesen Datentypen?!

Welchen Datentyp Python für die Eingaben verwendet, könntest du dir natürlich mit **type()** ausgeben lassen:

```
print(type(preis))  
print(type(zahlung))
```

Das Ergebnis der Typabfrage wäre aber immer gleich und nicht gerade spannend:



```
<class 'str'>  
<class 'str'>
```

Es spielt keine Rolle, was du eingegeben hast – die Eingaben werden in jedem Fall als String in die Variablen geschrieben.

Mit Strings kann man nicht rechnen!

Um mit den Werten rechnen zu können, musst du sie in **Zahlen** umwandeln. Und mit Sicherheit wird nicht nur in ganzen Euro gezahlt. Mit einfachen Integer-Zahlen (Ganzzahlen) kommst du also nicht weit, zumindest nicht weit genug. **float** ist der Datentyp unserer Wahl.

Du könntest das **float** für die Konvertierung direkt in die bestehende Eingabe schreiben:

```
preis = float(input('Gib den Preis ein: '))  
zahlung = float(input("Der Kunde zahlt: "))
```

Schön ist anders! Und was sagt das Zen of Python dazu?

*Ist schon klar:
Beautiful is better than ugly.*



Besser: Du konvertierst den Wert der Variablen in einem **eigenen Schritt** und weist ihn dann den Variablen neu zu:

```
preis = input('Gib den Preis ein: ')
zahlung = input("Der Kunde zahlt: ")
preis = float(preis)
zahlung = float(zahlung)
```

Schon sind die eingegebenen Werte in **Fließpunktzahlen** umgewandelt. Das könntest du natürlich wieder mit **type** überprüfen.

*Na, das will ich dir
ausnahmsweise mal glauben.*



Zahlen mit Komma – statt mit dem Punkt

Python arbeitet ja bei Zahlen mit dem englischen **Fließpunkt** anstelle des hier eher üblichen Kommas. Das bedeutet, dass auch die **Eingabe** von Zahlen mit Nachkommastellen mit **Punkt** gemacht werden muss. Bei einem Preis von **7,99** müsstest du also **7.99** eingeben. Für unsere kleinen Beispiele sollte das durchaus gehen. Trotzdem ist es gut zu wissen, dass du Zahleneingaben auch mit dem **Komma** machen kannst. Und zwar, indem du Python mitteilst, dass es lokale Einstellungen verwenden soll – die für Deutschland.



Das geht? Warum hast du das nicht gleich gesagt?

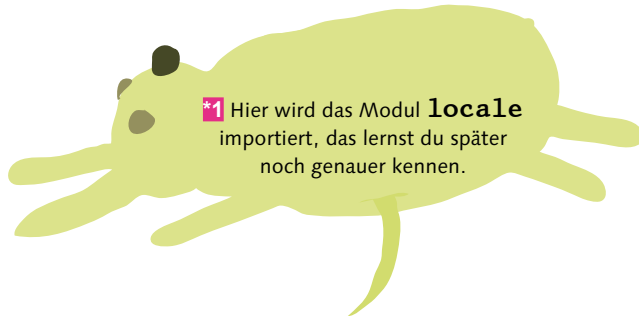


[Hintergrundinfo]

Wir müssen hier etwas vorgehen, da du ein **Modul** von Python importieren musst. Das ist nicht schwer, aber das lernst du etwas später kennen. Wir wollen ja nicht alles auf einmal machen. Deshalb schauen wir uns das nur mal so im Vorgriff an, quasi im Vorbeigehen.

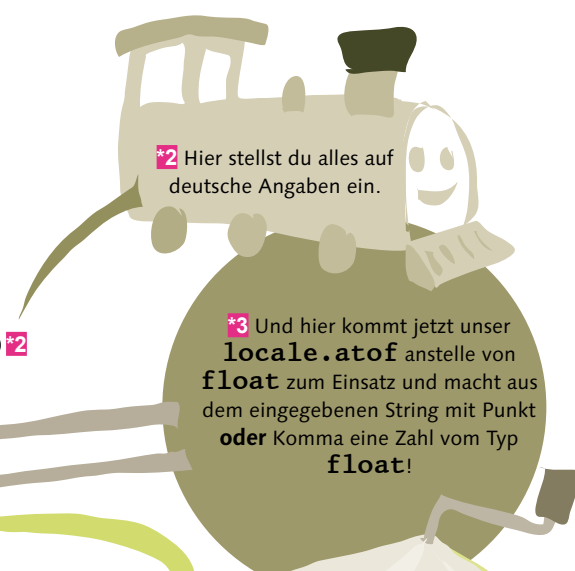


Du musst also ein **Modul** namens **locale** von Python importieren das mit lokalen Einstellungen umgehen kann. Die eingegebenen Werte wandelst du dann eben nicht mit der Funktion **float** um, sondern mit **locale.atof** aus dem importierten Modul – das kann nämlich Strings in Zahlen umwandeln, die einen Punkt **oder** ein Komma haben!



*1 Hier wird das Modul **locale** importiert, das lernst du später noch genauer kennen.

```
import locale*1
locale.setlocale(locale.LC_ALL, 'de_DE')*2
preis = input('Gib den Preis ein: ')
zahlung = input("Der Kunde zahlt: ")
preis = locale.atof(preis)*3
zahlung = locale.atof(zahlung)*3
rueckgeld = zahlung - preis*4
print('Wechselgeld:',
      locale.format_string('%.2f'*6, rueckgeld))*5
```



*2 Hier stellst du alles auf deutsche Angaben ein.

*3 Und hier kommt jetzt unser **locale.atof** anstelle von **float** zum Einsatz und macht aus dem eingegebenen String mit Punkt **oder** Komma eine Zahl vom Typ **float**!

*4 Für Python sind das jetzt alles passende **Fließpunkt-zahlen** – eben mit **Punkt**.



*6 Die etwas kryptische Angabe **'%.2f'** steht für eine Ausgabe mit zwei Stellen nach dem Komma.



*5 Sogar die Ausgabe kannst du dir mit **locale.format_string** korrekt für Deutschland ausgeben lassen – automatisch mit Komma.

Das soll an dieser Stelle genügen.

Wir machen mal ohne die lokalen Einstellungen weiter.
Aber wenn du möchtest, kannst du natürlich direkt damit arbeiten!

Die Zeit der Abrechnung ist gekommen

Zeit zu rechnen! Wir wollen jetzt wissen, wie viel Rückgeld der Kunde erhält. Dazu musst du den **Preis** von dem **gezahlten Geld** abziehen. Das Ergebnis ist das **Rückgeld**, das dem Kunden zurückgegeben werden muss. Das solltest du natürlich nicht nur berechnen, sondern auch mit **print** ausgeben:

```
rueckgeld = zahlung - preis*1  
print('Gegeben:', zahlung, 'Preis:', preis)*2  
print('Wechselgeld:', rueckgeld)*3
```

*1 Wir berechnen zuerst, wie viel Geld an den Kunden zurückgezahlt werden muss: Der Preis wird von der gezahlten Summe abgezogen. Das Ergebnis speichern wir in der Variablen **rueckgeld**.

*3 Und natürlich brauchen wir das eigentliche Ergebnis: Wie hoch ist das Wechselgeld?

*2 Mit dem ersten **print**-Befehl gibst du die eingegebenen Werte noch einmal aus. Das kann für den Benutzer hilfreich sein, um Fehler bei der Eingabe zu erkennen.

Fertig!

*Sind das nicht recht viele Variablen,
die wir für unser kurzes Programm benötigen?
Könnten wir uns da nicht was sparen?*

Natürlich. In unserem jetzigen Programm bräuchten wir die Variable **rueckgeld** nicht unbedingt. Das Ergebnis könnten wir direkt im letzten **print** berechnen:

```
print('Gegeben:', zahlung, 'Preis:', preis)  
print('Wechselgeld:', zahlung - preis)
```

Und was ist besser?

Im Zweifelsfall nimm lieber eine Variable mehr und gib ihr einen sprechenden Namen, den jeder Leser verstehen kann.

Denn das Zen of Python sagt:

»**Readability counts.** –
Lesbarkeit zählt.«





[Hintergrundinfo]

Computer sind heute derart leistungsfähig, dass es keine Rolle spielt, ob du ein paar (oder auch ein paar Dutzend) Variablen mehr oder weniger hast. Viel **wichtiger** ist doch: Machen die zusätzlichen Variablen den Code lesbarer? Und brauchst du die Variablen vielleicht noch (unverändert) an anderer Stelle?

[Zettel]

Der Programmcode und ganz besonders die Variablen sind auch immer ein Teil der Dokumentation.

Egal, ob wir das gegebene Rückgeld noch an anderer Stelle benötigen, die Variable **rueckgeld** beschreibt eindeutig den Zweck der Operation **zahlung - preis**. Das ist besser als jeder Kommentar oder jede aufwendig geschriebene Dokumentation, die ja doch keiner mehr lesen würde.

Eine Variable mehr, kann dir einiges an Arbeit sparen. Es wird sofort klar, was im Code passiert. Denk dran, deine Programme werden immer umfangreicher und komplizierter!

```
preis = input('Gib den Preis ein: ')
zahlung = input("Der Kunde zahlt: ")

preis = float(preis)
zahlung = float(zahlung)
rueckgeld = zahlung - preis

print('Gegeben:', zahlung, 'Preis:', preis)
print('Wechselgeld:', rueckgeld)
```

Hier unser kleines Programm »am Stück«. Die Leerzeilen sind nicht notwendig, können aber für eine bessere Lesbarkeit sorgen.



Damit kannst du rechnen – die Grundrechenarten

Natürlich kennt Python (nicht nur) die Grundrechenarten:

```
print(30 + 12)
print(50 - 8)
print(21 * 2)
```

Bis hierhin ist das Ergebnis immer genau **42**.