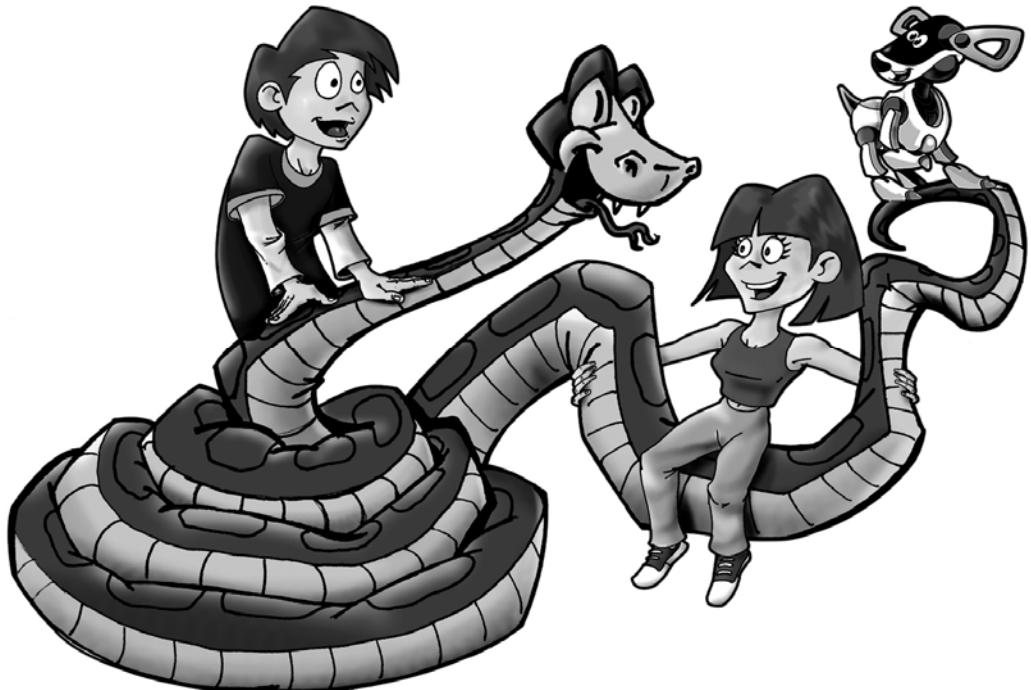


Python für Kids

Gregor Lingl



Python für Kids



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

Bei der Herstellung des Werkes haben wir uns zukunftsbewusst für
umweltverträgliche und wiederverwertbare Materialien entschieden.
Der Inhalt ist auf elementar chlорfreiem Papier gedruckt.

ISBN: 978-3-8266-8673-3
4., überarbeitete Auflage 2010

E-Mail: kundenbetreuung@hjr-verlag.de

Telefon: +49 89/2183-7928
Telefax: +49 89/2383-7620

© 2010 mitp, eine Marke der Verlagsgruppe Hüthig Jehle Rehm GmbH
Heidelberg, München, Landsberg, Frechen, Hamburg.



Dieses Werk, einschließlich aller seiner Teile, ist urheberrechtlich geschützt.
Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist
ohne Zustimmung des Verlages unzulässig und strafbar. Dies gilt insbesondere
für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die
Einspeicherung und Verarbeitung in elektronischen Systemen.

Lektorat: Katja Schrey
Sprachkorrektorat: Petra Heubach-Erdmann, Tanya Wegberg
Satz: Johann-Christian Hanke, Berlin
Printed in Germany

Inhaltsverzeichnis



Vorwort 13

Einleitung 15

Dies ist ein Buch über Programmieren 15

Ist dieses Buch das richtige für dich? 15

Die Programmiersprache ist Python 16

Turtle-Grafik 17

Fehler 18

Wie arbeitest du mit diesem Buch? 18

Zwei wichtige Informationsquellen zum Buch 20

Beim Schreiben dieses Buches ... 22

1 Was ist Programmieren? 23

Wozu dienen Programmiersprachen? 24

Unser Werkzeug: die IDLE 24

Die Arbeit mit dem interaktiven Python-Interpreter 27

Rechnen 28

sqrt ist eine Funktion 30

Schreiben 32

Dein erstes Programm 34

Wir erweitern unser erstes Programm 39

Syntax-Colouring: bunte Farben für den besseren

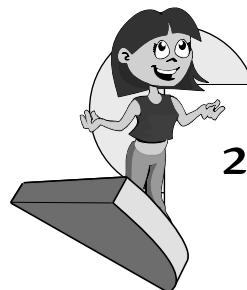
Durchblick 43

Mit Mustern arbeiten 43

Zusammenfassung 44

Zum Abschluss noch ein paar Übungsaufgaben ... 46

... und ein paar Fragen 47



2

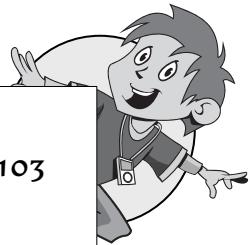
Was Schildkröten mit Grafik zu tun haben: Turtle-Grafik 49

- Die Turtle und der IPI 50
- Wie macht die Turtle das? 54
- Hilfe! 54
- Ein rotes Quadrat 55
- Füllen – und auf die Spitze! 58
- Programmcode kopieren 61
- Und jetzt drei Dreiecke! 64
- Programm codieren 67
- Noch ein paar Turtle-Grafik-Funktionen 68
- Es müssen nicht immer Ecken sein 70
- Schildkröte verstecken! Und weitere Kleinigkeiten 72
- Zusammenfassung 73
- Einige Aufgaben ... 74
- ... und einige Fragen 76

3

Namen 77

- Verschieden große Dreiecke 78
- Spielerei mit Namen 79
- Wir machen die Dreiecksseite variabel 80
- Dinge brauchen Namen 83
- Übung: Verschieden große Quadrate 88
- Und nun zu etwas ganz anderem 91
- Zahleneingaben 94
- Grafik-Programm mit Dialog 95
- Zusammenfassung 99
- Einige Aufgaben ... 100
- ... und einige Fragen: 101

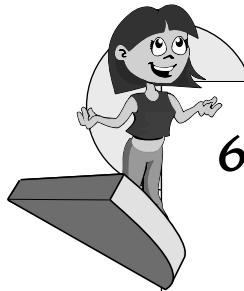


4 Wir erzeugen unsere eigenen Funktionen 103

- Vorbereitung – eine kleine Vereinfachung 104
- Wir (er)finden die Funktion dreieck 105
- Im Direktmodus Funktionen definieren 106
- Dreieck-Programm, heute neu 108
- Wie wird »dreieck07.py« ausgeführt? 109
- Noch ein Schritt weiter ... 110
- Noch eine Idee ... 112
- Welche ist die bessere Variante? 115
- Mini-Quiz 116
- Zunächst nur eine Frage 121
- Mini-Quiz erweitern 122
- Eine Funktion verwenden 125
- Mehrfachverzweigung 127
- Zusammenfassung 130
- Einige Aufgaben ... 131
- ... und einige Fragen 132

5 Funktionen mit Parametern 133

- Noch einmal Dreiecke 134
- Das geht auch mit gefüllten Dreiecken 138
- Funktionen mit mehreren Parametern 139
- Dreiecksmuster 142
- jump() 143
- Ran ans Dreiecksmuster 145
- »jump()« ist auch für später nützlich 146
- Seifenoper 149
- Zusammenfassung 153
- Einige Aufgaben ... 154
- ... und einige Fragen 155



6

Von oben nach unten und zurück 157

Aufgabenstellung: Yinyang 158

Weg 1: Top-down. Programmieren ohne Computer 158

Weg 2: Bottom-up. Schrittweise von unten nach oben 163

»jump()«, revisited 175

Zusammenfassung 176

Einige Aufgaben ... 177

... und einige Fragen 178

7

Schleifen, die zählen 179

Gestrichelte Linien 180

Die Zählschleife mit »for« 181

Faule Typen 184

Weiter mit der for-Schleife ... 187

... und zurück zur gestrichelten Linie 187

Dreiecke mit Schleife 189

Die Funktion »len()« 193

Eine Schleife für gefüllte Dreiecke 195

Ein bisschen eleganter und ein bisschen vielseitiger 199

Mini-Quiz umarbeiten 200

Zusammenfassung 205

Einige Aufgaben ... 206

... und einige Fragen 206

8

Mehr Schleifen: Friedenslogo, Superrosette 207

Friedenszeichen auf der Regenbogenfahne 208

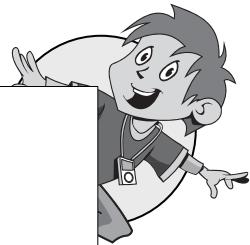
Die Regenbogenfahne 209

Das Friedenslogo 211

»tracer()« 213

n-Ecke 214

Wir machen daraus ein importierbares Modul 217



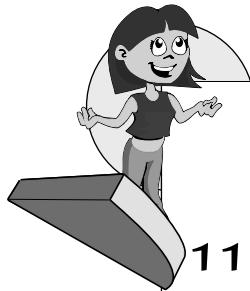
Rosetten	220
Farbenspiel	223
Farben durch Zahlen festlegen	224
Blau/Rot/Blau in einer Schleife	225
Zusammenfassung	227
Einige Aufgaben ...	227
... und einige Fragen	228

9 Der Zufall und bedingte Schleifen 229

Eine torkelnde Turtle	230
Der Zufallsgenerator »randint()«	230
Torkeln	232
»while«-Schleifen	237
Zurück zum random walk	241
Farbige Irrfahrten	246
Zusammenfassung	247
Einige Aufgaben ...	248
... und einige Fragen	248

10 Funktionen mit Wert 249

Manche Funktionen geben Objekte zurück	250
Wir definieren eine Funktion mit Rückgabewert	251
Flächeninhalt und Umfang eines Rechtecks	255
Eine Funktion, die einen String zurückgibt	257
Der Nachfolger in einer Tischrunde	259
Fak!	260
Langzahlarithmetik	264
Auch Funktionen sind Objekte	265
Laufzeitmessung	267
Was ist Nichts?	270
Zusammenfassung	273



11

Einige Aufgaben ... 273
... und einige Fragen 274

Objekte und Methoden 275

Sind Turtles Objekte? 276
Vier dynamische Spiralen 282
Sequenzen 285
Methoden von Sequenzen 294
»krange()« 302
»krange()« – als Generator ... 303
Wikipedia-Beispiel, revisited 307
Zusammenfassung 312
Einige Aufgaben ... 313
... und einige Fragen 313

12

Wörterbücher, Dateien und der alte Cäsar 315

Dictionaries 316
Verschlüsseln 320
Dateien 326
Zusammenfassung 331
Aufgaben ... 332
... und einige Fragen 332

13

Ereignisgesteuerte Programme 333

Ereignisse 334
»screen.onclick(goto)« 334
Scribble 337
Zeichenstift steuern 339
Kritzeln 340
Gefüllte Flächen 341
»undo()« und Tastatur-Ereignisse 343
Strichdicke einstellen 345
Farben 348



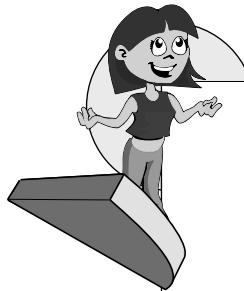
Viel mehr Farben	352
Hilfe	355
Eine andere Sorte Ereignis: Timer	357
Mach dir deine eigenen Turtle-Shapes	359
Uhr	361
Animation des Uhrzeigers	365
Datum und Uhrzeit ermitteln	366
Mehr Zeiger ...	368
Zusammenfassung	369
Aufgaben ...	370
... und einige Fragen	370

14 Neue Klassen definieren 371

Turtles, die mehr können!	372
Eine Unterklasse von Turtle	376
Namen sind Schall und Rauch, aber auch wieder nicht!	380
Der Konstruktor	383
Noch ein einfaches Beispiel: Boten	387
Zusammenfassung	394
Eine Aufgabe	395
... und nach diesem Kapitel keine Fragen	395

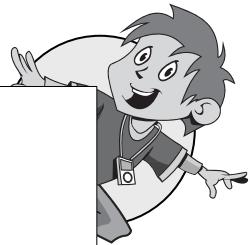
15 Moorhuhn 397

Das Spiel	398
Zunächst zwei spezielle technische Details	399
Die Bewegung der Hühner: Flug und Absturz	399
Bilder und Turtle-Grafik	401
Die Benutzeroberfläche, die Klasse MoorhuhnSpiel	403
Die Spiellogik	406
Die Klasse »Huhn«	408
Schießen und treffen	411



Fine tuning Moorhuhn	415
Konstanten	417
Cursor	418
Töne	419
Moorhuhn als selbstständig ausführbares Programm	422
Anhang A	427
Python installieren (Windows, Linux, Mac OS X)	427
Anpassung der Installation für die Arbeit mit dem Buch	430
Anhang B	435
Python-Programme ausführen	435
Anhang C	437
Was in Python 2.x anders ist	437
Anhang D	440
Tkinter-Farben	440
Anhang E	441
Weitere Informationen zu Python	441
Anhang F	442
IDLE auf zwei Arten verwenden	442
Der große Unterschied	442
Anhang G	444
Das Modul turtle.py: Die Referenz	444
I. Funktionen für die Kontrolle der Turtle	445
II. Funktionen für die Kontrolle des Turtle-Grafik-Fensters	453
III. Die Datei »turtle.cfg«	458
Stichwortverzeichnis	459

Vorwort



Ich freue mich sehr darüber, dass dieses Buch nun seine vierte Auflage erreicht. Gerichtet an Teenager (und einige pfiffige jüngere Kids) und deren Lehrer, verfolgt es einen leichtfüßigen Weg, ohne jemals kindisch zu sein oder die Dinge zu sehr zu vereinfachen. Ich bin außerdem neidisch: Ich selbst kannte mich nicht mit Computern aus, bis ich 18 wurde, und als ich zum ersten Mal lernte, einen zu programmieren, musste ich Lochkarten verwenden, um den Computer mit meinen Programmen zu füttern – die Computer füllten mehrere Räume. Wie viel einfacher doch die Dinge für die heutige Generation sind! (Sehen die Älteren das nicht immer so?) Aber ernsthaft, nicht nur die Technik, Computerprogramme zu schreiben, ist heutzutage viel unkomplizierter, sondern auch die Programmiersprachen sind viel einfacher zu verstehen.

Vor fast zwanzig Jahren, als ich Python erschuf, habe ich es nicht als Sprache für Lehrer entworfen: Ich hatte ein Werkzeug für faule Programmierer wie mich vor Augen. Aber schon bald, nachdem ich meine Erfindung in die Welt gesetzt hatte, erfuhr ich von anderen Programmierern, die ihren Kindern, Nichten und Neffen erfolgreich Python beibrachten. Das ist eigentlich gar nicht überraschend: Python verwendet viele Ideen von ABC, einer Lehrsprache, die zehn Jahre zuvor entstanden ist und ihrer Zeit weit voraus war. Ich gebe gerne zu, dass alle guten Ideen in Python von anderen Sprachen gestohlen sind. Der Clou liegt selbstverständlich darin zu entscheiden, welche der guten Ideen man stiehlt ...

Python ist in erster Linie eine Sprache für professionelle Programmierer. Aber in den letzten Jahren habe ich begriffen, dass es ebenso eine der besten Sprachen ist, um Programmieren zu lernen. Die Python-Community bemüht sich laufend darum, dass dies auch so bleibt. Wie alle lebenden Organismen entwickelt sich auch Python immer weiter. Ein großer Schritt in dieser Entwicklung ist eben erst erfolgt: die Freigabe von Python 3. Eines der Ziele von Python 3 war, viele derjenigen Eigenarten von Python zu beseitigen, die Einsteiger in die Programmierung verwirrt oder gestört haben. Dieses Buch lehrt Python 3, das beste Python, das es jemals gab. Damit bist du bereit für die Zukunft.

Ich bin sehr glücklich, dass dieses Buch ursprünglich auf Deutsch geschrieben wurde; dies ist keine schnelle Übersetzung aus dem Englischen. Der Autor hat sich offensichtlich viele Gedanken über die Vermittlung von Wissen gemacht, und Kinder und Jugendliche verdienen das Beste, was auch

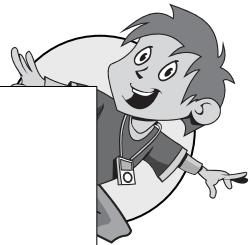


den Bezug zur eigenen Kultur – und nicht zu der von Übersee – beinhaltet. Zu guter Letzt bin ich auch sehr dankbar dafür, dass der Autor, ganz im Geiste der Open-Source-Bewegung, mit der Entwicklung einer verbesserten und erweiterten Version des Turtle-Moduls selbst einen Beitrag *für* die Python-Community geleistet hat.

Kalifornien, Oktober 2009

Guido van Rossum (Erfinder von Python)

Einleitung



Dies ist ein Buch über Programmieren

Programmieren heißt, eine Reihe von Denkweisen, Vorgangsweisen und Techniken anzuwenden, um von einer Programmidee über einen Programmamentwurf zu einem fertigen Programm zu kommen. Dazu braucht die Programmiererin oder der Programmierer eine Menge Fertigkeiten, die von der verwendeten Programmiersprache und den verwendeten Softwarewerkzeugen unabhängig sind.

In der Hauptsache will ich dir in diesem Buch solche Fertigkeiten vermitteln. Ich führe dich in kleinen Lernschritten von den einfachsten Grundlagen bis hin zur Erstellung grafischer Spielprogramme. Von Anfang an steht dabei der Begriff des »Objekts« im Mittelpunkt. Während du zunächst nur ganz einfache Objekte verwendest, Zahlen und Wörter und Grafikfunktionen, gelangst du schließlich dahin, selbst Objekte nach deinen Ideen zu erzeugen und anzuwenden, um durchaus anspruchsvolle Programmieraufgaben zu meistern.

Fast alle neuen Begriffe und Programmieranweisungen führe ich mit einfachen Beispielen aus der Grafik-Programmierung ein, so dass du direkt zu sehen kannst, was man damit machen kann und wie sie funktionieren. Das wird dir bestimmt das Verständnis dieser Dinge erleichtern.

Ist dieses Buch das richtige für dich?

Wenn du also beginnen willst, Programmieren zu lernen, dann ist dieses Buch sicher das richtige für dich.

Wenn du schon etwas programmieren kannst und nun einen Einstieg in die objektorientierte Programmierung suchst, weil du das schon länger einmal verstehen wolltest, dann kannst du die Kapitel 1 bis 10 zügig durcharbeiten und die darauf folgenden genauer. Daraus wirst du sicher Gewinn ziehen.

Wenn du dich auch mit objektorientierter Programmierung schon gut auskennst und nun Python dazulernen willst, dann suche dir ein anderes Buch



– eine Einführung in Python für Programmierer. Davon gibt es einige, doch für Einsteiger sind sie in der Regel zu schwer.

Die Programmiersprache ist Python

Natürlich braucht man zum Programmieren auch eine Programmiersprache. Ich habe für dieses Buch Python gewählt. Python hat einen äußerst weit gesteckten Bereich von Anwendungen in der professionellen Programmierung. Python ist aber auch sehr gut zum Programmierenlernen geeignet.

In diesem Buch wird Python 3 verwendet. In Python 3 wurden einige veraltete Sprachelemente des Python-2.x-Zweiges entfernt oder geändert, um die Sprache zu modernisieren. Python 3 ist also die modernste, eleganteste Python-Variante. Der Preis für diesen Fortschritt war, dass Python 3 nicht mehr 100%-ig kompatibel zu Python 2.x ist. Es *kann* daher sein, dass mit Python 3 erstellte Programme unter Python 2.x nicht korrekt laufen – und auch umgekehrt. Über einige Unterschiede zwischen Python 3.x und Python 2.x und wie man mit diesen umgeht, gibt Anhang C Auskunft.

Python befindet sich also derzeit (Oktober 2009) in einer Umstellungsphase. Man nimmt an, dass im Laufe des Jahres 2010 der Punkt erreicht werden wird, wo bereits mehr Code mit Python 3 erzeugt wird als mit Python 2.x. Mit der Entscheidung für Python 3 wollte ich nicht nur sicherstellen, dass die jungen Leserinnen und Leser dieses Buches das beste Python lernen, sondern auch das, dem die Zukunft gehört.

Hier einige wichtige Eigenschaften von Python:

- ❖ Python ist eine sehr hoch entwickelte, sehr leistungsfähige Programmiersprache. Sie ist leicht zu lernen, zu lesen und zu verwenden. Python-Programme sind in der Regel viel kürzer als Programme, die dasselbe tun, aber in anderen Sprachen wie Java, C++ und so weiter geschrieben sind. Weil Python-Programme so leicht zu lesen und zu verstehen sind, enthalten sie auch weniger Fehler und können leichter erweitert und gewartet werden.
- ❖ Python ist eine von Grund auf objektorientierte Sprache. Die Programmierung von Objekten und Klassen, wie sie in der modernen Softwaretechnik gebräuchlich ist, ist daher in Python besonders einfach. Wie das geht, erfährst du allerdings erst ab Kapitel 11. Bis dahin geht es um die einfacheren Grundlagen des Programmierens.



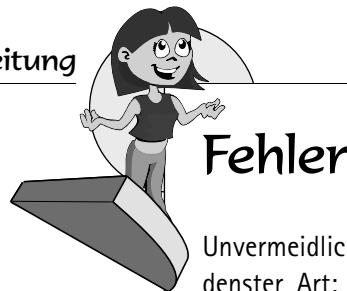
- ❖ Und vor allem ist Python eine Sprache, die mit einem interaktiven Interpreter daherkommt, den du gleich im ersten Kapitel kennen lernen wirst. Mit dem kannst du deine Ideen sofort ausprobieren und kontrollieren und Fragen sofort klären. Er ist ein unschätzbar wertvolles Werkzeug zum Lernen wie zum Arbeiten.
- ❖ Python ist eine plattformübergreifende Sprache. Das heißt, Python ist für Windows-Systeme genauso erhältlich wie für Linux oder Macs. Python-Programme laufen auf all diesen Rechnern. Die Beispiele in diesem Buch sind alle auf einem Windows-Rechner erstellt.
- ❖ Python ist eine Sprache, die aus der Open-Source-Bewegung kommt (wie Linux). Daher ist die jeweils neueste Version stets frei aus dem Internet zu haben. Du brauchst dich also nicht mit veralteten oder beschränkten Versionen abzumühen, sondern bist gleich auf dem neuesten Stand der Technik. (Und du kannst die Python-Software auch an deine Freundinnen und Freunde weitergeben, ohne dabei ein Gesetz zu übertreten!)
- ❖ Python kommt mit einer »Standard-Bibliothek«, die eine große Anzahl von vorgefertigten »Modulen« zur Programmierung von Anwendungen enthält. Außerdem gibt es zahllose, in Python geschriebene und frei erhältliche Softwarepakete, die du in deine Programme einbinden kannst.

Turtle-Grafik

Ein Modul, von dem ich in diesem Buch viel Gebrauch machen werde, ist das Turtle-Grafik-Modul namens `turtle`. Ich habe dieses Modul zunächst eigens für dieses Buch geschrieben, als eine erweiterte und verbesserte Fassung des Turtle-Grafik-Moduls, das in älteren Versionen von Python enthalten war. Seit Python 2.6 ist dieses Modul in die Standard-Bibliothek aufgenommen worden. Ich stelle es dir in Kapitel 2 vor.

Turtle-Grafik ist eine besonders anschauliche Form der Grafik, die vor etwa 20 Jahren für junge Programmieranfänger zusammen mit der Programmiersprache LOGO erdacht wurde. Du brauchst fast keine Mathematik, um mit Turtle-Grafik zu arbeiten und interessante geometrische Forschungen anzustellen.

Ich verwende sie in diesem Buch aber vor allem aus einem anderen Grund: Du kannst mit ihr den Ablauf deiner Programme direkt auf dem Bildschirm verfolgen. So erhältst du leicht Aufschluss, ob deine Ideen zielführend sind und wie du sie eventuell ändern musst, um zum Ziel zu kommen. Und so ganz nebenbei erhältst du auch noch lustige Grafiken.



Fehler

Unvermeidlich wirst du bei der Arbeit viele Fehler machen, Fehler verschiedenster Art: Tippfehler, logische Fehler, Bedienungsfehler und so weiter. Betrachte Fehler nicht als ein Übel, sondern als eine wesentliche Quelle des Lernens.

Es gibt keine Programmierer, die keine Fehler machen. (Es gibt auch kein größeres Stück Software, das keine Fehler enthält! Ja, ich finde immer noch ab und zu Fehler in meinem turtle-Modul, und sogar dieses Buch wird bestimmt eine Menge Fehler enthalten.) Es kommt aber darauf an, die Fehler genau und geduldig zu untersuchen, um zu sehen, was der Grund war. Hast du einmal verstanden, wie es zu einem Fehler kam, wird es dir leichter fallen, ihn zukünftig zu vermeiden.

Wie arbeitest du mit diesem Buch?

Dieses Buch ist ein Arbeitsbuch. Das heißt, du musst es von vorne bis hinten durcharbeiten. Es ist kein Nachschlagewerk.

Das Buch ist voll von Arbeitsanweisungen und Arbeitsvorschlägen. Wenn du sie alle durcharbeitest, wirst du verschiedene Arbeitsweisen kennen lernen, die dich »von der Idee zum Programm« führen. Manche werden dir mehr zusagen, manche weniger. Wenn du später einmal selbstständig programmierst, wirst du die Wahl haben, welche Verfahren du anwendest. Du solltest sie jedenfalls alle kennen und einmal ausprobiert haben, dann hast du die Freiheit der Auswahl.

Fragen und Aufgaben

Am Ende jedes Kapitels findest du einen Abschnitt mit Wiederholungsfragen und einige Übungsaufgaben. Mit den Wiederholungsfragen kannst du überprüfen, ob du den Inhalt eines Kapitels gut erfasst hast. (Die richtigen Antworten findest du auf der Buch-CD.)

Die Übungsaufgaben solltest du ebenfalls zu lösen versuchen. Programmierenlernen erfordert unter anderem auch viel Übung. (Das ist ähnlich, wie wenn du Skateboard fahren oder Klavier spielen lernst.) Jede gelöste Übungsaufgabe ist ein tolles Erfolgserlebnis. Und wenn du eine Aufgabe nicht schaffst, kannst du doch das Ergebnis deiner Bemühungen mit einer

Wie arbeitest du mit diesem Buch?

Musterlösung vergleichen – und wieder einmal aus Fehlern lernen – auch sehr wertvoll. (Musterlösungen für die Aufgaben findest du ebenfalls auf der Buch-CD.)

Um dir den Weg durch das Buch zu erleichtern, sind manche Abschnitte durch besondere Symbole gekennzeichnet:



Arbeitsschritte

Wenn du dieses Zeichen siehst, heißt das: Es gibt etwas zu tun. Mit solchen Arbeitsschritten kommst du deinem nächsten Ziel näher.

» Mach mit!

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

»Mach mit!«, gefolgt von >>> ... bedeutet immer einen besonderen Typ von Arbeitsschritten, nämlich dass du Eingaben für den interaktiven Python-Interpreter nachvollziehen sollst.

Probleme, Notfälle, Vertiefungen

Wenn die Gefahr besteht, an einer Stelle etwas falsch zu machen, oder wenn es wichtig ist, irgendwelche Besonderheiten zu beachten, dann steht dir Buffi mit Hilfe und Rat zur Seite.

Buffi tritt auch in Erscheinung, wenn es um ausführliche Erläuterungen oder weitergehende Informationen allgemeiner Art geht.

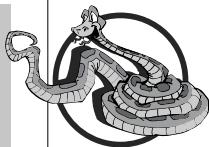


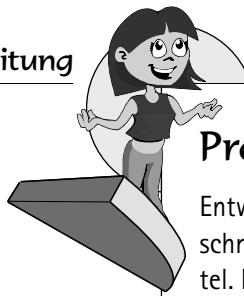
Wichtige Stellen im Buch

Nicht selten findest du ein Ausrufezeichen am Textrand. Dann ist das eine Stelle, an der etwas besonders Wichtiges steht.



Manchmal sind spezielle Dinge zu erklären, die Besonderheiten der Sprache Python betreffen. (So etwas ist dann meistens nicht ganz leicht in eine andere Programmiersprache zu übernehmen.) Für diese Dinge hat sich Buffi die weise Mitarbeiterin Clara Pythia engagiert.





Programmentwürfe

Entwürfe von Programmen oder auch von Abschnitten von Programmen schreibe ich mir »im richtigen Leben« meistens mit Bleistift auf einen Zettel. Dies wird im Buch durch eine eigene Schrift angedeutet:

Programmentwürfe
Auf einem Blatt Papier,
in gewöhnlicher Handschrift.

Zuletzt kommen beim Programmieren immer wieder die gleichen Denkmuster und die gleichen Codemuster vor. Auch die werden im Text durch Kästen hervorgehoben. Präge dir solche Muster gut ein!

Muster

Muster 1: Einfaches Python-Skript

```
from modul import *      Wird nicht in jedem Programm gebraucht
```

```
Anweisung 1  
Anweisung 2  
...
```

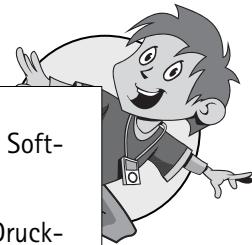
Zwei wichtige Informationsquellen zum Buch

Zu *Python für Kids* gibt es zwei weitere Informationsquellen, die du als wichtige Bestandteile unbedingt nutzen solltest:

Die CD-ROM zum Buch

Auf der CD-ROM zum Buch findest du:

- ❖ eine Datei `index.html`, die du mit deinem Webbrowser öffnen und ansehen kannst. Sie enthält wichtige Informationen zum Start mit die-



sem Buch – du solltest sie jedenfalls ansehen, bevor du mit der Software-Installation und der Arbeit mit dem Buch beginnst.

- ❖ Python-Software für Windows-Systeme und Mac OS X: die (bei Drucklegung des Buches) aktuelle Version Python 3.1.1 vom 17. August 2009. Bevor du mit Python arbeiten kannst, musst du diese Software auf deinem Rechner installieren. Halte dich dazu *unbedingt* an die Anleitung in *Anhang A zur Installation von Python*. Linux-Systeme brauchen an die jeweilige Distribution angepasste Versionen von Python. Diese können in den meisten Distributionen mit dem mitgelieferten Software-Installationsprogramm direkt aus dem Internet installiert beziehungsweise aktualisiert werden. Auf der CD findest du aber auch den Quellcode von Python 3.1.1 und eine (beispielhafte) Anleitung, wie du ihn unter Ubuntu 9.04 zu einem lauffähigen Python-Interpreter kompilieren kannst. Das Verfahren sollte leicht auf andere Linux-Distributionen übertragbar sein.
- ❖ alle im Buch vorgestellten Python-Programme sowie die Lösungen zu vielen Übungsaufgaben.
- ❖ zu jedem Kapitel die richtigen Antworten auf die Wiederholungsfragen.
- ❖ einige Hilfsdateien, zum Beispiel für die Konfiguration des turtle-Moduls.

Um Python für die Arbeit mit dem Buch einzurichten, sind einige Schritte zusätzlich zur Standardinstallation erforderlich. Sie werden in Anhang A beschrieben.



Die Website zum Buch: python4kids.net

Unter <http://python4kids.net> findest du eine Website zu diesem Buch. Auf dieser Seite möchte ich verschiedenes Material zum Buch veröffentlichen:

- ❖ Fehlerberichtigungen
- ❖ Vielleicht Beispielprogramme und Ideen, die von Leserinnen und Lesern beigesteuert wurden
- ❖ Antworten auf FAQs, »frequently asked questions«
- ❖ Hinweise und Links auf weiterführende Informationen über Python im Internet
- ❖ Schließlich findest du dort auch eine E-Mail-Adresse, unter der du mich erreichen kannst, wenn du bei der Arbeit stecken bleibst und dringend Hilfe brauchst.



Beim Schreiben dieses Buches ...

... haben mich viele Freunde, Freundinnen, Kollegen und Kolleginnen unterstützt. Ganz besonderen Dank schulde ich Kurt Winterstein für das sorgfältige Durcharbeiten vieler Beispielprogramme, Wolfgang Urban, Erwin Hasenleithner und Patrizia Piculjan für das sorgfältige Lesen des Manuskripts und viele hilfreiche Diskussionen. Ihren Ratschlägen verdankt das Buch sehr viel. Meiner Frau Karin möchte ich für das sorgfältige Durcharbeiten der ersten sechs Kapitel danken und mehr noch für die Geduld, mit der sie meine häufige praktische und gedankliche Abwesenheit während der Arbeitsphasen für die Neuauflagen erträgt. Gerhard Stenger danke ich für die Anfertigung einer Reihe von Handzeichnungen.

Meinen Schülern aus dem Gymnasium Bernoullistraße und dem Realgymnasium Schuhmeierplatz danke ich für das Auffinden einer ganzen Reihe von Fehlern in der 2. bzw. 3. Auflage und das sorgfältige Testen des turtle-Moduls. Ihre Aufmerksamkeit hat viel zu den Verbesserungen der neueren Auflagen beigetragen.

Eine wichtige Rolle haben auch Diskussionen in verschiedenen Mailing-Listen der Python Community für mich gespielt, an die ich mich oft mit offenen Fragen gewendet habe. Besonders Danny Yoo, Alan Gauld und Magnus Lyckå von der Tutor-Liste (tutor@python.org) haben mir mit großer Geduld und brillanten Erklärungen Anregungen geliefert.

Schließlich möchte ich mich auch noch bei Frau Katja Schrey und Frau Sabine Schulz vom bhv-Verlag bedanken, ohne deren Einsatz und ermunternden Zuspruch es dieses Buch wohl kaum bis zur vierten Auflage geschafft hätte, sowie bei Herrn Johann-Christian Hanke, der das Buch wieder mit großem Engagement und Einfühlungsvermögen neu gesetzt hat.

Wien, Oktober 2009

Gregor Lingl

1

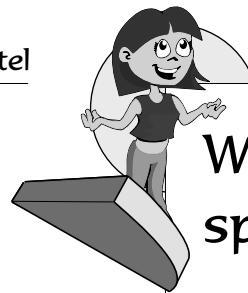


Was ist Programmieren?

Programmieren ist ... wenn du einem Computer Anweisungen gibst und er tut, was du willst. Das klingt einfacher, als es ist. Aber keine Angst, Computer die richtigen Anweisungen zu geben, kann jeder lernen. Zunächst musst du dazu eine Sprache »sprechen«, die dein Computer auch versteht. Du hast dich dafür entschieden, den langen Weg zur Programmiererin oder zum Programmierer mit der Programmiersprache Python zu beginnen. Das wird dir deinen Weg sehr erleichtern! Das erste Stück dieses Weges werden wir gemeinsam gehen.

In diesem Kapitel wirst du ...

- ◎ die Programmierumgebung IDLE in Betrieb nehmen, die du brauchst, um mit Python zu programmieren und mit ihr deine ersten praktischen Erfahrungen sammeln.
- ◎ das PYTHON SHELL-Fenster kennen lernen und beginnen, mit dem interaktiven Python-Interpreter die Programmiersprache Python zu erforschen.
- ◎ mit Python rechnen und schreiben.
- ◎ das Editor-Fenster der IDLE kennen lernen, das du zum Erstellen von Programmen brauchst und
- ◎ schließlich dein erstes Programm schreiben und gleich noch weiterentwickeln.



Wozu dienen Programmiersprachen?

Programmiersprachen sind dazu gemacht, einem Computer Anweisungen zu geben, damit er bestimmte Aufgaben ausführen kann. Solche Aufgaben sind etwa: Eine Meldung anzeigen, eine Rechnung ausführen, die dir selbst zu langweilig ist, oder eine bunte Grafik zeichnen.

Leistungsfähige Computerprogramme – Textverarbeitungsprogramme, Bildbearbeitungsprogramme, Computerspiele – bestehen aus einer Vielzahl solcher Anweisungen, sogar bis zu mehreren Millionen. Gemeinsam ist allen Programmen, kleinen wie großen, dass die Anweisungen nach ganz genauen Regeln aufgeschrieben werden müssen.

In diesem ersten Kapitel wirst du einige Anweisungen der Programmiersprache Python kennen lernen und auch gleich ausprobieren, denn das geht mit Python ganz leicht. Am Ende wirst du dann aus ein paar solcher Anweisungen dein erstes Programm zusammenbasteln.

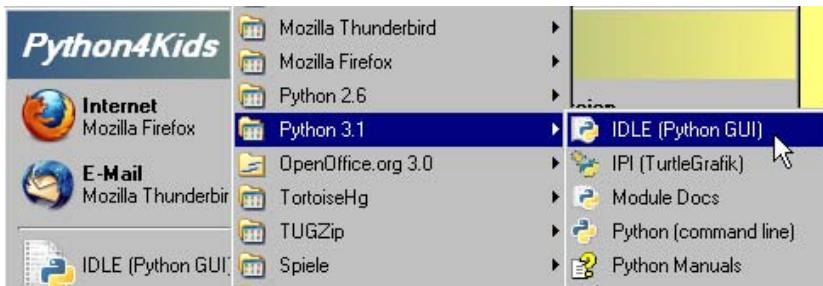
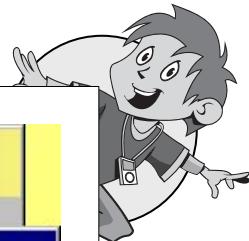
Nun ist es aber so, dass die CPU deines Computers, sein eigentliches Herzstück und Arbeitstier, nicht Python kann. Die CPU ist ein Mikroprozessor, also eine Maschine und versteht nur Maschinensprache, z.B.: 000010100110101101001010100111. *Maschinensprache* braucht nur zwei Zeichen, 0 und 1. Das ist ihr ganzes Alphabet. Es ist so einfach, dass die vielen Millionen elektronischen Schalter in der CPU in rasender Geschwindigkeit Anweisungen abarbeiten können: Strom ein – Strom aus.

Damit dein Computer aber deine Python-Anweisungen verstehen und ausführen kann, braucht er einen Helfer, der ihm die Anweisungen dieser *höheren Programmiersprache* in Maschinensprache übersetzt. Dieser Helfer ist selbst ein Computerprogramm und bei der Software dabei, die du mit dem Python-System auf deinem Computer installiert hast: Der Name des Programms ist – wenig verwunderlich – Python und es wird im Computerfachchinesisch als *Python-Interpreter* bezeichnet.

Unser Werkzeug: die IDLE

Nach der Installation von Python, wie sie im Anhang A beschrieben ist, findest du in deinem Startmenü START|ALLE PROGRAMME|PYTHON3.1|IDLE (PYTHON GUI).

Unser Werkzeug: die IDLE



Die IDLE wird vom Startmenü aus ...

... oder über ein Desktop-Icon gestartet.

- Starte IDLE (PYTHON GUI). Nun erscheint folgendes Fenster auf dem Bildschirm:

The screenshot shows the Python Shell window with the following content:
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> |
Ln: 3 Col: 4

Zur integrierten Entwicklungsumgebung IDLE gehört ein SHELL-Fenster.

IDLE ist ein Werkzeug zum Programmieren mit Python. IDLE ist die Abkürzung für »Integrated Development Environment«, auf Deutsch: integrierte Entwicklungsumgebung. Hat aber daneben noch andere Bedeutungen. (Sieh mal in einem Englischwörterbuch nach oder schau dir auf You Tube ein paar von Monty Python's Sketches an!)

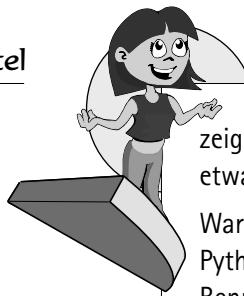
»Integriert« bedeutet, dass mehrere Software-Werkzeuge zu einem Programm zusammengesetzt sind. Während unserer Arbeit mit Python werden wir häufig mit IDLE arbeiten. Das Schönste an IDLE ist, dass sie sich gleich mit einem interaktiven Python-Interpreter meldet, der

>>>

ins PYTHON SHELL-Fenster schreibt. Das ist sein Bereitschaftszeichen. Heute sagt man dazu meistens *Prompt*, weil sich dieses Wort, das im Englischen für das Bereitschaftszeichen verwendet wird, bereits im Deutschen eingebürgert hat. Rechts vom Prompt siehst du einen blinkenden Cursor |. So



1



zeigt der interaktive Python-Interpreter an, dass er darauf wartet, dass du etwas eingibst.

Warum Shell? Die Python-Shell ist quasi die Hülle, oder Schale, die den Python-Kern, den Interpreter, umgibt und die Verbindung zwischen dir, dem Benutzer, und dem Interpreter herstellt. Du gibst Anweisungen ein und der Python-Interpreter führt sie aus und gibt Antworten zurück.

Sehen wir mal nach, ob er wenigstens ganz einfache Dinge versteht.

➤ Tippe folgende Frage ein:

```
>>> Wie viel ist eins und eins?
```

Ups!

Syntax ist die Rechtschreibung bei einer Programmiersprache.

The screenshot shows a Windows-style window titled "Python Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The title bar displays "Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32". The main window contains the following text:

```
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> Wie viel ist eins und eins?
SyntaxError: invalid syntax (<pyshell#0>, line 1)
>>> |
```

The line "Wie viel ist eins und eins?" is highlighted in red, indicating a syntax error. The status bar at the bottom right shows "Ln: 5 Col: 4".

Wir haben in unserem Eifer übersehen, dass der interaktive Python-Interpreter nicht Deutsch kann, sondern nur Python. Die gestellte Frage hält nun mal nicht die »Rechtschreibregeln« von Python ein. Im Computer-Kauderwelsch sagt man: Die Eingabe entspricht nicht der *Syntax* von Python. Deswegen beklagt der interaktive Python-Interpreter einen *Syntaxfehler* gleich in Zeile 1!

➤ Also fragen wir anders herum:

```
>>> 1 + 1
2
>>>
```

Hmmm! Das hat der interaktive Python-Interpreter verstanden.

Der gültige arithmetische Ausdruck $1+1$ ist offenbar *auch* ein gültiger *Python-Ausdruck*. Muss wohl daran liegen, dass die Sprache der Mathematik international ist.

Kann der interaktive Python-Interpreter auf diesem Sektor noch mehr?

Die Arbeit mit dem interaktiven Python-Interpreter



Der Python-Interpreter meldet sich im PYTHON SHELL-Fenster mit dem Bereitschaftszeichen:

>>>

Die Arbeit mit ihm besteht darin, dass man nach diesem Bereitschaftszeichen einen *Python-Ausdruck* oder eine *Python-Anweisung* eingibt. Er wertet die Eingabe aus und schreibt dann darunter eine Antwort in das PYTHON SHELL-Fenster. Auf diese Weise kannst du lernen und erforschen, wie Python funktioniert.



Du wirst im Folgenden viele »interaktive Übungen« mit dem interaktiven Python-Interpreter machen. Ich schreibe dann einfach:

➤ Mach mit!

Dann kommt immer eine Folge von Eingaben für den Python-Interpreter, die hinter dem Prompt

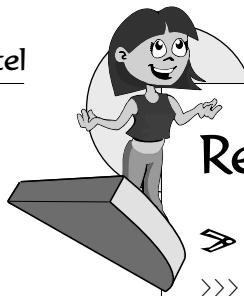
>>>

beginnen. Dazwischen streue ich Bemerkungen und Erklärungen ein. Das Beste ist, wenn du diese Eingaben einfach eintippst und prüfst, ob sie bei dir dieselben Ergebnisse liefern. Auf diese Weise lernst du, wie man Python-Anweisungen richtig schreibt und wahrscheinlich auch, welche Fehler man leicht macht und wie man sie vermeidet. Wenn du dann auch noch darüber nachdenkst, warum deine Eingaben gerade die Ergebnisse liefern, die im Fenster erscheinen, dann lernst du auf diese Weise Python verstehen.



Also machen wir unsere erste interaktive Sitzung mit dem Python-Interpreter!

1



Rechnen

» Mach mit!

>>> 3 * 4

12

Der Python-Interpreter kann rechnen. Er wertet arithmetische Ausdrücke aus.

The screenshot shows the Python Shell window with the following text:

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32
bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> Wie viel ist eins und eins?
SyntaxError: invalid syntax (<pyshell#0>, line 1)
>>> 1 + 1
2
>>> 3 * 4
12
>>> |
```

Ln: 9 Col: 4

>>> 13 + 4 * 3

25

>>> (13 + 4) * 3

51

>>> (3 - 5) * (13 + 4)

-34

Mit *arithmetischen Ausdrücken* kann der Python-Interpreter offenbar gut umgehen. Kennt sogar die Vorrangregeln und die Klammerregel. Ist mindestens genau so gut wie dein Taschenrechner!

In Python gibt es folgende Rechenzeichen (oder Rechenoperatoren):

+ - * / // % **

Einige kommen dir vielleicht spanisch vor, aber du wirst sie alle in diesem Buch noch kennen lernen. Wenn du Lust hast, kannst du jetzt schon mit ihnen experimentieren. Gib dem Python-Interpreter einfach ein paar Ausdrücke mit diesen Rechenzeichen ein. Vielleicht findest du heraus, zu welchen Rechenoperationen sie gehören?

Ob Python auch Wurzeln ziehen kann?

Rechnen



Na ja – so viel Mathe braucht man ja nicht immer. Aber doch immer wieder. Deshalb hält Python einige mathematische Funktionen nicht ständig bereit, sondern hat sie in einem eigenen *Modul* zusammengefasst, das bei Bedarf geladen werden kann. Dieses Modul heißt `math`. Dort ist auch die Quadratwurzel drin. Wenn du sie verwenden willst, musst du die Funktionen aus diesem Modul *importieren*, das heißt: in den Arbeitsspeicher laden. Das geschieht mit der folgenden Python-Anweisung:

```
>>> from math import *
```

Beachte die Kleinschreibung (genauere Erklärung weiter hinten in diesem Kapitel)!

Nun berechnen wir die Wurzel aus 4:

```
>>> wurzel(4)
```

A screenshot of a Windows-style application window titled "Python Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main area contains the following Python session:

```
>>> 1 + 1
2
>>> 3 * 4
12
>>> 13 + 4 * 3
25
>>> (13 + 4) * 3
51
>>> (3 - 5) * (13 + 4)
-34
>>> from math import *
>>> wurzel(4)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    wurzel(4)
NameError: name 'wurzel' is not defined
>>>
```

The text "Lnr. 21 Col. 4" is visible at the bottom right of the window.

Eine Fehlermeldung:
Die Art des Fehlers steht in
der letzten Zeile.

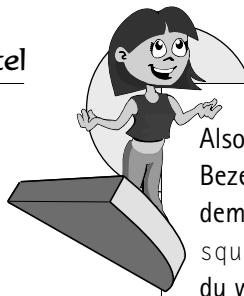
Ups! Schon wieder eine *Fehlermeldung*! Damit musst du umgehen lernen! Fehlermachen ist beim Programmieren so unvermeidlich und gleichzeitig so wichtig fürs Weiterkommen wie beim Skaten!

Für den Anfang wird es wohl das Beste sein, dass du dir in solchen Fehlermeldungen nur die letzte Zeile ansiehst:

```
NameError: name 'wurzel' is not defined
```

Das erste Wort gibt immer die Art des Fehlers an: `NameError`. Das heißt, dass in der Eingabe ein *Name* vorgekommen ist, den der Python-Interpreter nicht kennt. Welcher das war, schreibt er dir auch hin: `wurzel!`





Also ist alles meine Schuld! Ich hätte dir gleich sagen sollen, dass sich die Bezeichnungen in Python – wie in fast allen Programmiersprachen – aus dem Englischen ableiten. Und im Englischen heißt *Quadratwurzel* *sqraroot*. Die entsprechende Python-Funktion heißt aber `sqrt`, damit du weniger tippen musst!

```
>>> sqrt(4)  
2.0  
>>>
```

Richtig! Und doch ist wieder etwas Neues dabei! Eine Kommazahl kommt als Ergebnis heraus! Also kann Python auch mit Kommazahlen rechnen?

Beachte: Das Dezimalkomma ist ein Punkt, kein Beistrich!

```
>>> 3.125 * 0.8  
2.5  
>>> sqrt(2)  
1.4142135623730951  
>>> 2 * sqrt(2)  
2.8284271247461903
```

Fällt dir was auf? Hättest du das mit der Hand (oder im Kopf) gerechnet, dann hättest du ein anderes Ergebnis erhalten. (Wenn auch nur ein bisschen anders.)

```
>>> sqrt(2)*sqrt(2)  
2.0000000000000004
```

Diese Beispiele zeigen dir, dass das Rechnen mit Kommazahlen auf Computern unvermeidlich mit kleinen Ungenauigkeiten verbunden ist. Sie entstehen aus Fehlern, die beim Auf- und Abrunden im Mikroprozessor auftreten. (Das gilt in gleichem Maße für deinen Taschenrechner, auch wenn der bei der zweiten Rechnung den Rundungsfehler verschweigen würde, weil er nur zehn oder zwölf Stellen anzeigt, obwohl er intern mit mehr Stellen arbeitet.)

sqrt ist eine Funktion

Wahrscheinlich ist dir aufgefallen, dass ich hier die Zahl 2, aus der die Wurzel gezogen wurde, in runde Klammern geschrieben habe. Das habe ich nicht gemacht, weil es netter aussieht, sondern aus folgendem Grund: `sqrt` ist der Name einer *Funktion*, der Quadratwurzel-Funktion. In Python wimmelt es von Funktionen. Sie spielen eine ganz wichtige Rolle und du wirst bald selber in die Lage kommen, welche zu programmieren.

sqrt ist eine Funktion



Der Ausdruck `sqrt(2)` ist ein Aufruf der Funktion `sqrt` aus dem Modul `math`. Damit diese Funktion ihre Arbeit ausführen kann, braucht sie eine bestimmte Information, nämlich die Angabe der Zahl, aus der die Wurzel gezogen werden soll. Diese Information muss ihr beim Aufruf übergeben werden. Man sagt: Beim *Funktionsaufruf* `sqrt(2)` wird der *Funktion* `sqrt` der Wert 2 als *Argument* oder *Eingabewert* übergeben.

Um eine Funktion richtig verwenden zu können, musst du als Programmiererin oder Programmierer wissen, wie viel und welche Art von Information die Funktion braucht. Das heißt, du musst wissen, wie viele Argumente du der Funktion beim Aufruf übergeben musst und von welcher Art diese Argumente sein müssen.

Wenn du weißt, dass `sqrt()` als Argument eine Zahl verlangt, wirst du wohl kaum auf die Idee kommen, zu schreiben:

```
>>> sqrt("ziemlich viel")      # Unsinn!!
```

Wie du gleich sehen wirst, gibt es Funktionen, die mehr als ein Argument verlangen, aber auch solche, die ganz ohne Argumente auskommen.

Die Funktion `sqrt` braucht aber nicht nur einen Eingabewert, sondern sie liefert auch einen *Rückgabewert* ab, zum Beispiel an die Python-Shell, die ihn sofort in das Shell-Fenster schreibt:

```
>>> sqrt(2)
1.4142135623730951
```

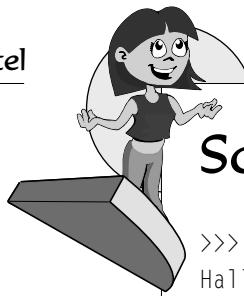
oder in einen Rechenausdruck einsetzt, wie in den anderen Beispielen weiter oben.

Wie du ebenfalls gleich sehen wirst, gibt es in Python auch Funktionen, die keinen Rückgabewert haben. (Ups! Das ist jetzt ein bisschen oberflächlich. Warte ab, in Kapitel 10 kommt das Thema noch einmal, dann viel genauer.)



Da hab' ich noch eine Idee: Hast du nicht zufällig eine unfertige Mathe-Hausaufgabe rumliegen? Ich kenne das, der Computer hat ja oft eine größere Anziehungskraft als Hausaufgaben. Wenn ja, dann hole sie dir rasch her und verwende zur Lösung der Aufgaben doch Python als Ersatz für deinen Taschenrechner – sehr praktisch, zwei Fliegen auf einen Schlag! Hausaufgabe erledigt und Python besser kennen gelernt.

1



Schreiben

```
>>> print("Hallo Große! Du wirst sehen, Python macht Spaß!")  
Hallo Große! Du wirst sehen, Python macht Spaß!  
>>>
```

Da haben wir schon wieder eine neue Python-Anweisung verwendet: die `print`-Anweisung. Sind dir die runden Klammern aufgefallen? Die `print`-Anweisung ist ebenfalls ein Funktionsaufruf: ein Aufruf der `print()`-Funktion. Und – gerade war davon die Rede – die `print()`-Funktion hat keinen Rückgabewert.

Die `print()`-Funktion dient zur Ausgabe von Ausdrücken auf dem Bildschirm. Im obigen Beispiel hat sie einen Text ausgegeben. Texte sind Folgen von Zeichen (Buchstaben, Ziffern, Leer-, Satz- und Sonderzeichen), die zwischen Anführungsstrichen stehen müssen. Diese zeigen dem Python-Interpreter an, dass diese Zeichen keine Namen oder Ausdrücke mit anderer Bedeutung sind, und er schreibt sie einfach buchstäblich – also Zeichen für Zeichen – hin. Solche Folgen von Zeichen nennt man im Computerlatein *Zeichenketten* oder *Strings*. (Merkwürdig, dass das Computerlatein meistens aus dem Englischen kommt!)

```
>>> print("3 * 12")  
3 * 12  
>>> print(3 * 12)  
36
```

Das ist ein wichtiger Unterschied: "3*12" ist ein String und wird von `print()` Zeichen für Zeichen hingeschrieben. $3*12$ ist dagegen ein arithmetischer Ausdruck. Er wird (vom Python-Interpreter) zuerst ausgerechnet und das Ergebnis wird dann auf dem Bildschirm ausgegeben.

```
>>> print(36)  
36  
>>> print("36")  
36
```

Solltest du nun meinen, dass wenigstens 36 und "36" das Gleiche sind, dann hilft sicher Folgendes, um dich vom Gegenteil zu überzeugen:

```
>>> print(36 * 10)  
360  
>>> print("36" * 10)  
36363636363636363636
```



Die `print()`-Funktion hat noch eine weitere Besonderheit: sie kann beliebig viele Eingabewerte verarbeiten. Diese müssen durch Beistriche getrennt nebeneinander geschrieben werden.

Somit kann `print()` auch mehrere Dinge nebeneinander ausgeben:

```
>>> print(1, 2, 3 * 4)
1 2 12
>>> print("3 * 12 =", 3 * 12)
3 * 12 = 36
>>> print("3 * 12 =", 3 * 12, "und 4 * 12 =", 4 * 12)
3 * 12 = 36 und 4 * 12 = 48
```

Zähl nach: Die letzte `print`-Anweisung hat vier Dinge als Eingabewerte übernommen: zwei Strings und zwei Zahlen, die aus arithmetischen Ausdrücken berechnet wurden. Diese vier Dinge hat sie dann auf den Bildschirm geschrieben. (Das ist ja schließlich der Zweck der `print`-Anweisung.)

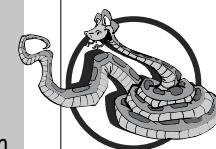
Clara Pythias Python-Special

Vielleicht willst du einmal, dass `print()` so etwas ausgibt:

Uwe rief "Oh!" und erbleichte.

So geht das leider nicht:

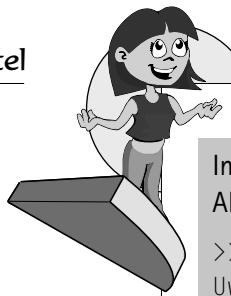
```
>>> print("Uwe rief "Oh!" und erbleichte.")
SyntaxError: invalid syntax (<pyshell#10>, line 1)
>>>
```



Dieser Fehler kündigt sich bereits während der Eingabe durch die Farben an. Der Python-Interpreter liest den String "Uwe rief " und weiß dann mit Oh! nichts mehr anzufangen.

Mit Python gibt's da einen leichten Ausweg: Du darfst Strings auch mit ' ' begrenzen. Das einfache Hochkomma ' ist das Zeichen, das auf den meisten Tastaturen auf einer Taste gemeinsam mit dem # zu finden ist.

```
>>> print('Uwe!')
Uwe!
```



Innerhalb solcher Strings können ohne Probleme " " verwendet werden.
Also:

```
>>> print('Uwe rief "Oh!" und erbleichte.')
Uwe rief "Oh!" und erbleichte.
```

Ich rate dir aber, wann immer möglich, einheitlich die " " zu verwenden.

Dein erstes Programm

Wir haben bisher ausschließlich im Direktmodus gearbeitet, das heißt: Wir haben unsere Anweisungen direkt dem Python-Interpreter eingegeben. Diese Anweisungen wurden sofort ausgeführt. Wenn du auf diese Weise heute mit vielen interaktiven Grafik-Anweisungen eine schöne Figur zeichnest, dann kannst du sie morgen niemandem mehr zeigen, außer du gibst all die Anweisungen wieder ein. Damit das nicht nötig ist, gibt es die Möglichkeit, diese Anweisungen zu speichern. Das nennt man dann ein *Programm*.

Ich finde, dass es jetzt an der Zeit ist, dass du dein erstes Programm schreibst. Es wird zwar nur einige `print`-Anweisungen enthalten, aber das ist ja immerhin schon etwas!

Hier geht's darum zu lernen, was ein Programm ist, wie man es schreibt und wie man es ausführt.

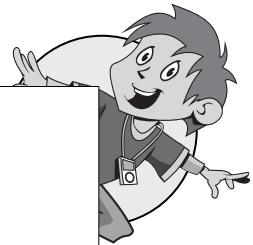
Um gleich von Anfang an etwas Ordnung in deine Programme zu bringen:



Lege ein Verzeichnis an, in das du deine Programme speichern willst. Vorschlag: Du erzeugst auf dem Laufwerk C: ein Verzeichnis mit dem Namen `py4kids`. (Wenn du die Vorschläge aus Anhang A befolgt hast, ist es wahrscheinlich schon da.) Dort legst du dann nach Bedarf Unterverzeichnisse `kap01`, `kap02` usw. an, in die deine Programme, die du zu den einzelnen Buch-Kapiteln schreiben wirst, hineinkommen.

Aufgabenstellung für dein erstes Programm: Schreibe ein Programm, das folgende Ausgabe erzeugt:

Dein erstes Programm



Hi Kleiner!
Wie viel ist eins und eins?
Ganz leicht!
 $1 + 1 = 2$

Wie geht man das an?

Einfache Python-Programme bestehen aus einer Folge von Python-Anweisungen, die als Programmtext in einer Datei gespeichert werden. Solche Programme werden auch oft als *Scripts* bezeichnet.

Mit der Entwicklungsumgebung IDLE erstellst du Python-Scripts mit folgenden Schritten.

- ⇒ **Schritt 1:** Öffne ein »Editor-Fenster«. Wähle dazu im SHELL-Fenster den Menüpunkt FILE|NEW WINDOW oder drücke die Tastenkombination **[Strg]+[N]**.

Ein ganz leeres Editor-Fenster öffnet sich. Es hat einen anderen Menü-Balken als das SHELL-Fenster und im Titelbalken steht: UNTITLED. Es dient der Eingabe und Bearbeitung von Programmtexten.



The screenshot shows the Python IDLE application window. On the left is the Python Shell window, which contains the following text:
36
>>> print(36 * 10)
360
>>>
3636
>>>
1 2
>>>
3 *
>>>
3 *
>>>
Synt
>>>
Uwe!
>>>
Uwe
>>>

On the right is the Untitled editor window, which has a menu bar with File, Edit, Format, Run, Options, Windows, Help. The title bar says "Untitled". The main area of the editor is empty.

FILE|NEW WINDOW öffnet ein leeres Editor-Fenster.

Beachte! Ab sofort verwenden wir zwei Arten von Fenstern: ein PYTHON SHELL-Fenster für den Direktmodus und ein Editor-Fenster für die Programme, die wir schreiben.



1



Das PYTHON SHELL-Fenster und die Editor-Fenster verhalten sich *ganz unterschiedlich*:

Die Shell versteht Python und wertet deine Ausdrücke, einen nach dem anderen, aus. Die IDLE hat immer nur *ein* Fenster mit einem interaktiven Python-Interpreter. In der Titelleiste des SHELL-Fensters steht PYTHON SHELL. Das erkennst du am Python-Prompt:

>>> |

Ein Editor-Fenster dient zum Schreiben von Programmen – es ist eigentlich nur ein kleines Textverarbeitungsprogramm. (Ein bisschen Python versteht auch ein Editor-Fenster. Das wirst du später sehen, wenn es dir hilft, Python-Scripts in der richtigen Form zu schreiben.) Du kannst in der IDLE gleichzeitig mehrere Editor-Fenster geöffnet haben. In der Titelleiste eines »neuen« Editor-Fensters steht UNTITLED. Sobald eine Datei abgespeichert oder neu geladen ist, steht dort der Dateiname. Du kannst Programme von ihrem Editor-Fenster aus ausführen. Die Ausgabe erscheint dann im SHELL-Fenster.

➤ **Schritt 2:** Schreibe ins Editor-Fenster die Programmanweisungen. Für unser Beispiel sind das folgende:

```
print("Hi Kleiner!")
print("Wie viel ist eins und eins?")
print("Ganz leicht!")
print("1 + 1 =", 1 + 1)
```

Achte darauf, dass der Text jeder Zeile *ganz links* beginnt! Leerzeichen vor einer einfachen Python-Anweisung sind hier *nicht erlaubt*.

In Kapitel 3 wirst du genauer erfahren: In Python haben Leerzeichen am Anfang von Programmzeilen eine besondere *Bedeutung*. Es wird daher durch die Syntaxregeln von Python festgelegt, wo und zu welchem Zweck Leerzeichen hingehören.

Falsche Leerzeichen führen zu Syntaxfehlermeldungen.

➤ **Schritt 3:** Speichere das Programm unter einem geeigneten Namen, z. B. hi.py, im Verzeichnis C:\py4kids\kap01 ab. Python-Programme müssen die Endung .py haben. Wähle dazu im Editor-Fenster das Menü FILE|SAVE AS...



Dein erstes Programm



Dateinamen sollten einen Bezug zum Inhalt des Programms haben, damit du auch später noch leicht erkennen kannst, was das Programm macht.

- Schritt 4: Führe das Programm in der IDLE aus: Wähle dazu im *Editor-Fenster* das Menü RUN|RUN MODULE oder drücke die Taste **F5**. (Diesen Menüpunkt gibt es nur im Editor-Fenster. Im PYTHON SHELL-Fenster ist er nicht zu finden.)

The screenshot shows the IDLE Editor window with the file 'hi.py' open. The code contains four print statements. A context menu is open over the code, with the 'Run Module F5' option highlighted. The status bar at the bottom right shows 'Ln: 5 Col: 0'.

Nach dem Sichern wird das Programm vom Editor-Fenster aus gestartet.

Solltest du vergessen haben, das Programm vor der Ausführung zu speichern, wirst du in einem SAVE BEFORE RUN-Dialogfenster gefragt, ob du das möchtest. In diesem Fall klicke auf den OK-Knopf. Von der IDLE aus kann nur ein nach der letzten Änderung gespeichertes Programm ausgeführt werden.

Die IDLE benutzt den Python-Interpreter, um das eingegebene Programm auszuführen. Das PYTHON SHELL-Fenster wird aktiviert und zeigt die Programm-Ausgabe an:

The screenshot shows the IDLE Python Shell window. It displays the code from 'hi.py' and its execution results. The shell shows the output of the print statements and then an error message for an invalid syntax. After a restart, it shows the program's output again. The status bar at the bottom right shows 'Ln: 34 Col: 4'.

Das Programm *hi.py* wurde ausgeführt.

1



Das ist die Ausgabe unseres Programms. Der Cursor blinkt neben einem neuen Eingabe-Prompt. Der Python-Interpreter, der eben noch dein Programm ausgeführt hat, kann nun wieder interaktiv verwendet werden.

Vor der Programmausführung wurde ein === RESTART === ausgeführt. Das hat für dich noch keine Bedeutung. Im Anhang F (Seite 442) kannst du mehr darüber erfahren.



Bevor wir die Arbeit mit diesem Programm abschließen, wollen wir noch in das Programm hineinschreiben, wer wann wo zu dieses Programm gemacht hat. Natürlich wird das kein Text für den Python-Interpreter sein – sondern Text für dich, wenn du später mal das Programm wieder anschaust. Solche »Kommentare« dienen auch für andere LeserInnen, etwa deine Freundin oder deinen Lehrer als Information.

In Python werden Kommentare durch das #-Zeichen markiert. Alles, was in einer Zeile hinter diesem Zeichen steht, wird vom Interpreter als Kommentar erkannt und ignoriert.

➤ Schritt 5: Schreibe an den Anfang deines Programms einen *Kopfkommentar*, bestehend aus drei Kommentarzeilen nach dem unten stehenden Muster:

```
hi.py - C:/py4kids/kap01/hi.py*
File Edit Format Run Options Windows Help
# Autor: Gregor Lingl
# Datum: 4. 8. 2009
# hi.py: Begrüßung und leichte Frage

print("Hi Kleiner!")
print("Wie viel ist eins und eins?")
print("Ganz leicht!")
print("1 + 1 =", 1 + 1)

I

Ln: 9 Col: 0
```

➤ Schritt 6: Speichere das Programm neuerlich ab! Jetzt reicht FILE|SAVE oder [Strg]+[S], weil das Programm schon einmal gespeichert wurde.



Beim Speichern wird die frühere Fassung deines Programms nun durch die neue überschrieben. Wenn du das Programm erneut ausführst, wird es dieselbe Ausgabe erzeugen. Am Programmablauf ändert sich durch das Einfügen von Kommentaren nichts!

Wir erweitern unser erstes Programm



Ein Blick auf die Titel-Leiste des Editor-Fensters zeigt dir stets, ob das Programm seit dem letzten Speichern geändert wurde: Dann findest du Sternchen vor und nach dem Titel. Diese Sternchen verschwinden beim Abspeichern. Du kannst das in den letzten beiden Abbildungen sehen.

- » Schritt 7: Schließe alle IDLE-Fenster, z.B. über den Menüpunkt FILE|EXIT oder mittels **Strg**+**Q**.

Wir erweitern unser erstes Programm

Oft wirst du bei der Arbeit mit diesem Buch vor der Aufgabe stehen, aus einem Programm, das du geschrieben hast, ein neues zu entwickeln. Im Folgenden zeige ich dir, wie man dabei vorgeht:

Aufgabenstellung: Wir wollen unser Programm `hi.py` so erweitern, dass es folgende Ausgabe erzeugt:

Hi, Kleiner!
Wie viel ist eins und eins?
Ganz leicht!
 $1 + 1 = 2$

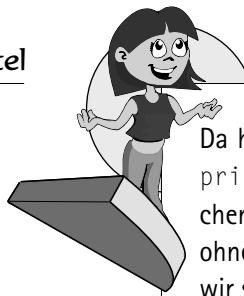
Und wie viel ist die Wurzel aus 4?
Nicht mehr ganz so leicht!
Die Wurzel aus 4 ist 2.0

Lösung: Alles, was man dazu braucht, haben wir weiter oben schon ausprobiert. Ausgenommen: Wie erzeugt man eine Leerzeile? Wir werfen wieder die IDLE an und probieren:

- » Mach mit!

```
>>> print("irgendwas")
irgendwas
>>> print("")
>>> print()
>>>
```

1



Da haben wir gleich zwei Möglichkeiten zur Auswahl: Entweder wir lassen `print` einen so genannten *Leerstring* ausgeben, also zwei Anführungszeichen ohne etwas dazwischen: `""`. Das ist sozusagen eine Zeichenkette ohne Zeichen. (Erinnert irgendwie an die leere Menge aus Mathe ...) Oder wir schreiben überhaupt nur `print`.

Apropos Mathe! Um die Quadratwurzel auszurechnen, brauchen wir die Funktionen aus `math`. Daher muss unser Programm diese auch importieren! Wie die `import`-Anweisung aussieht, weißt du schon von unserer interaktiven Sitzung. Wir könnten sie genau so in unser Programm übernehmen.

Mit

```
from math import *
```

werden aber alle Funktionen aus dem Modul `math` importiert. Da wir aber nur eine brauchen, importieren wir diesmal zielgerichtet auch nur die eine:

```
from math import sqrt
```

Beginnen wir also. Ich gehe davon aus, dass die IDLE gestartet ist, dass aber kein Editor-Fenster geöffnet ist.

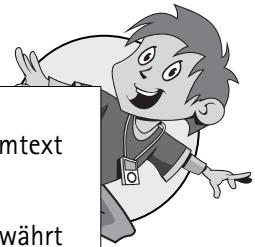
- ⇒ Öffne vom PYTHON SHELL-Fenster aus dein erstes Programm `hi.py`. (Menü FILE|OPEN... oder bequemer den Menüpunkt FILE|RECENT FILES|`C:\py4kids\kap01\hi.py`)
- ⇒ Wir wollen dem neuen Programm den Namen `himath.py` geben, um uns daran zu erinnern, dass es das Modul `math` benutzt. Also ändere im Kopfkommentar den Programmnamen auf `himath.py` ab, ändere falls nötig das Datum und aktualisiere auch die Beschreibung in der Zeile mit dem Programmnamen. Dann speichere das Ganze über den Menüpunkt FILE|SAVE AS unter dem neuen Namen ab.
- ⇒ Unter den Kopfkommentar schreibe als erste Anweisung:

```
from math import sqrt
```

Die `import`-Anweisungen stehen immer am Anfang eines Scripts.

Nach den bereits vorhandenen vier `print`-Anweisungen sind jetzt noch weitere vier `print`-Anweisungen anzufügen. Die erste soll eine Leerzeile erzeugen. Die nächsten beiden sollen bestimmten Text ausgeben, dazu verwenden wir Strings. Welche das sind, kannst du der Aufgabenstellung zu dieser Übung sofort entnehmen. Die letzte `print`-Anweisung muss einen String und das Ergebnis der Wurzelberechnung ausgeben.

Wir erweitern unser erstes Programm



- ⇒ Schritt 1: Füge die vier print-Anweisungen an den Programmtex an.
- ⇒ Schritt 2: Sichere das erweiterte Programm. Achtung, hier bewährt sich wieder die Tastenkombination **[Strg]+[S]** im Editor-Fenster.
- ⇒ Schritt 3: Führe das Programm mit **[F5]** (bei aktivem *Editor-Fenster*) aus.

Hat alles geklappt und ist im PYTHON SHELL-Fenster die Programmausgabe erschienen?

The screenshot shows two windows side-by-side. The left window is a text editor titled "himath.py - C:\py4kids\kap01\himath.py" containing Python code. The right window is a "Python Shell" window showing the program's output.

himath.py Content:

```
# Autor: Gregor Lingl
# Datum: 4. 8. 2009
# himath.py: Begrüßung, leichte Frage
#             und eine schwerere (Wurzel!)

from math import sqrt

print("Hi Kleiner!")
print("Wie viel ist eins und eins?")
print("Ganz leicht!")
print("1 + 1 =", 1 + 1)

print()
print("Und wie viel ist die Wurzel aus 4?")
print("Nicht mehr ganz so leicht!")
print("Die Wurzel aus 4 ist", sqrt(4))
```

Python Shell Output:

```
Type "copyright", "credits" or "license()" for more information.
>>> print("irgendwas")
irgendwas
>>> print("")
>>> print()

>>> ===== RESTART =====
>>>
Hi Kleiner!
Wie viel ist eins und eins?
Ganz leicht!
1 + 1 = 2

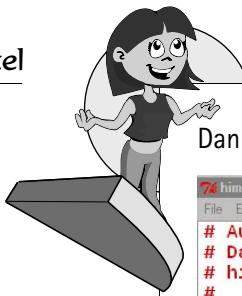
Und wie viel ist die Wurzel aus 4?
Nicht mehr ganz so leicht!
Die Wurzel aus 4 ist 2.0
>>> |
```

Das Programm *himath.py* und seine Ausgabe.

Sollte sich dagegen ein Fehler eingeschlichen haben, wird im SHELL-Fenster eine Fehlermeldung erscheinen. Ich führe dir das an einem Beispiel vor. Angenommen, du hättest geschrieben:

```
print "1 + 1 =", 1 + 1
Print ""
print "Und wie viel ist die Wurzel aus 4?"
```

1



Dann hätte der Versuch, das Programm auszuführen, zu Folgendem geführt:

```

himath.py - C:/py4kids/kap01/himath.py
File Edit Format Run Options Windows Help
# Autor: Gregor Lingl
# Datum: 4. 8. 2009
# himath.py: Begrüßung, leichte Frage
#           und eine schwerere (Wurzel!)

from math import sqrt

print("Hi Kleiner!")
print("Wie viel ist eins und eins?")
print("Ganz leicht!")
print("1 + 1 =", 1 + 1)

Print()
print("Und wie viel ist die Wurzel aus 4?")
print("N")
print("D")
print("Wie viel ist eins und eins?
Ganz leicht!
1 + 1 = 2

Und wie viel ist die Wurzel aus 4 ?
Nicht mehr ganz so leicht!
Die Wurzel aus 4 ist 2.0
>>> ===== RESTART =====
>>>
Hi Kleiner!
Wie viel ist eins und eins?
Ganz leicht!
1 + 1 = 2
Traceback (most recent call last):
  File "C:/py4kids/kap01/himath.py", line 13, in <module>
    Print()
NameError: name 'Print' is not defined
>>>

```

Python ist case-sensitive. Das heißt, dass es Groß- und Kleinschreibung unterscheidet. Das Wort `Print` ist zu unterscheiden vom Wort `print`.

Ein Namenfehler? Ja, jetzt fällt dir auf, dass in der fünften print-Anweisung `Print` großgeschrieben steht. Und das ist für Python ein anderes Wort als das kleingeschriebene `print`. Du hättest das auch daran erkennen können, dass dieses `Print` nicht violett eingefärbt war wie die anderen.

Python unterscheidet Groß- und Kleinschreibung! Das hat es mit anderen wichtigen Programmiersprachen wie C, C++ und Java und mit vielen weiteren gemeinsam!

Wörter, die sich in der Groß-/Kleinschreibung unterscheiden, sind für Python **verschiedene Wörter!**

Die nächsten Schritte: Fehler ausbessern und nochmals laufen lassen, bis keine Fehlermeldungen mehr auftreten.





Syntax-Colouring: bunte Farben für den besseren Durchblick

Bei der Arbeit mit der IDLE wird dir aufgefallen sein, dass der Text in unterschiedlichen Farben erscheint. Das Wort `print` ist violett, das Wort `import` orange, Strings sind grün und Kommentare sind rot.

Das liegt daran, dass die IDLE die Python-Syntax kennt. Sie weiß, dass Strings in Python-Programmen eine besondere Rolle spielen, sie kennt die so genannten *reservierten Wörter* von Python und einiges mehr.

Auf diese Weise kann die IDLE die Struktur deiner Programme mit Farben verdeutlichen.

Diese Einfärbung der Wörter gemäß der Syntax von Python nennt man *Syntax-Colouring*. Leider können die Bilder in diesem Buch dies nur unvollkommen durch unterschiedliches Grau darstellen.

Gewöhne dir an, darauf zu achten. Denn dann kann dir so ein Fehler wie Großschreibung eines reservierten Wortes nicht passieren. Du hast gesehen, ein `Print` färbt die IDLE nicht violett ein. Da muss also was faul sein.

Reservierte Worte werden für grundlegende Bestandteile der Sprache Python verwendet. Dazu gehört die `import`-Anweisung. Beachte, dass auch das Wort `from` ein reserviertes Wort ist.

Es gibt in Python insgesamt 33 *reservierte Wörter*. Sie werden für grundlegende Bestandteile der Sprache Python verwendet.

Sie dürfen für keinerlei andere Zwecke verwendet werden als für die, für die sie in Python vorgesehen sind.

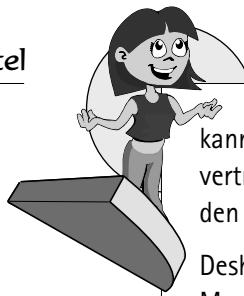
Bis jetzt kennst du erst zwei reservierte Wörter: `from` und `import`.



Mit Mustern arbeiten

Programmierer arbeiten sehr viel mit *Mustern*. Manche Muster betreffen ganze Programme oder große Teile von Programmen, andere Muster betreffen kleine Bestandteile.

Das funktioniert im Wesentlichen so: Du stehst vor einer Aufgabe und suchst in deinem Kopf nach einem dir bekannten Denk-, Programm- oder Anweisungsmuster, das für die Lösung der Aufgabe angewendet werden



kann. Jedenfalls ist es immer leichter, wenn du eine Aufgabe mit einem vertrauten Muster lösen kannst, als wenn du einen neuen Lösungsweg finden musst.

Deshalb werde ich dir für immer wieder vorkommende Problemstellungen Muster formulieren. Versuche, sie in deinem Kopf abrufbar zu verankern.

Sicher wirst du bei der Arbeit mit diesem Buch noch einige einfache Python-Scripts schreiben, daher gebe ich dir dafür gleich ein einfaches Muster:

Muster 1: Einfaches Python-Script

```
# Autor: Wer
# Datum: Wann
# Dateiname: Was
```



Kopfkommentar

```
from modul import *
```



... alle oder einzelne Namen.
Wird nicht in jedem Programm gebraucht.

```
Anweisung 1
Anweisung 2
```

...

(Leere Zeilen können nach Belieben eingeschoben werden, um das Programm leichter lesbar zu machen.)

Die in dieser Musterbeschreibung *kursiv* geschriebenen Wörter werden durch konkrete Informationen ersetzt. Im obigen Muster z.B. *Wer*. Ich schreibe dafür Gregor Lingl hinein. Du eben deinen Namen.

Oder *modul*: Da kommt es drauf an, welches Script du schreibst. Willst du etwas Kompliziertes rechnen, musst du dort *math* einsetzen. Willst du etwas zeichnen, dann brauchst du auch *dafür* ein anderes passendes Modul. Manche Programme – wie unser *hi.py* – müssen gar keine Namen aus Modulen importieren.

Zusammenfassung

- ❖ Um Python-Programme auszuführen, brauchst du einen Übersetzer von der höheren Programmiersprache Python in die Maschinensprache deines Computers: den Python-Interpreter.
- ❖ Die Rechtschreibregeln einer Programmiersprache nennt man auch ihre Syntax.

Zusammenfassung

- ❖ Python-Anweisungen müssen die Syntax der Sprache Python einhalten.
- ❖ Um mit Python zu arbeiten, verwendest du das Entwicklungswerkzeug IDLE.
- ❖ Die IDLE hat im SHELL-Fenster einen interaktiven Python-Interpreter. Er führt direkt und sofort einzelne Python-Anweisungen aus und schreibt Ergebnisse an.
- ❖ Die IDLE hat auch Editor-Fenster. Das sind Fenster zum Verfassen und Bearbeiten von Programmtexten.
- ❖ In diesem Kapitel hast du zwei Python-Anweisungen kennen gelernt:
 - Die `from import`-Anweisung
 - Die `print`-Anweisung
- ❖ Python kann rechnen und verwendet dafür die uns bekannte mathematische Schreibweise. Doch beachte: Kommazahlen werden nicht mit einem Komma, sondern mit einem Punkt geschrieben.
- ❖ Für höhere Rechnungen muss Python Funktionen aus dem Modul `math` importieren.
- ❖ Beim Rechnen mit Kommazahlen macht Python Rundungsfehler.
- ❖ Python kann auch Texte (Strings genannt) schreiben.
- ❖ Python-Programme sind Dateien, die eine Folge von Python-Anweisungen enthalten.
- ❖ Python-Programme können durch `#` gekennzeichnete Kommentarzeilen enthalten.
- ❖ Am Kopf jedes Programms sollte ein Kommentar mit Angaben zu Autor, Erstellungsdatum und Programmzweck stehen.
- ❖ Python-Programme können innerhalb der IDLE ausgeführt werden. Dazu wird im Editor-Fenster das Menü RUN|RUN MODULE oder die Taste `F5` benutzt.
- ❖ Die IDLE kennt die Python-Syntax und kennzeichnet verschiedene Bestandteile eines Programms durch Farben.





Zum Abschluss noch ein paar Übungsaufgaben ...

Vielleicht hast du es übertrieben gefunden, den Python-Interpreter ausrechnen zu lassen, wie viel $1 + 1$ ist. Vielleicht dachtest du gar, man hätte als letzte Anweisung in `hi.py` schreiben können:

```
print("1 + 1 = 2")
```

Das hätte wohl dieselbe Ausgabe ergeben. Und trotzdem ist es gut, dass du Python rechnen lässt, wenn Python schon rechnen kann:

Aufgabe 1: Ändere `himath.py` so zu `himath2.py` ab, dass es die Quadratwurzel von 152399025 ausgibt.

Aufgabe 2: Ändere `himath2.py` weiter so zu `himath3.py` ab, dass das Programm im ersten Teil nicht nach $1 + 1$, sondern nach $123456789 * 987654321$ fragt und dieses Produkt dann auch ausgibt.

Wenn dir die Lösung von Aufgabe 2 gelungen ist, dann wirst du merken, dass Python im Rechnen höchstwahrscheinlich besser ist als dein Taschenrechner!

Aufgabe 3: Python kann auch Potenzen berechnen. Zum Beispiel zwei hoch zehn, oder 2^{10} . Der Potenz-Operator in Python ist `**`. Also: 2^{**10} . Experimentiere im SHELL-Fenster mit 2er-Potenzen. Wie groß sind die Zahlen 2^8 , 2^{10} , 2^{20} , 2^{24} ? Sind dir diese Zahlen im Zusammenhang mit Computern schon untergekommen? Kannst du mit Python auch höhere Potenzen von 2 berechnen? Was ist deiner Meinung nach größer: 2^{1000} oder 1000^2 ?

Aufgabe 4: Python kennt auch Strings, die mit drei aufeinanderfolgenden Anführungszeichen geöffnet und abgeschlossen werden. Beispiel:

➤ Mach mit:

```
>>> print("""Ene  
mene  
muh!  
und...?""")
```

Tatsächlich! Eine `print`-Anweisung, die vier Zeilen ausgibt (und es wären auch noch mehr drin!).

Entwickle aus `himath3.py` ein Programm `himath4.py`, das die gleiche Ausgabe erzeugt, aber mit zwei `print`-Anweisungen auskommt.



... und ein paar Fragen

1. Was ist der Unterschied zwischen `print("1+2")` und `print(1+2)`?
2. Mit welcher Funktion berechnet man in Python Quadratwurzeln?
3. Was ist der Unterschied zwischen dem PYTHON SHELL-Fenster und einem Editor-Fenster?
4. Rätsel: Wie erzielst du folgende Bildschirmausgabe:

Sie sagte: "No, don't do it."

(Es gibt mehrere richtige Antworten.)

5. Beunruhigt dich dies:

```
>>> 0.4  
0.4  
>>> 0.2  
0.2  
>>> 0.4 + 0.2  
0.6000000000000001
```

6. Was erwartest du von:

```
>>> print(0.4 + 0.2)
```

Zu einigen Buchkapiteln gibt es auf der Buch-CD zusätzliches Material. Sieh dir dazu auf dieser CD mit einem Webbrowser die Datei `index.html` an.

Weitere und/oder neuere Informationen gibt es vielleicht auf der Webseite <http://python4kids.net>.



2

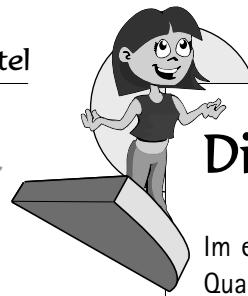


Was Schildkröten mit Grafik zu tun haben: Turtle-Grafik

Bevor dir nun womöglich langweilig wird wegen der vielen Erklärungen über Zeichen und Zahlen, wollen wir lieber ein paar Grafiken zeichnen.

In diesem Kapitel lernst du ...

- ◎ ein Software-Tierchen namens Turtle kennen. Es wohnt im Turtle-Grafik-Modul `turtle.py`.
- ◎ einige Anweisungen kennen, die die Turtle zum Zeichnen bringen.
- ◎ wie du dir einen Plan machst, um eine Grafik nach deinen Vorstellungen zu erzeugen...
- ◎ ... und diesen dann in ein Programm umsetzt.



Die Turtle und der IPI

Im ersten Kapitel hast du erfahren, dass selbst so geläufige Dinge wie die Quadratwurzel in einem Modul untergebracht sind, das nicht zum Kern der Sprache Python gehört. Es wird dich daher wohl kaum wundern, dass es sich mit der Grafik ganz genau so verhält.

In der Tat gibt es verschiedene Grafik-Module, mit denen man in Python arbeiten kann.

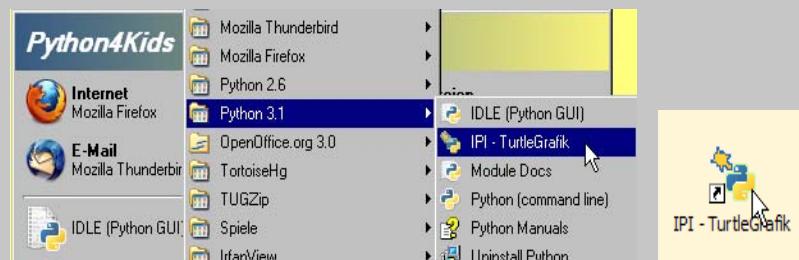
Zum Programmierenlernen finde ich die so genannte Turtle-Grafik besonders geeignet. Deshalb habe ich – ursprünglich für die Arbeit mit diesem Buch – ein Turtle-Grafik-Modul geschrieben: es heißt `turtle.py`. Seit Python 2.6 ist dieses Modul in die Standardbibliothek von Python aufgenommen worden. Das heißt, wenn du Python 3.1 installiert hast, ist es schon mit dabei.

Interaktive Grafik und die IDLE

Damit du mit dem *interaktiven Python-Interpreter* Grafik-Befehle ausführen kannst, braucht die IDLE unbedingt eine ganz bestimmte Einstellung.

Mehr über diese Einstellung findest du in Anhang A, ab Seite 427.

Der Menüpunkt START|ALLE PROGRAMME|PYTHON2.5|IPI-TURTLEGRAFIK startet die IDLE mit dieser Einstellung. Alternativ kannst du dazu auch das Icon IPI-TURTLEGRAFIK doppelt anklicken.



Für Turtle-Grafik-Programme: Achte darauf, den IPI-TURTLEGRAFIK zu verwenden!

IPI? Ach ja – das ist meine Abkürzung für »Interaktiver Python-Interpreter« – ich werde sie ab nun im ganzen Buch für die Form der IDLE Python Shell verwenden, die besonders für die Turtle-Grafik-Arbeit eingestellt ist. Fast alles, was du in Kapitel 1 über die IDLE(PYTHON GUI) gelernt hast, gilt auch für den IPI für die Turtle-Grafik.

Die Turtle und der IPI



Nun zur Turtle selbst. Klingt für dich echt spanisch? Na ja – »turtle« ist doch wieder ein englisches Wort und heißt auf Deutsch *Schildkröte*. Diese Schildkröte ist es, die auf einer Zeichenfläche Grafiken erzeugt. Sie ist ein kleines Software-Tierchen und wird im Zeichenfenster grafisch dargestellt. Der Kopf zeigt in ihre Bewegungsrichtung.

Bevor ich dir lange Erklärungen gebe, verwenden wir den IPI, um zu sehen, wie das funktioniert. Starte also den IPI-TURTLEGRAFIK. Es geht ein PYTHON SHELL-Fenster auf, das etwas anders aussieht als IDLE(PYTHON GUI):

A screenshot of a Windows-style application window titled "IPI Shell / Turtle Grafik". The menu bar includes File, Edit, Debug, Options, Windows, and Help. The main window displays the Python 3.1.1 command prompt:

```
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>> |
```

The status bar at the bottom right shows "Ln: 4 Col: 4".

So sieht die Shell aus, die für die Turtle-Grafik gebraucht wird: In der Titelleiste steht IPI SHELL / TURTLEGRAFIK und in der Zeile über dem ersten Prompt steht
==== No Subprocess ====.

➤ IPI-Session! Mach mit!

```
>>> from turtle import *
```

Wir importieren alle Funktionen aus dem Turtle-Grafik-Modul `turtle`. (Obwohl wir anfangs nur einige wenige davon verwenden werden.)

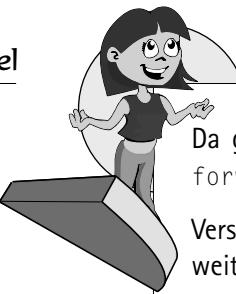
```
>>> forward(100)
```

A screenshot of the "IPI Shell / Turtle Grafik" window. The command prompt shows:

```
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>> from turtle import *
>>> forward(100)
>>>
```

Below the shell window is a smaller window titled "Python Turtle Grafik" which displays a yellow star drawn by the turtle.

Ins IPI SHELL / TURTLEGRAFIK-Fenster schreibst du die Turtle-Grafik-Befehle. Im Grafik-Fenster zeichnet die Turtle.



2

Da geht nun ein neues Fenster auf und unsere Schildkröte ist ein Stück forward, vorwärts, gewandert.

Verschiebe, falls nötig, die beiden Fenster auf dem Bildschirm so, dass du weitere Eingaben im IPI-Fenster machen kannst.

Vielleicht ist dir sogar aufgefallen, dass beim Eintippen von `forward()` ein Tipp erschienen ist: In der ersten Zeile steht, was da jetzt weiter einzugeben ist, nämlich eine Distanz. In der zweiten Zeile steht, was die Funktion `forward()` bewirkt.

```

IPI Shell / Turtle Grafik
File Edit Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32
bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>> from turtle import *
>>> forward(100)
      (distance)
      Bewege die Turtle um distance nach vorne.
  
```

»Call-Tipps« helfen dir zu verstehen, wie Turtle-Grafik-Funktionen verwendet werden.

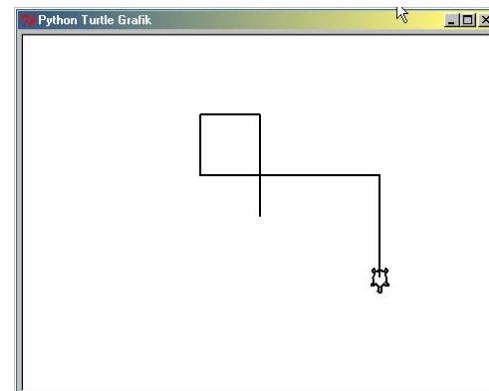
100 gibt also die Distanz an, die die Turtle weitergeht. Und dabei zeichnet sie eine Strecke.

Experimentieren wir noch ein wenig weiter:

⇒ Mach mit:

```

>>> left(90)
>>> forward(60)
>>> left(90)
>>> forward(60)
>>> left(90)
>>> forward(180)
>>> right(90)
>>> forward(100)
>>>
  
```



Hat deine Turtle dasselbe gezeichnet wie meine? Wenn nicht, hast du vielleicht einen Zahlenwert falsch eingetippt? Einmal versehentlich `left` mit `right` verwechselt? Oder gefällt dir einfach nur die Zeichnung nicht?

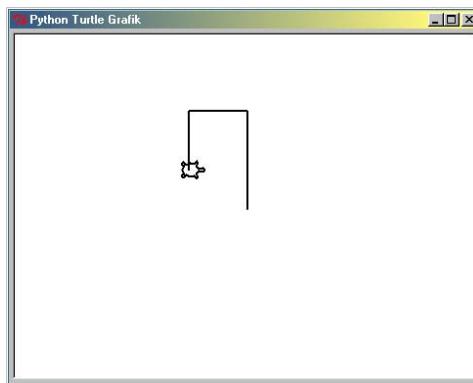
⇒ Kein Problem, mach mit!

```

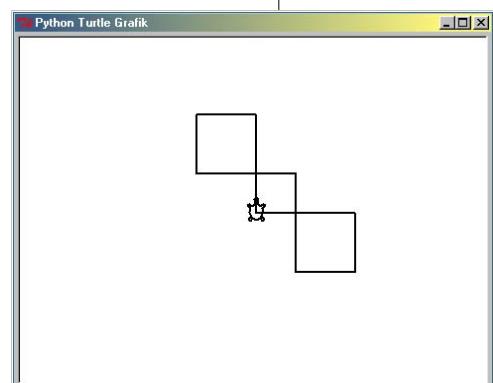
>>> undo()
>>> undo()
>>> undo()
>>>
  
```

Die Turtle und der IPI

Die `undo()`-Anweisung kann `forward()`-, `left()`-, `right()`- und noch andere `turtle`-Anweisungen rückgängig machen. Das ist sehr praktisch für das Arbeiten im Direkt-Modus, weil man da doch öfter mal Fehler macht. Du kommst dann mit ein paar `undo()`-Anweisungen an einen früheren Punkt zurück, wo noch alles gestimmt hat. Damit ersparst du dir, wegen eines Fehlers die ganze Zeichnung von vorne beginnen zu müssen.



Versuche nun beispielsweise nebenstehende Grafik zu erzeugen. Oder experimentiere mit zwei, drei, vielen weiteren `forward()`-, `left()`- und `right()`-Anweisungen. Und wenn dir einmal etwas nicht gelingt: `undo()`. Vielleicht kommt ein interessantes grafisches Muster heraus. Vielleicht aber auch nur chaotisches Gekritzeln. Macht nichts, du erfährst dabei, wie diese Anweisungen arbeiten.



Die Anweisung `forward(100)` ist ein Aufruf der Funktion `forward` aus dem Modul `turtle`. Um ihre Arbeit ausführen zu können, braucht sie eine Zahlenangabe, um wie viele Einheiten die Turtle vorwärts laufen soll. Diese Information muss ihr beim Aufruf als Argument übergeben werden.

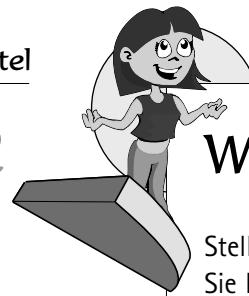
Wie die meisten Funktionen aus dem Modul `turtle` hat auch `forward` keinen Rückgabewert. Ihre Aufgabe ist es, eine Aktion auszuführen: eine Linie zu zeichnen.



Bei deinen ersten Experimenten hast du gesehen, dass die Turtle nicht nur vorwärts gehen kann, sie kann sich auch drehen, mit Anweisungen wie `left(90)` und `right(60)`. Warst du so mutig und hast auch andere Winkel als 90° verwendet? Du kannst der Turtle beim Drehen direkt zusehen:

```
>>> left(450)
```

Ganz nett. Für die Zeichnung hat aber `left(90)` den gleichen Effekt!



2

Wie macht die Turtle das?

Stelle dir die Schildkröte als Tierchen vor, das nur geradeaus gehen kann. Sie hat einen Messstock dabei, mit dem sie die Strecke abmessen kann. Sie hat aber auch noch einen Winkelmaßstab dabei. Wenn du ihr nun die Anweisung gibst, sich um einen bestimmten Winkel zu drehen, kann sie diesen Winkel abmessen und deiner Anweisung Folge leisten. Sie schaut dann in eine neue Bewegungsrichtung. Außerdem hat sie einen Stift, mit dem sie auf der Zeichenfläche eine Spur entlang ihres Weges zieht.

Wenn du ihr also die Anweisung `forward(100)` erteilst, dann geht sie um 100 Einheiten (so genannte *Pixel*) nach vorne, in die Richtung, in die sie gerade schaut. Erteilst du ihr die Anweisung `right(90)`, dann dreht sie sich um 90° nach rechts. Mit `forward(50)` zeichnet sie dann einen weiteren, diesmal kürzeren, Strich in die neue Richtung.

Hilfe!

Du wirst in den folgenden Kapiteln eine Reihe verschiedener Turtle-Grafik-Funktionen verwenden. Manchmal wirst du dich vielleicht nicht daran erinnern, wozu eine bestimmte Funktion gut ist oder wie man sie richtig verwendet. Python bietet dir dafür seine Hilfefunktion `help()` an. Mit ihr kannst du Hilfe zu Turtle-Grafik-Funktionen abrufen. Sehen wir uns das für die Funktion `left()` an:

```
>>> help(left)
```

Mit der `help()`-Funktion des IPI kannst du Hilfe zu allen Turtle-Grafik-Funktionen herbeiholen.

```
IPI Shell / Turtle Grafik
File Edit Debug Options Windows Help
>>> help(left)
Help on function left in module turtle:
left(angle)
    Drehe die Turtle um angle Winkeleinheiten nach links.

    Alias-Namen: left | lt

    Argument:
    angle -- eine Zahl (ganze Zahl oder Gleitkommazahl)

    Drehe die Turtle um angle Winkeleinheiten nach links.
    (Voreingestellte Winkeleinheit ist Grad; das kann mit den Funktionen degrees() und radians() geändert werden.)

    Beispiel:

    >>> heading()
    22.0
    >>> left(45)
    >>> heading()
    67.0

    >>> |
Lr: 67 | Col: 4
```

Ein rotes Quadrat

Hier steht zunächst eine kurze Beschreibung, was die Funktion tut. In der zweiten Zeile stehen manchmal *Alias-Namen*. Das sind Funktionsnamen, die du alternativ verwenden kannst. Hier z. B. `lt` für `left`. (Diesen Alias-Namen gibt es, damit du weniger tippen musst.) Danach folgt eine Angabe der. Hier wird eine Zahl als Argument verlangt, die du beim Aufruf von `left()` zwischen die runden Klammern einsetzen musst. Weiter folgen:

- ❖ Eine ausführlichere Erklärung der Arbeitsweise der Funktion.
- ❖ Beispiel(e) wie die Funktion verwendet werden kann, die du auch im IPI eingeben und ausprobieren kannst.
- Sieh mal nach, ob es für `forward()` und `right()` auch Alias-Namen gibt.

Ein rotes Quadrat

➤ Mach weiter mit!

```
>>> reset()
```

Keine Angst, diese Anweisung startet den Computer nicht neu! Sie löscht nur die Zeichnungen im Turtle-Grafik-Fenster und setzt die Turtle auf ihren Anfangszustand zurück! Wir wollen jetzt ein Quadrat zeichnen und zwar farbig und mit dickem Zeichenstift.

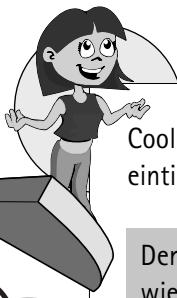
```
>>> pensize(5)  
>>> pencolor("red")
```

Da habe ich gleich zwei neue Turtle-Grafik-Funktionen verwendet. `pensize()` stellt die Strichdicke ein, hier auf 5 Pixel. `pencolor()` stellt die Farbe des Zeichenstifts der Turtle ein. Beachte die Anführungszeichen: "red" ist eine Zeichenkette. Für die Turtle ist es die Bezeichnung für die Farbe Rot. Dass die Turtle Englisch spricht, hast du ja sicherlich schon bemerkt. Als zukünftige Programmiererin muss man sich an so was gewöhnen. Die Turtle versteht auch "blue", "green", "yellow" usw. Eine Liste der teilweise sehr fantasievollen Farbnamen, die die Turtle versteht, findest du im Anhang D über *Farben*.

Tatsächlich siehst du diese Einstellungen der Turtle gleich an, weil sie selbst mit den gewählten Werten für Strichdicke und Stiftfarbe dargestellt wird. Sie ist also etwas größer und rot geworden. Jetzt wollen wir sehen, ob sie rot und dick zeichnet:

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





Was Schildkröten mit Grafik zu tun haben: Turtle-Grafik

Cool! Um das Quadrat fertig zu zeichnen, müsstest du das noch drei Mal eintippen. Zum Glück bietet uns der IPI da eine Abkürzung an:

Der IPI merkt sich alle Eingaben: Frühere Anweisungen kannst du durch wiederholtes Drücken der Tastenkombination **[Alt]+[P]** in die Eingabezeile zurückrufen und durch Drücken der **[←]-Taste** ausführen.

- Halte die **[Alt]-Taste** gedrückt und drücke die Taste **[P]**. Nun steht in der Eingabezeile die letzte Eingabe: `left(90)`. Die brauchen wir jetzt nicht. Drücke nun bei weiterhin gedrückter **[Alt]-Taste** nochmals die Taste **[P]**. Nun steht in der Eingabezeile die vorletzte Eingabe, und die war `forward(100)`. Drücke **[←]**. Die Turtle zeichnet die nächste Quadratseite.
- Dieselbe Tastenkombination nochmals ausgeführt, bewirkt nun die Ausführung von `left(90)`.
- Mache das noch dreimal und dein Quadrat ist fertig.

Aufgabenstellung: Schreibe nun ein *Programm*, das dieses rote, dicke Quadrat zeichnet.

Okay, ich sehe ein, es ist nicht so leicht, bei so viel Neuem den Überblick zu bewahren. Ich werde nochmals beschreiben, wie dieses Programm aufgebaut sein muss:

Programmentwurf: rotes Quadrat

Kopfkommentar

Aus dem Modul turtle alle Funktionen importieren

Strichdicke 5 Pixel einstellen

Stiftfarbe Rot einstellen

100 Einheiten vorwärts gehen

um 90 Grad nach links drehen

100 Einheiten vorwärts gehen

um 90 Grad nach links drehen

100 Einheiten vorwärts gehen

um 90 Grad nach links drehen

100 Einheiten vorwärts gehen

Wieder in die ursprüngliche Richtung schauen!

Um 90 Grad nach links drehen

Ein rotes Quadrat

Ich habe hier das Programm nicht in Python aufgeschrieben, sondern mit deutschen Wörtern, aber doch in der Form eines Programms. Jede Zeile entspricht einer Python-Anweisung. (Bis auf die vorletzte: Die entspricht einem Python-Kommentar.)

Warum die Anweisung in der letzten Zeile, wo doch das Quadrat schon fertig gezeichnet ist?

Mit dieser Anweisung wird die Turtle in die Richtung gedreht, die sie vor dem Zeichnen des Quadrats hatte. In unserem Fall schaut sie wieder nach oben. Wenn ich jetzt mit der Turtle noch etwas zeichnen will, brauche ich mich nicht darum zu kümmern, was sie vorher gezeichnet hat.

Das sollten wir für die Zukunft vereinbaren: Zeichnest du mit der Turtle eine Figur, dann stelle sie am Ende in dieselbe Richtung, in die sie vorher geschaut hat!

Gut. Schreibe jetzt das Programm `quadrat.py`:

- » Öffne zunächst vom IPI-Grafik-Shell-Fenster aus ein Editor-Fenster über das Menü FILE|NEW WINDOW.
- » Schreibe zuerst den Kopfkommentar hinein und speichere als `quadrat.py` im Ordner `c:\py4kids\kap02`.
- » Übersetze die einzelnen Zeilen des obigen Programmentwurfs in Python-Code. (In deinem IPI-SHELL-Fenster sollten noch die notwendigen Anweisungen stehen, dort kannst du falls nötig nachsehen!)

Fertig? Dann das Übliche:

- » Programm abspeichern: [Strg]+[S]
- » Programm ausführen: [F5]. Falls nötig, Fehler ausbessern und zum vorigen Schritt zurück!

Ist das Programm nun korrekt abgelaufen? Vielleicht ist dir aufgefallen, dass es im IPI beim Programmstart kein === RESTART === gibt. (Das ist einer der Unterschiede zur IDLE(PYTHON-GUI). Du brauchst ihn hier noch nicht weiter zu beachten.

Sollte bei dir bei der Programmausführung doch === RESTART === erscheinen, dann bist du auf dem falschen Dampfer! Wahrscheinlich sind bis hierher auch einige sonderbare Dinge aufgetreten. Du solltest unbedingt sofort Anhang F ab Seite 442 lesen!





Vielleicht ist dir weiter aufgefallen, dass bei mehrmaligem Programmablauf von `quadrat.py` das rote Quadrat jeweils über das vorhergehende gezeichnet wird. (Außer du hast das Grafik-Fenster vorher geschlossen.) Das ist nicht besonders praktisch, lässt sich aber leicht beheben:

- Füge in `quadrat.py` gleich nach der `import`-Anweisung als nächste Anweisung `reset()` ein.
- Speichere `quadrat.py` und führe es neuerlich (mehrmals) aus. Okay?

Füllen – und auf die Spitze!

Das heißt nicht, dass wir jetzt irgend etwas auf die Spitze treiben wollen. Wir wollen nur ganz einfach das Programm `quadrat.py` so weiterentwickeln, dass das Quadrat (1) mit einer Farbe gefüllt wird und (2) auf der Spitze steht.



Wir werden in diesem Buch häufig so vorgehen, dass wir aus einer einfachen Fassung eines Programms – wie hier `quadrat.py` – schrittweise eine Folge von Programmen entwickeln. Dazu ist es sinnvoll, mit einer Arbeitsversion des Programms, etwa `quadrat_arbeit.py` zu arbeiten. Immer, wenn ein funktionierendes Programm zu einem Teilziel erreicht ist, speichern wir eine Kopie davon unter einem eindeutigen neuen Namen mit Hilfe von FILE|SAVE COPY AS... ab.

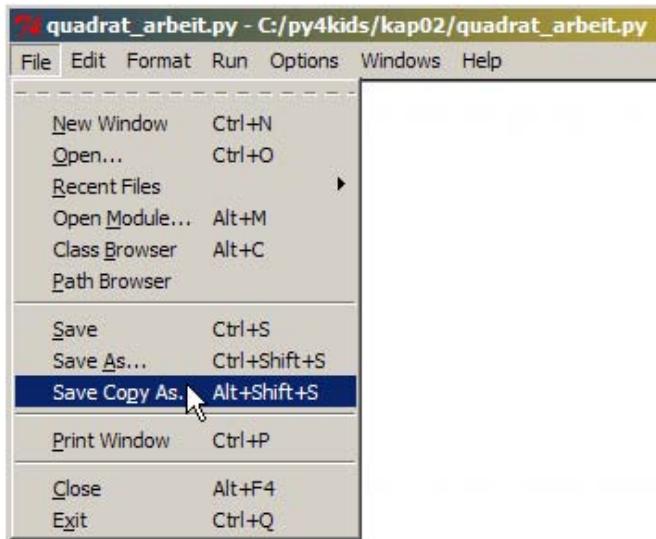
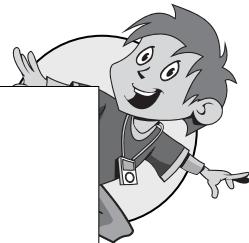
Sieh dir das am folgenden Beispiel mit `quadrat.py` genau an!

- Speichere `quadrat.py` mittels FILE|SAVE As (»Speichern unter«) unter dem neuen Namen `quadrat_arbeit.py`. Beachte, dass nun dieser neue Name in der Titelleiste des Editor-Fensters erscheint.

Nur dieses Programm `quadrat_arbeit.py` wollen wir in der Folge verändern! Die Fassung, die wir gerade erarbeitet haben, wollen wir als `quadrat01.py` sichern.

- Ändere im Kopfkommentar den Programmnamen auf `quadrat01.py` ab.
- Speichere eine Kopie des aktuellen `quadrat_arbeit.py` unter dem Namen `quadrat01.py` ab. Nutze dazu den Menüpunkt FILE|SAVE COPY AS... (»Eine Kopie Speichern unter«)

Füllen – und auf die Spitze!



Mit FILE|SAVE COPY AS... speicherst du eine Kopie des aktuellen Programms unter einem neuen Namen ab. Im Editor bleibt das Programm unter dem alten Namen.

In der Titelleiste des Editor-Fensters muss jetzt immer noch `quadrat_arbeit.py` stehen. Das ist die Arbeitskopie für unsere Quadrat-Programmierung.

Nun zur neuen Aufgabe! Zwei Schritte.

Schritt (1): Das Quadrat soll mit Farbe gefüllt werden. Eine nette Farbe ist "cyan". Wir probieren wieder im IPI. (Ich zeige dir dabei gleich einen schmutzigen Trick, den ich in Programmen nicht so gerne sehe – nicht weitersagen.)

➤ Mach mit!

```
>>> reset()  
>>> pensize(5)  
>>> pencolor("red")
```

Jetzt stellen wir auch noch eine Füllfarbe ein:

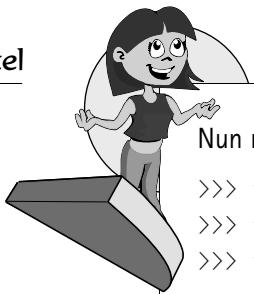
```
>>> fillcolor("cyan")
```

Oh! Sieh dir mal die Turtle an! Und nun sagen wir an, dass die Zeichnung, mit der wir jetzt beginnen wollen, gefüllt werden soll:

```
>>> begin_fill()
```

Jetzt nützen wir noch schamlos aus, dass man in Python auch mehrere Anweisungen in eine Zeile schreiben kann, wenn man dazwischen immer ein Semikolon (ein Strichpunkt-Zeichen) hinschreibt. Das ist zwar nicht die feine englische Art, aber hier im interaktiven Modus geht's halt schneller:

```
>>> fd(100); lt(90)
```



Nun noch drei Mal [Alt] + [P] und [←].

```
>>> fd(100); lt(90)
>>> fd(100); lt(90)
>>> fd(100); lt(90)
```

Jetzt ist das Quadrat fertig und kann gefüllt werden:

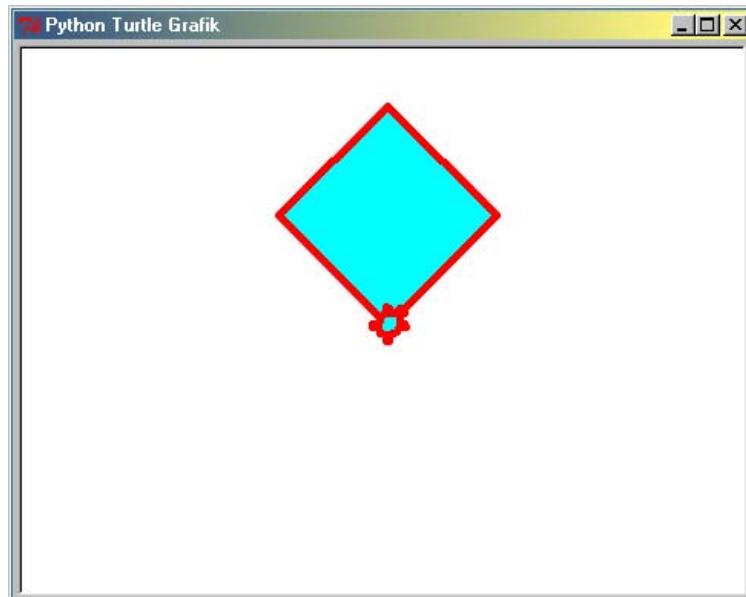
```
>>> end_fill()
```

Wenn du dir dies nochmals genau durchsiehst, wirst du bemerken, dass genau drei Turtle-Grafik-Anweisungen in unsere `quadrat_arbeit.py` eingefügt werden müssen, damit ein cyangefülltes Quadrat erzeugt wird.

➤ Führe diese Ergänzungen in `quadrat_arbeit.py` aus. Speichere und führe das Programm aus. Entsteht ein cyangefülltes Quadrat?

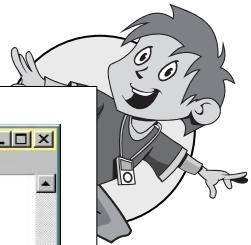
Schritt (2): Das Quadrat soll auf der Spitze stehen. Wenn ich dir jetzt sage, dass du dazu nur eine einzige Anweisung (an der passenden Stelle) einfügen musst, dann hast du nun wohl eine hübsche kleine Denkaufgabe vor dir – oder?

Welche Anweisung ist wo einzufügen, damit das Quadrat auf der Spitze steht?



➤ Wenn du die Aufgabe gelöst hast, ändere den Programmnamen im Kopfkommentar auf `quadrat02.py` sowie die Beschreibung des Programms ab und speichere eine Kopie des Programms unter diesem Namen ab. Wir werden das Original `quadrat_arbeit.py` im Editor-Fenster gleich zur Weiterarbeit verwenden.

Programmcode kopieren



```
quadrat_arbeit.py - C:/py4kids/kap02/quadrat_arbeit.py
File Edit Format Run Options Windows Help
# quadrat02.py: Ein rotes Quadrat, das auf der Spitze
#                      steht, zeichnen.

from turtle import *

reset()
pensize(5)

right(45)      # Das issses!

pencolor("red")
fillcolor("cyan")
begin_fill()
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
end_fill()

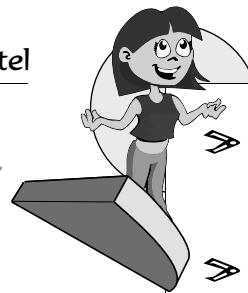
Ln: 21 Col: 0
```

Eine Kopie dieses Programms ist als `quadrat02.py` abzuspeichern.

Programmcode kopieren

Es steht so einsam da! Wir wollen mehr davon. Daher muss der Code, der das Quadrat zeichnet, mehrfach ausgeführt werden. Da es aber ausgesprochen langweilig wäre, dies alles mehrfach einzutippen, wählen wir einen etwas zweckmäßigeren Weg:

- Gehe in `quadrat_arbeit.py` mit dem Cursor ans Ende des Programmtextes und füge dort eine Leerzeile ( drücken!), eine Zeile mit der Anweisung `right(60)` und eine weitere Leerzeile an!
- Markiere (durch Ziehen der Maus mit gedrückter linker Taste) im Programm den Bereich von der `pencolor()`-Zeile bis ans Ende (einschließlich der letzten Leerzeile)!
- Kopiere den markierten Bereich in die Zwischenablage, am besten mit  + !
- Stelle den Cursor an das Ende des Programms (nach der letzten Leerzeile)!
- Drücke  + ! Dies kopiert den Inhalt der Zwischenablage ans Ende des Programms.



Was Schildkröten mit Grafik zu tun haben: Turtle-Grafik

- Speichere das Programm ab: **[Strg]+[S]**. Nun führe es aus: **[F5]**. Du siehst, das Quadrat wird nun zwei Mal gezeichnet und die Turtle schaut danach auch schon in die richtige Richtung für das nächste.
- Drücke vier Mal **[Strg]+[V]**! Dies kopiert den Inhalt der Zwischenablage weitere vier Mal ins Programm. Überprüfe, indem du den Text im Editor-Fenster nach oben schiebst, dass die dreizehn Anweisungen nun wirklich sechs Mal im Programm stehen.
- Sichere das Programm: **[Strg]+[S]** und führe es aus **[F5]**.
- Versuche zu verstehen, was abläuft – nämlich, was die Turtle abläuft?

Kannst du dir ausmalen, wie der Programmablauf gewesen wäre, wenn wir *nicht* vor dem Kopieren die Anweisung `right(60)` angefügt hätten?

Jetzt wäre doch nett, wenn die Quadrate in unterschiedlichen Farben gemalt und gefüllt würden! Dazu brauchen wir sechs Farben. Nehmen wir zunächst die drei Grundfarben deines Computer-Displays: "red", "green", "blue". Und dazu die »Komplementärfarben«. Die Komplementärfarbe einer Grundfarbe ergibt sich aus der Mischung der beiden anderen:

Farbe	Komplementärfarbe
"red"	"cyan"
"green"	"magenta"
"blue"	"yellow"

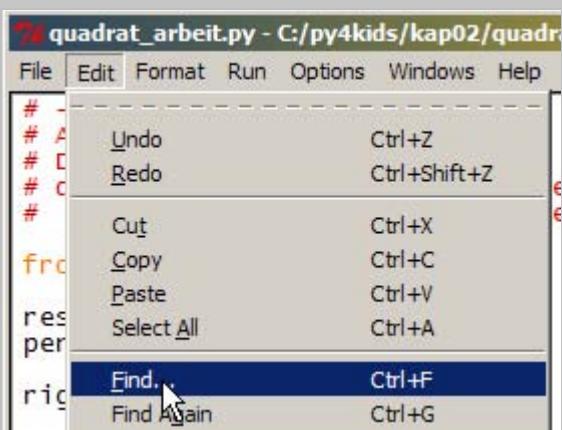
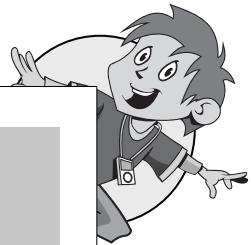
Sorge dafür, dass die Quadrate in verschiedenfarbigen Rändern erscheinen, indem du an den geeigneten Stellen Anweisungen einfügst, die die Stiftfarbe der Turtle auf die Farben "green", "blue", "cyan", "magenta", "yellow", einstellen! (Das erste Quadrat soll weiterhin rot gezeichnet werden.) Du musst dazu fünf Mal den Funktionsaufruf von `pencolor()` aufsuchen.



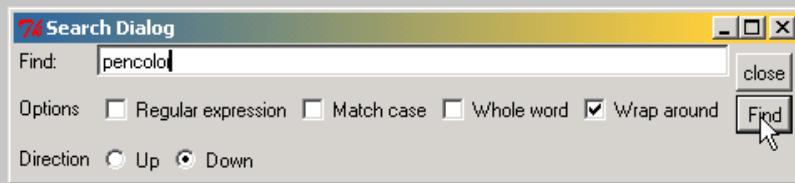
Suchen und finden. Um bestimmte Wörter im Programmtext zu finden (und wiederzufinden), kann man die Suchfunktion des Editor-Fensters benutzen.

Über den Menüpunkt **EDIT|FIND...** (oder über die Tastenkombination **[Strg]+[F]**) öffnet sich ein Suchdialogfenster:

Programmcode kopieren



Finde die Stellen in *quadrat_arbeit.py*...



... wo das Wort *pencolo* vorkommt.

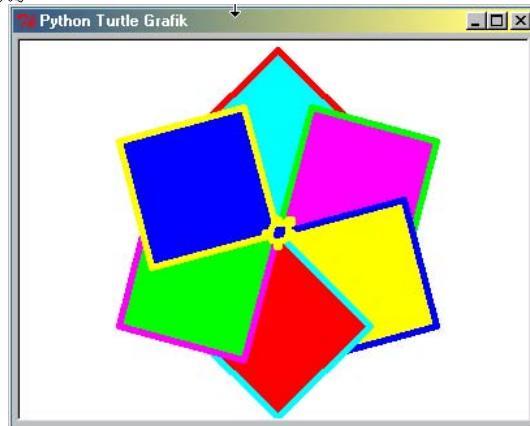
Klicken auf den FIND-Knopf findet das erste Vorkommen des Wortes nach der aktuellen Cursor-Position. Du solltest deshalb vor dem Suchen den Cursor an den Anfang der Datei stellen.

Um weitere Vorkommen des Suchwortes zu finden, verwendest du am besten [Strg] + [G] (Nochmals finden).

➤ Suche nun die Aufrufe von `pencolor()` und trage anstelle von "red" die oben vereinbarten Farben ein. Sichere das Programm und führe es aus.

Jetzt zeigt sich, dass es doch viel schöner wäre, wenn die Quadrate auch mit verschiedenen Farben gefüllt würden. Machen wir es grell:

➤ Ändere die Füllfarben der Quadrate so, dass jedes mit der Komplementärfarbe seiner Randfarbe gefüllt wird. (Für die Stiftfarbe "red" ist als Füllfarbe schon die Komplementärfarbe "cyan" eingetragen.)



Was lernen wir daraus?

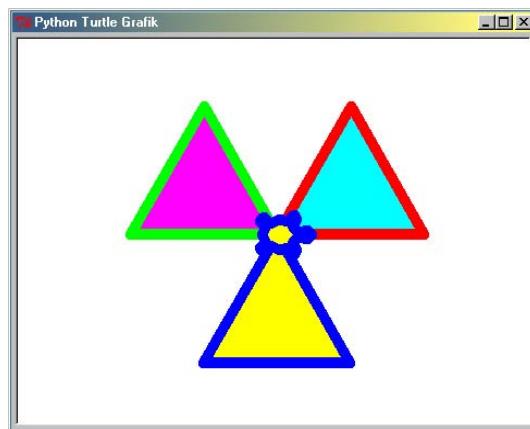
Dies zeigt uns: Unsere Folge von `forward()`- und `left()/right()`-Anweisungen zeichnet immer dieselbe Figur, in unserem Beispiel ein Quadrat. Die Figur kann aber auf der Zeichenfläche mehr oder weniger stark gedreht erscheinen, je nach Ausgangszustand der Turtle, das heißt, in welche Richtung sie am Anfang schaut.

Ist ja ganz logisch! Wenn die Turtle immer dasselbe macht, kommt immer dasselbe raus!

⇒ Speichere eine Kopie von `quadrat_arbeit.py` (nach passender Änderung des Kopfkommentars) als `quadrat03.py` ab.

Und jetzt drei Dreiecke!

Als nächste Aufgabenstellung nehmen wir uns vor, ein Programm zu schreiben, das folgende Grafik erzeugt:



Und jetzt drei Dreiecke!



Die Dreiecke mit der Seitenlänge 135 haben wieder die Randfarben Rot, Grün und Blau und als Füllfarben deren Komplementärfarben. (Oder auch ganz andere Farben, nach deinem Geschmack – siehe Anhang D.)

Diesmal machen wir es einmal so: Wir nehmen wieder Bleistift und Papier und schreiben einen Programmentwurf. Einfach, um uns schon vorher klar zu werden, wie die Sache laufen soll.

So schwer kann das nicht sein, ist es doch der vorigen Übung ähnlich. Überlegen wir mal:

Programmentwurf: drei farbige Dreiecke

Strichdicke 10 Pixel einstellen

Stiftfarbe Rot einstellen

Füllfarbe Cyan einstellen

Dreieck zeichnen

Turtle um ??? Grad drehen

Um wie viel Grad? Nach drei solchen Drehungen sollte die Turtle wieder in die ursprüngliche Richtung schauen. Da muss sie sich also insgesamt um 360° gedreht haben. Also bei einer Drehung nur um ... mal rasch den IPI fragen:

```
>>> 360 / 3
```

```
120.0
```

```
>>>
```

... ur-praktisch!!! ... also um 120° . Jetzt ist ja unser Programmentwurf schon fast geritzt:

Programmentwurf: drei farbige Dreiecke:

Strichdicke 10 Pixel einstellen

Turtle um 90 Grad nach rechts drehen

Stiftfarbe Rot einstellen

Füllfarbe Cyan einstellen

Füllen beginnen

Dreieck zeichnen

Füllen ausführen

Turtle um 120 Grad drehen



Stiftfarbe Grün einstellen
Füllfarbe Magenta einstellen
Füllen beginnen
Dreieck zeichnen
Füllen ausführen
Turtle um 120 Grad drehen

Stiftfarbe Blau einstellen
Füllfarbe Gelb einstellen
Füllen beginnen
Dreieck zeichnen
Füllen ausführen
Turtle um 120 Grad drehen

Doch halt – eine Sache muss noch geklärt werden, bevor es ans Eintippen geht! Wie zeichnet die Turtle ein gleichseitiges Dreieck?

Das sollten wir uns noch genau ansehen:

Programmentwurf: gleichseitiges Dreieck

135 Einheiten vorwärts gehen
Um ??? Grad nach links drehen
...

Welche Überlegung hilft uns hier? Versetze dich in die Lage der Schildkröte und gehe ein gleichseitiges Dreieck ab! Um welchen Winkel hast du dich an den Eckpunkten gedreht? (Dieser Winkel ist der Außenwinkel des gleichseitigen Dreiecks.)

Programmentwurf: gleichseitiges Dreieck

135 Einheiten vorwärts gehen
Um 120° Grad nach links drehen
135 Einheiten vorwärts gehen
Um 120° Grad nach links drehen
135 Einheiten vorwärts gehen
Um 120° Grad nach links drehen



Programm codieren

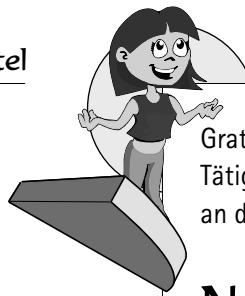
Jetzt musst du nur noch unseren Programmentwurf in Python übersetzen (codieren).

Beginne vielleicht so: Schreibe zuerst ein Programm, das nur *ein* gleichseitiges Dreieck zeichnet. Mit der Kopiertechnik, die du in diesem Kapitel schon geübt hast, kannst du diesen Teil des Programms (Codes) noch zweimal kopieren und dann die Details nachtragen, wie sie im Entwurf festgehalten sind:

- Bevor du beginnst, achte darauf, dass nur das IPI-SHELL-Fenster offen ist! Schließe andere offene Fenster!
- Öffne ein neues Editor-Fenster!
- Schreibe einen Kopfkommentar für das Programm `dreieck_arbeit.py`!
- Schreibe die `import`-Anweisung für die Funktionen des `turtle`-Moduls!
- Schreibe den Code für das Zeichnen des gefüllten gleichseitigen Dreiecks!
- Sichere und führe das Programm aus! Überzeuge dich, dass es richtig funktioniert!
- Aktualisiere den Kopfkommentar und speichere *eine Kopie* des Programms unter `dreieck01.py` ab!

Im Folgenden arbeitest du weiter mit `dreieck_arbeit.py`. Du solltest darauf achten, das Programm während der Entwicklung mehrmals zu testen. Wenn du einige Zeilen Code hinzugefügt hast, probiere öfter mal aus, ob es läuft – und das tut, was du wolltest. Auf diese Weise kannst du Fehler frühzeitig erkennen und beseitigen. Du findest einen Fehler nämlich viel leichter, wenn du weißt, dass er in ein paar neuen Zeilen Code stecken muss. Am Ende, wenn du zu dem Programm vielleicht schon zwanzig oder dreißig Zeilen Code angefügt hast und nun darin womöglich noch drei oder vier Fehler auftreten, ist die Fehlersuche viel mühsamer!

- Kopiere den Dreieckscode noch zweimal in das Programm und stelle das Programm dann entsprechend dem Entwurf fertig!
- Wenn es zufriedenstellend funktioniert, sichere wieder das Programm in Form einer Kopie mit dem Namen `dreieck02.py`.



Gratulation! Jetzt bist du schon recht selbstständig! Für weitere kreative Tätigkeiten gebe ich dir einige weitere Werkzeuge aus dem turtle-Modul an die Hand.

Noch ein paar Turtle-Grafik-Funktionen

Ich zeige dir noch ein paar Turtle-Grafik-Funktionen, die nützlich sind, um abwechslungsreichere Zeichnungen zu gestalten.

➤ Mach mit.

```
>>> reset()
```

Das kennst du schon. Eine frische Zeichenfläche entsteht.

```
>>> pensize(3)
>>> forward(40)
>>> penup()
```

```
>>> pensize(3)
>>> forward(30) I
>>> penup()
()
```

Hebe Zeichenstift an. Zeichne in der Folge nicht.

Aha, penup() ist also eine Turtle-Grafik-Funktion, die kein Argument erwartet. Also machst du die Klammer gleich wieder zu!

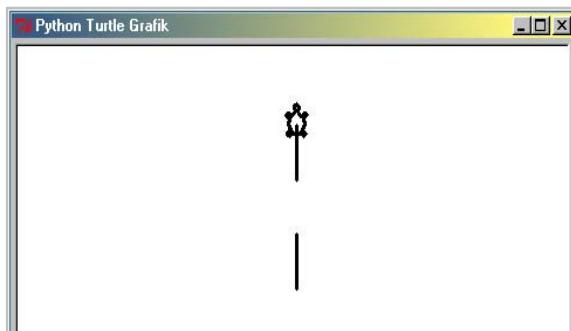
```
>>> penup()
>>> forward(40)
```

Die Turtle hat sich bewegt, aber sie hat nichts gezeichnet! Wir können uns das so vorstellen, dass penup() den Zeichenstift anhebt. Natürlich kannst du ihn auch wieder senken!

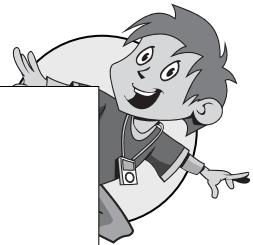
```
>>> pendown()
>>> forward(40)
```

*penup() und
pendown().*

*penup() und
pendown() sind sehr
nützlich – man kann
damit die Turtle an
eine beliebige Stelle des Fensters führen, ohne zu zeichnen.*



Noch ein paar Turtle-Grafik-Funktionen



Beispiel:

```
>>> reset()  
>>> penup()  
>>> forward(120)  
>>> left(90)  
>>> back(120)
```

Hier habe ich gleich noch eine neue Turtle-Grafik-Funktion verwendet: `back()`. Funktioniert wie `forward()`, nur lässt sie die Turtle rückwärts laufen.

Nun steht die Turtle rechts oben im Fenster, schaut nach links und kann dort zum Beispiel ein Quadrat zeichnen. Aber vorher natürlich: Stift runter!

```
>>> pendown()  
>>> pensize(3)  
>>> forward(240)  
>>> left(90)  
>>> fd(240); lt(90)
```

Du kannst im IPI eine Anweisung aus einer früheren Zeile nicht nur mit **Alt**+**P** zurückholen, sondern auch so:

- Gehe mit **↑** eine (wenn nötig auch mehrere) Zeile(n) hinauf, bis der Cursor in der Zeile steht, die nochmals ausgeführt werden soll.
- Drücke **←**! Nun steht `fd(200); lt(90)` in der aktuellen Eingabezeile.
- Drücke nochmals **←**! Die Anweisung wird ausgeführt. (Und dann nochmals!)

```
>>> fd(240); lt(90)  
>>> fd(240); lt(90)
```

Ein schönes Quadrat steht jetzt im Grafik-Fenster.

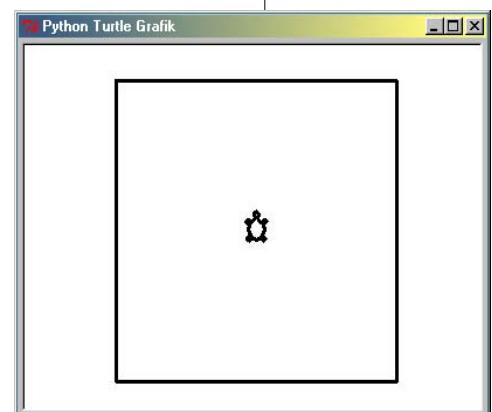
Sein Mittelpunkt ist der Ausgangspunkt der Turtle.

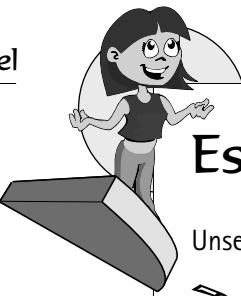
```
>>> penup()  
>>> home()
```

Achtung, sieh zu, was geschieht! Die Turtle geht in die Mitte der Zeichenfläche zurück und stellt sich auf ihre Anfangsorientierung, nach oben, ein. Jetzt hat sie ihre Ausgangslage wieder erreicht!

Die Turtle springt zu einer Ecke, zeichnet das Quadrat und springt wieder zurück.

So kannst du die einzelnen Elemente deiner Zeichnung an beliebigen Stellen im Zeichen-Fenster platzieren und auch die Turtle wieder an ihren Ausgangspunkt zurückführen.





2

Es müssen nicht immer Ecken sein

Unsere Turtle kann auch Kreise (oder auch nur Teile davon) zeichnen:

➤ Mach mit:

```
>>> clear()
```

Dies löscht alle Zeichnungen der Turtle, ohne ihre Lage zu verändern!

```
>>> forward(60)    # Die Feder ist noch oben!
```

```
>>> pendown()
```

```
>>> circle(60)
```

Aha! Die Turtle zeichnet einen Kreis, nach links (im Gegenuhrzeigersinn). Der Radius des Kreises ist 60. Der Mittelpunkt des Kreises liegt links von der Turtle. Hast du den »Call-Tipp« beachtet und bemerkt, dass die Funktion `circle()` auch mehr als ein Argument übernehmen kann?

<pre>>>> circle(80, 180)</pre>
<code>(radius, extent=None, steps=None)</code>
Zeichne Kreis(bogen) mit radius und Winkel extent.

Call-Tipp für die Funktion `circle()`.

Das zweite Argument beschreibt die »Ausdehnung« des Kreisbogens, der gezeichnet werden soll, durch einen Winkel. 180° entspricht daher einem Halbkreis:

```
>>> circle(80, 180)
```

Kürzere Kreisbögen sind auch kein Problem:

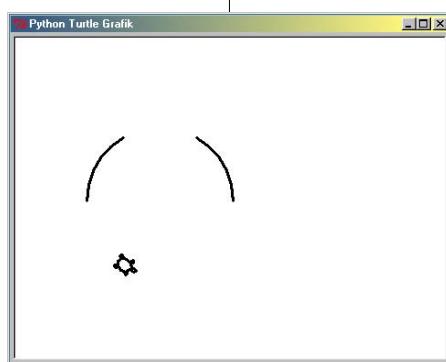
```
>>> reset(); pensize(3)
```

```
>>> circle(80,60); penup(); circle(80,60); pendown()
```

```
>>> circle(80,60); penup(); circle(80,60); pendown()
```

```
>>> circle(80,60); penup(); circle(80,60); pendown()
```

Damit ist es auch leicht, Kreisausschnitte (Kreissektoren) zu zeichnen, wenn man weiß, dass die Kreisradien bei einem Kreissektor immer senkrecht auf den Bogen stehen. Das Argument, das man für `extent` einsetzen muss, ist der Winkel im Kreismittelpunkt. In unserem Beispiel ist das 60° .



Die Turtle nach zwei Kreisbögen.

Es müssen nicht immer Ecken sein

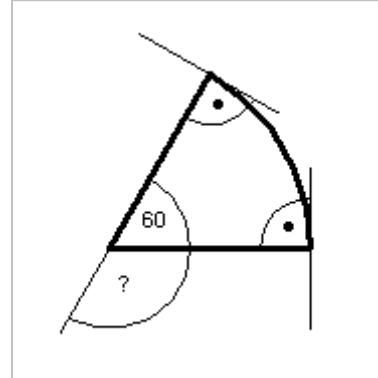
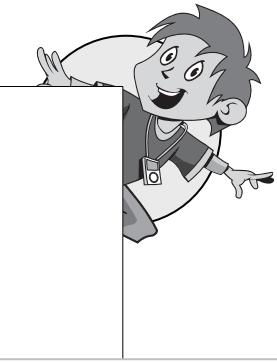
» Mach mit!

```
>>> reset()  
>>> right(90)  
>>> forward(100)  
>>> left(90)  
>>> circle(100,60)  
>>> left(90)  
>>> forward(100)  
>>> left(120)
```

Überlege: wie bestimmt man den Winkel im letzten `left()`-Aufruf? Er hängt wohl mit der Größe des Winkels im Kreissektor, also mit dem zweiten Argument `extent` im `circle()`-Aufruf (5. Zeile) zusammen?

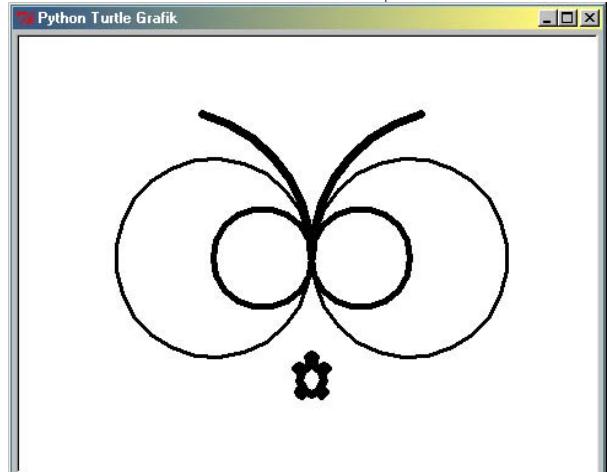
Die Turtle kann mit der Funktion `circle()` auch Kreise zeichnen, die nach der anderen Seite gebogen sind. Dazu muss der Radius negativ gewählt werden:

Kreissektor: Wie groß ist der Winkel für die letzte Drehung der Turtle?



» Mach mit!

```
>>> reset()  
>>> pensize(3)  
>>> circle(80)  
>>> circle(-80)  
>>> pensize(5)  
>>> circle(40)  
>>> circle(-40)  
>>> pensize(7)  
>>> circle(120, 75)  
>>> pu()  
>>> circle(120, -75)  
>>> pd()  
>>> circle(-120, 75)  
>>> pu()  
>>> circle(-120, -75)  
>>> back(100)
```

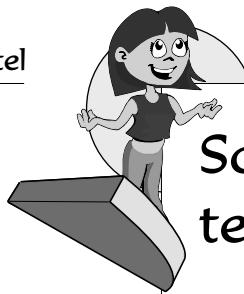


Wenn du für Zeichnungen dieser Art auch noch Farben zum Zeichnen und Füllen verwendest, kann schon wieder eine sehr nette Grafik entstehen.

Ach, du möchtest noch wissen, was `pu()` bedeutet? Verwende

```
>>> help(pu)
```

2



Schildkröte verstecken! Und weitere Kleinigkeiten

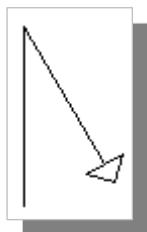
Wenn du mit deiner Turtle nach einiger Mühe eine schöne oder interessante Grafik erstellt hast, hast du vielleicht das Gefühl, dass die Schildkröte selbst gar nicht gut in das Bild passt. `pu(); fd(10000)` wäre eine Möglichkeit, sie zu vertreiben. Ist aber etwas rücksichtslos – und außerdem, vielleicht brauchst du sie später noch. Sanfter geht es mit zwei Turtle-Grafik-Funktionen, die die Turtle verstecken bzw. wieder sichtbar machen:

» Mach mit:

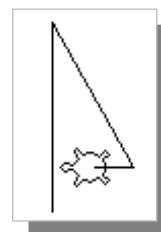
```
>>> reset()
>>> forward(50)
>>> hideturtle()  # und schon isse weg
>>> forward(40)  # auch versteckt kann sie zeichnen
>>> right(150)
>>> forward(80)
>>> showturtle()  # und schon isse wieder da
```

Hast du die Turtle mittlerweile lieb gewonnen? Oder findest du das Tierchen etwas kindisch und sie geht dir womöglich schon auf die Nerven? In diesem Fall kannst du so ihr Aussehen ändern:

```
>>> shape("arrow")
>>> right(120)
>>> forward(20)
>>> shape("turtle")
```



`shape("arrow")`



`shape("turtle")`

Du hast die Wahl! In der Folge nimm dir einfach immer die Turtle-Gestalt, die dir besser zusagt. (Mit der Funktion `shape()` kann man noch viel mehr anfangen, aber das ist etwas für spätere Kapitel ...)

Und noch etwas: Ganz leicht kann es passieren, dass die Turtle beim Zeichnen aus der sichtbaren Zeichenfläche hinausläuft. Sollte aber auch kein Problem sein – in diesem Fall mache einfach mit der Maus das Fenster größer, bis du die ganze erstellte Zeichnung siehst.

Damit hast du nun genügend Mittel in der Hand, um Grafiken zu den Aufgaben am Ende des Kapitels zu programmieren. Noch besser ist es natürlich, zusätzlich auch noch eigene Ideen zu entwickeln und zu verfolgen.



Zusammenfassung

In diesem Kapitel hast du folgende Turtle-Grafik-Funktionen kennen gelernt:

Funktionen aus dem Modul »turtle«

<code>forward(<i>distanz</i>)</code>	Turtle geht um <i>distanz</i> Einheiten vorwärts.
<code>fd(<i>distanz</i>)</code>	
<code>back(<i>distanz</i>)</code>	Turtle geht um <i>distanz</i> Einheiten rückwärts.
<code>bk(<i>distanz</i>)</code>	
<code>left(<i>winkel</i>)</code>	Turtle dreht sich um <i>winkel</i> Grad nach links.
<code>lt(<i>winkel</i>)</code>	
<code>right(<i>winkel</i>)</code>	Turtle dreht sich um <i>winkel</i> Grad nach rechts.
<code>rt(<i>winkel</i>)</code>	
<code>undo()</code>	Macht die letzte Turtlegrafik-Anweisung rückgängig. Kann mehrfach angewendet werden.
<code>circle(<i>radius</i>, <i>extent</i>)</code>	Turtle zeichnet einen Kreis oder, wenn <i>extent</i> angegeben ist, einen Kreisbogen, der zum Winkel <i>extent</i> gehört. Wenn <i>radius</i> > 0 ist, liegt der Kreismittelpunkt links von der Turtle, wenn <i>radius</i> < 0 ist, rechts von der Turtle im Abstand <i>radius</i> .
<code>penup()</code>	Turtle hebt den Zeichenstift an und bewegt sich ab jetzt, ohne eine Spur zu hinterlassen.
<code>pu()</code>	
<code>pendown()</code>	Turtle senkt den Zeichenstift und zeichnet ab jetzt bei Bewegungen ihre Spur.
<code>pd()</code>	
<code>pensize(<i>dicke</i>)</code>	Stellt die Strichdicke für den Zeichenstift der Turtle auf <i>dicke</i> Pixel ein.
<code>pencolor(<i>farbe</i>)</code>	Stellt die Farbe für den Zeichenstift der Turtle ein. <i>farbe</i> ist ein String wie "red", "blue" usw.
<code>fillcolor(<i>farbe</i>)</code>	Stellt die Füllfarbe der Turtle ein.
<code>begin_fill()</code>	Schaltet Füll-Modus der Turtle ein. Wird aufgerufen, wenn das Zeichnen einer Figur begonnen wird, die gefüllt werden soll.
<code>end_fill()</code>	Schaltet Füll-Modus der Turtle aus. Wird aufgerufen, wenn die Figur fertig gezeichnet ist, um sie zu füllen.
<code>reset()</code>	Löscht alle Zeichnungen im Grafik-Fenster und setzt den Zustand der Turtle (Ort, Strichdicke, Farbe, Strich, Stift unten) auf die Standardwerte zurück.
<code>clear()</code>	Löscht alle Zeichnungen im Grafik-Fenster. Zustand der Turtle (Ort, Strichdicke, Farbe, Stift oben/unten) bleibt erhalten.
<code>home()</code>	Führt die Turtle in ihre Ausgangslage und Ausgangsorientierung (Nord) zurück.
<code>hideturtle()</code>	Macht die Turtle unsichtbar, ohne ihre sonstigen Eigenschaften zu verändern.
<code>showturtle()</code>	Macht die Turtle sichtbar.
<code>shape(<i>form</i>)</code>	Stellt die Form der Turtle ein. Als <i>form</i> kann unter anderem "turtle" oder "arrow" gewählt werden.

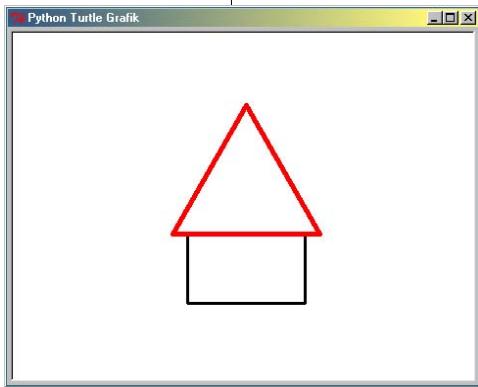


Außerdem hast du gesehen:

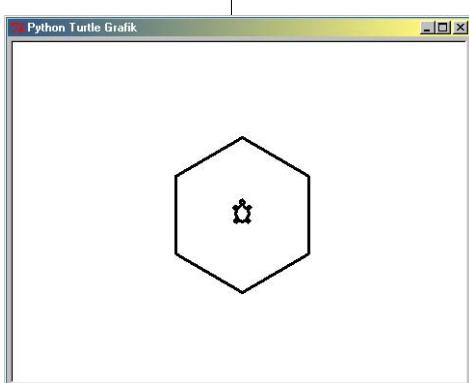
Für kompliziertere Programmieraufgaben ist es sinnvoll, einen *Programmentwurf* zu schreiben. Dabei ist zu beachten:

- ❖ Schreibe den Entwurf mit deutschen Worten!
- ❖ Schreibe ihn in einer ähnlichen Form wie ein Programm: Eine Zeile soll jeweils einer Anweisung entsprechen.
- ❖ Manchmal ist es sinnvoll, einer Zeile den Namen einer Teilaufgabe zu geben. Diese Teilaufgabe kann dann in einem eigenen (Unter-)Programmentwurf genauer beschrieben werden.

Einige Aufgaben ...



Aufgabe 1: Schreibe ein Turtle-Grafik-Programm, haus01.py, das ein einfaches Haus zeichnet! Wähle die Abmessungen so, dass sie ungefähr der folgenden Abbildung entsprechen!

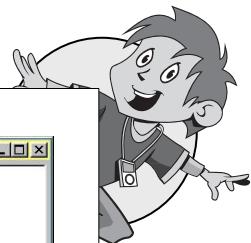
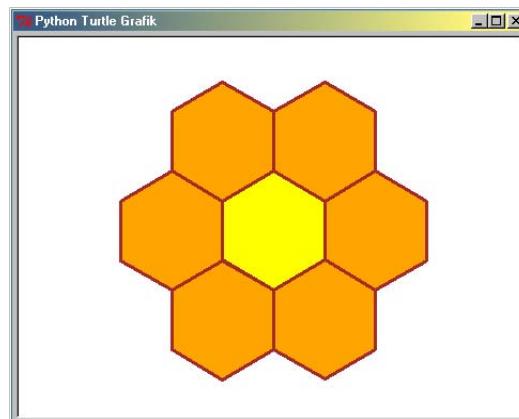


Aufgabe 2: (a) Schreibe ein Turtle-Grafik-Programm, sechseck01.py, das ein regelmäßiges Sechseck mit der Seitenlänge 80 zeichnet. **(b)** Sobald du das erreicht hast, ändere das Programm so ab, dass der Mittelpunkt des Sechsecks die Anfangsposition der Turtle ist (sechseck02.py).

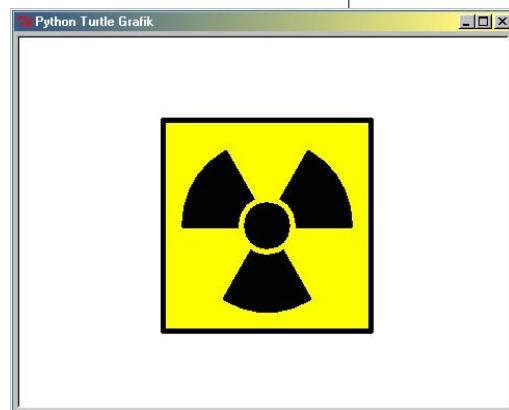
Bei den folgenden drei umfangreicheren Aufgaben empfiehlt sich ein Programmentwurf. Verwende für den Entwurf Skizzen und erstelle eine Programmbeschreibung in der Form, wie wir es für das Programm »drei farbige Dreiecke« gemacht haben!

Einige Aufgaben ...

Aufgabe 3: Schreibe ein Turtle-Grafik-Programm, wabe.py, das eine solche Bienenwabe zeichnet. Die Sechsecke sollen in brauner Farbe gezeichnet werden. Die äußeren Sechsecke sind orange, das innere ist gelb gefüllt. Du wirst sehen, das schaut richtig nach Honig aus!



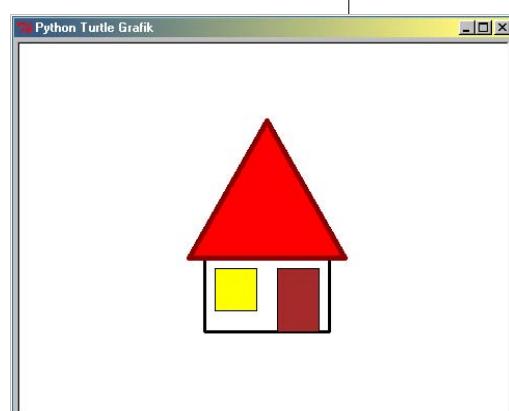
Aufgabe 4: Schreibe ein Turtle-Grafik-Programm, radioaktiv.py, das ein Radioaktivitätssymbol zeichnet. Zerlege dir die Aufgabe in Teilaufgaben: (1) Zeichne ein großes gelb gefülltes Quadrat mit schwarzem Rand. (2) Zeichne in dieses Quadrat drei schwarze Kreissektoren. (3) Zeichne darüber einen kleinen schwarzen Kreis mit gelbem Rand. Vielleicht ist es günstig, diese Teilaufgaben getrennt zu lösen und dann zu einer Gesamtlösung zusammenzusetzen.

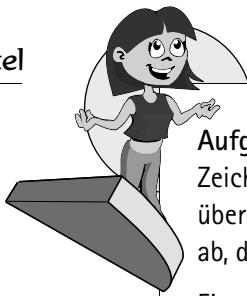


Aufgabe 5: Schreibe ein Turtle-Grafik-Programm, haus02.py, das folgendes Haus zeichnet:

Gehe dazu von der Lösung von Aufgabe 1 aus!

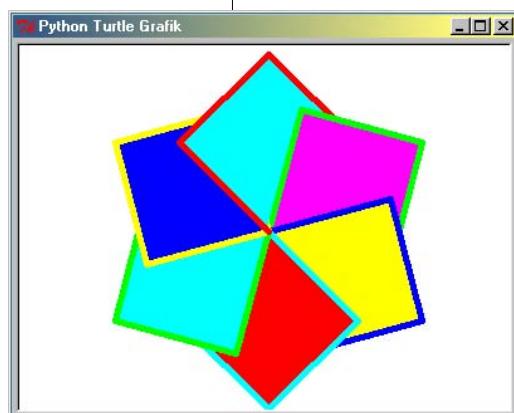
Aufgabe 6: Finde ein Logo, ein Markenzeichen, ein grafisches Symbol, ein Graffiti oder was immer, das dir besonders gut gefällt, und programmiere es mit Turtle-Grafik nach. Wenn du mit deiner Lösung zufrieden bist, sende mir dein Programm per E-Mail: glingl@aon.at. Die schönsten Grafiken werde ich auf der Website <http://python4kids.net> veröffentlichen. (Wenn du selbst keine Idee hast, sieh auf der angegebenen Website nach – vielleicht willst du etwas nachprogrammieren.)





Aufgabe 7: In nebenstehender Abbildung ist unsere Sechs-Quadrat-Zeichnung so verbessert worden, dass das letzte Quadrat das erste nicht überdeckt. Ändere `quadrat03.py` so zu `aufgabe-02-07_quadrat.py` ab, dass es diese Grafik erzeugt.

Eine Lösung findest du leichter, wenn du vorher folgende Übung ausführst:



```
>>> from turtle import *
>>> pensize(9)
>>> fillcolor("yellow")
>>> begin_fill()
>>> fd(140); rt(120)
>>> pencolor("red")
>>> fd(140); rt(120)
>>> penup()
>>> fd(140); rt(120)
>>> end_fill()
```

... und einige Fragen

1. Was bewirken die Turtle-Grafik-Funktionen
 (a) `reset()`
 (b) `clear()` und
 (c) `home()`?
2. Du siehst folgende Anweisung:
`back(55)`
 - a) Was bewirkt diese Anweisung?
 - b) Welcher Typ von Anweisung ist das?
 - c) Wie heißt die Funktion?
 - d) Was ist hier das Argument?
3. Im IPI gibt es zwei Wege, früher eingegebene Anweisungen in die aktuelle Eingabezeile zu holen. Welche sind das?
4. Wie findet man in einem Programm (in einem Editor-Fenster) am einfachsten ein bestimmtes Wort? Wie findet man weitere Vorkommen desselben Wortes?
5. Welche Anweisungen werden benötigt, um eine gezeichnete Figur mit einer Farbe zu füllen?



3 Namen

Im zweiten Kapitel hast du Programme geschrieben, die zwar allerhand zeichnen können, aber teilweise sehr lang waren. Wenn du nun diese Programme ändern möchtest, beispielsweise um die Größe der Zeichnung zu ändern, ist das eine Menge Arbeit. Unerfreulich! Doch da gibt es viele Wege, sich diese Arbeit zu erleichtern. Einen davon werden wir jetzt beschreiben.

In diesem Kapitel lernst du ...

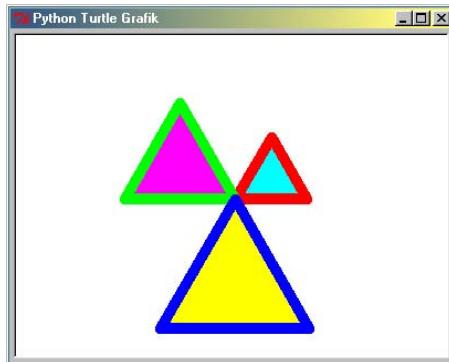
- ◎ mit welchen Dingen Python und die Turtle beim Zeichnen arbeiten.
- ◎ für diese Dinge Namen einzuführen.
- ◎ wie vielseitig man sie verwenden kann.
- ◎ mit kleinen Programmänderungen deine Grafiken zu verändern.
- ◎ Benutzereingaben zu programmieren.
- ◎ den Unterschied zwischen Zahlen und Zeichen kennen.

3



Verschieden große Dreiecke

dreieck03.py zeichnet dieses Bild.

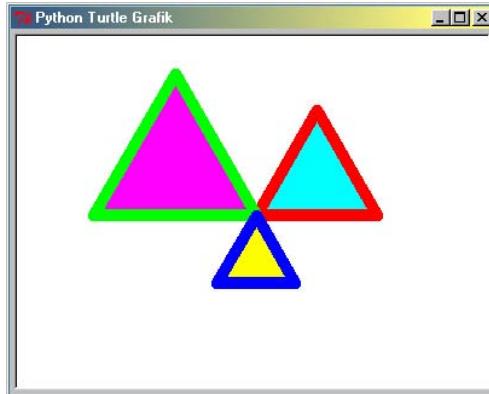


⇒ Überlege, was dazu an unserem Programm dreieck02.py zu ändern ist! Führe diese Änderungen aus und speichere das Programm unter dem Namen dreieck03.py.

Das war nicht schwierig, oder? Für das erste Dreieck waren die drei forward(135)-Anweisungen einfach durch drei andere – zum Beispiel forward(65)-Anweisungen – zu ersetzen. Und für das zweite Dreieck etwa durch drei forward(100)-Anweisungen!

Oder wäre es dir so lieber?

dreieck04.py zeichnet dieses Bild.



Jetzt müsstest du wieder alle diese forward()-Anweisungen ändern? Auf die Dauer ist das irgendwie langweilig. (Deswegen findest du die Programme dreieck03.py und dreieck04.py auch auf der Buch-CD.)

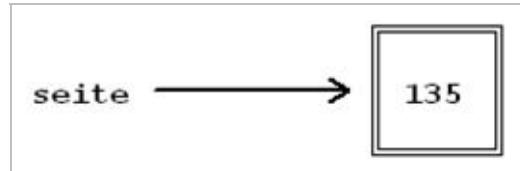
Überdenken wir die Aufgabe, ein Dreieck zu zeichnen: Da haben wir zunächst ein Ding, den Zahlenwert 135. Dieser spielt die Rolle der Seitenlänge des Dreiecks. Dieses Ding – 135 – muss an drei Stellen beim Funktions-

Spielerei mit Namen



aufruf von `forward()` als Argument eingesetzt werden. Will man ein anderes Ding dafür verwenden (z. B. den Zahlenwert 65), muss man das an drei Stellen ändern.

Um diese Umstandsmeierei zu vermeiden, gibt es in Python wie in allen Programmiersprachen eine einfache Technik: *Wir geben dem Ding einen Namen*. Damit der Name auch ausdrückt, welche Rolle das Ding spielt, wählen wir hier zum Beispiel den Namen `seite`.



Wertzuweisung: Der Name `seite` verweist auf den Wert 135.

Wenn wir dann in irgendwelchen Anweisungen diesen Namen anschreiben, wird der Python-Interpreter das Ding verwenden, das wir mit diesem Namen bezeichnet haben.

Spielerei mit Namen

Bevor ich dir zeige, was das bringt, machen wir in einer IPI-Sitzung eine kleine Spielerei mit Namen.

- Starte IPI-TurtleGrafik
- Mach mit!

```
>>> seite = 80
```

Durch diese Wertzuweisung wird der Name `seite` erzeugt und zwar für den Zahlenwert 80. Fortan ist dieser Name dem Python-Interpreter bekannt – ebenso wie das Ding, das er bezeichnet. Um ihn zu fragen, welches Ding den Namen `seite` hat, schreiben wir einfach den Namen `seite` hin:

```
>>> seite  
80  
>>> 4 * seite  
320
```

Die letzte Eingabe zeigt, dass der IPI mit Namen auch rechnen kann. (Natürlich rechnet der IPI mit den Dingen, auf die die Namen verweisen: also mit den Zahlen.) Kann man Namen auch beim Zeichnen verwenden?

```
>>> from turtle import *  
>>> pensize(3)  
>>> forward(seite)
```

3



Das ist fein! Wir werden mutig und probieren noch:

```
>>> left(90)
>>> forward(2*seite)
```

Der IPI kann also beim Zeichnen auch rechnen.

```
>>> left(90)
>>> forward(seite)
>>> left(90)
>>> forward(2*seite)
```

Somit haben wir ein Rechteck erhalten. (Sollte irgendwas falsch gelaufen sein – zum Beispiel die Turtle – denk an `undo()`). Zeichnen wir noch ein Rechteck, in die andere Richtung:

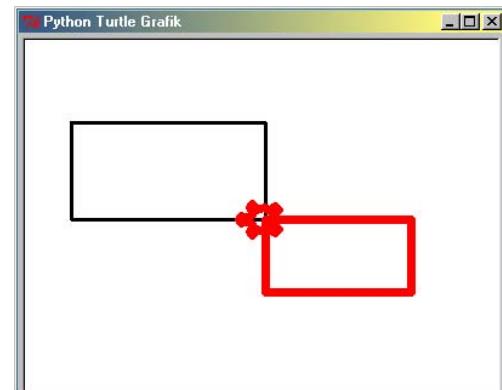
```
>>> right(90)
```

Aber nun wollen wir mit `seite` einen anderen Wert bezeichnen. Außerdem will ich rot und dicker weiterzeichnen.

```
>>> seite = 60
>>> pencolor("red")
>>> pensize(7)
```

... und noch ein Rechteck (benutze `Alt+P` oder `↑-Tasten + ←!`):

```
>>> forward(seite)
>>> left(90)
>>> forward(2*seite)
>>> left(90)
>>> forward(seite)
>>> left(90)
>>> forward(2*seite)
```



Zwei Rechtecke mit verschiedenen Werten von `seite`.

Nun hat dieselbe Anweisungsfolge ein kleineres Rechteck erzeugt.

Wir machen die Dreiecksseite variabel

Diese Technik werden wir jetzt anwenden, um unser letztes Dreieck-Programm aus Kapitel 2 weiterzuentwickeln.

Wir machen die Dreiecksseite variabel



- Lade dreieck_arbeit.py aus dem Ordner py4kids\kap02 und speichere es sofort (mit dem gleichen Namen) in py4kids\kap03.

Wir wollen es nun so ändern, dass es eine Grafik mit den drei verschiedenen großen Dreiecken erzeugt, wie sie am Anfang dieses Abschnitts abgebildet ist.

In diesem Programm wird dreimal ein Dreieck gezeichnet. Dazu wird (unter anderem) neun Mal forward(135) aufgerufen. Wir wollen die 135 in den forward()-Aufrufen durch einen Namen, seite, ersetzen. Dazu führen wir zunächst den Namen ein, indem wir ihm den Zahlenwert 135 zuweisen. Danach ändern wir die forward()-Anweisungen ab.

- Füge nach der vierten Anweisung (also vor der Zeile pencolor("red")) die neue Zeile seite = 135 ein!

```
*dreieck_arbeit.py - C:/py4kids/kap03/dreieck_arbeit.py
File Edit Format Run Options Windows Help
# Python für Kids -- 4. Auflage
# Autor: Gregor Lingl
# Datum: 5. 8. 2008
# dreieck_arbeit.py: 3 gleichseitige Dreiecke

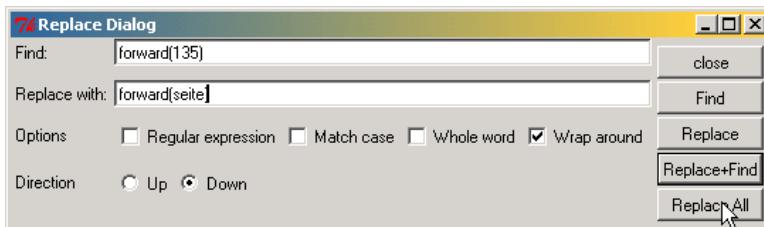
from turtle import *
reset()
pensize(10)
right(90)

seite = 135      # <== ein Name!

pencolor("red")
fillcolor("cyan")
```

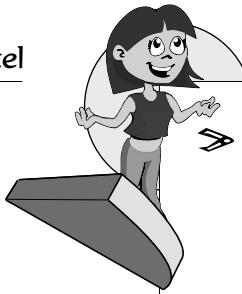
Als Nächstes müssen wir alle neun Einträge forward(135) auf forward(seite) abändern. Da hilft uns wieder das Editor-Fenster mit dem EDIT|REPLACE...-Menü. Das geht so:

- Wähle den Menüpunkt EDIT|REPLACE... oder drücke **Strg**+**H**. Es erscheint ein »Suchen-Ersetzen-Dialogfenster«:



- Trage in der Zeile FIND: forward(135) ein und in der Zeile REPLACE WITH: forward(seite). Klicke auf die Schaltfläche REPLACE ALL.
- Speichere (**Strg**+**S**) und führe das Programm aus (**F5**)! Das Programm sollte korrekt laufen und dieselbe Grafik erzeugen wie das ursprüngliche Programm.
- Ändere nun in der Anweisung seite = 135 am Anfang des Programms den Wert, der dem Namen seite zugewiesen wird, von 135 auf 100 ab!
- Führe das Programm aus! Du siehst, dass nun alle drei Dreiecke kleiner gezeichnet worden sind.

3



- ⇒ Wenn du Lust dazu hast, mache dasselbe vielleicht auch noch mit einem erheblich größeren Wert. Dazu musst du wahrscheinlich auch das Grafik-Fenster vergrößern, um das ganze Bild sehen zu können. Setze am Ende den Wert von `seite` wieder auf 135 zurück.

Damit haben wir im Vergleich zu `dreieck02.py` schon eine wichtige Verbesserung erreicht: Du kannst jetzt unsere Drei-Dreiecke-Figur in jeder beliebigen Größe zeichnen lassen, indem du nur einen einzigen Zahlenwert in dem Programm änderst, nämlich den Zahlenwert, der den Namen `seite` hat.

Wie du schon im IPI gesehen hast, darf man den Wert, auf den ein Name verweist, innerhalb eines Programms durchaus ändern. Das wollen wir jetzt auch tun und zwar so, dass die drei Dreiecke verschieden groß gezeichnet werden.

- ⇒ Markiere die Zeile, in der `seite = 135` steht! Kopiere sie in die Zwischenablage ([Strg]+[C]) und füge sie zwei Mal, jeweils vor der `pen-color()`-Anweisung des zweiten und dritten Dreiecks ein ([Strg]+[V])!
- ⇒ Ändere in der ersten Zuweisung an `seite` den Zahlenwert auf 65 ab und in der zweiten auf 100!
- ⇒ Sichere das Programm und führe es aus!

Hast du eine Grafik erhalten, die wie die erste Abbildung in diesem Kapitel aussieht?

- ⇒ Speichere eine Kopie des Programms als `dreieck05.py` ab. (Kopfkommentar!)
- ⇒ Ändere `dreieck_arbeit.py` nun noch so ab, dass es ein Bild wie `dreieck04.py` erzeugt. Speichere dann wieder eine Kopie als `dreieck06.py` ab.

Beachte, dass zur Durchführung dieser Aufgabe im Programm (abgesehen vom Kopfkommentar) nur drei Zeilen geändert werden müssen. Schon dies zeigt dir, dass die Einführung von Namen sehr praktisch ist. Zum Vergleich sieh dir nochmals `dreieck03.py` an und überlege, wie viel davon geändert werden muss, um `dreieck04.py` zu erzeugen.



Dinge brauchen Namen

Damit man in einem Python-Programm Dinge, die immer wieder gebraucht werden, leicht benutzen kann, gibt man solchen Dingen einen Namen. Programmierer sagen statt Ding lieber *Objekt*. Also werde ich das ab jetzt auch so machen und du wirst dich bald daran gewöhnen, dass mit »Objekten« immer irgendwelche Dinge gemeint sind.

Wenn du das Vorwort zu diesem Buch oder auch nur den hinteren Buchdeckel durchgelesen hast, dann weißt du ja schon, dass Python eine objekt-orientierte Programmiersprache ist.

Was das bedeutet – und das ist eine ganze Menge –, wird dir erst im Laufe der Arbeit mit diesem Buch klarer werden. Hier soll aber schon Folgendes festgehalten werden: In Python ist (fast) alles ein Objekt.

Manche Objekte sind einfach – wie zum Beispiel Zahlen. Zahlen kann man sogar hinschreiben, ohne ihnen einen Namen zu geben, wie eben z. B. 135. Trotzdem – in der letzten Aufgabe hast du ja gesehen, dass es äußerst vorteilhaft sein kann, für Zahlen Namen einzuführen.

Andere Objekte sind komplizierter. Schreibt man den Namen eines Objekts, dann erfolgt eine Beschreibung (Typ, Name, Identitätsnummer). Diese wird bei dir sicher einen anderen Wert haben.

```
>>> penup  
<function penup at 0x02032170>
```

Du siehst: penup ist auch ein Name für ein Ding, und zwar – wie du ja schon weißt – für eine Funktion aus dem Modul turtle. Damit sage ich nichts anderes, als dass auch Funktionen Objekte sind. Eine Funktion ist wohl ein ganz anderer Typ von Ding als eine Zahl. Sie hat zum Beispiel die Eigenschaft, dass man sie aufrufen kann: penup(). Schreibt man die Klammern dazu, dann wird das als Aufruf interpretiert und die Funktion penup wird ausgeführt.

Kannst du dir klar machen, dass du die Funktion penup() gar nicht benutzen könntest, wenn sie keinen Namen hätte?

Um auszudrücken, dass man sie aufrufen kann, kennzeichne ich ab jetzt *Funktionen* im Text immer durch ein Klammernpaar () hinter dem Namen. Beispiel:

Die Funktion pencolor() hat einen Parameter.



3



Das Klammerpaar drückt einfach aus, dass `pencolor` der Name einer Funktion ist. Die Parameter der Funktion, falls sie welche hat, lasse ich in diesem Zusammenhang weg.

In Python gibt es Objekte der unterschiedlichsten Typen, die du erst ganz allmählich kennen lernen wirst. Zu allem Überfluss kann der Programmierer auch noch eigene Typen von Objekten dazu erfinden. Die meisten davon kannst du nur über Namen ansprechen und verwenden.

Schon im Beispiel `dreieck05.py` mit den drei Dreiecken hast du gesehen, dass der Name `seite` nicht immer für denselben Zahlenwert steht.

Zunächst ist `seite` der Name für den Wert 65. Dies haben wir erreicht durch die Programm-Anweisung

```
seite = 65
```

Einige Anweisungen weiter wird dem Namen `seite` der Wert 100 und schließlich der Wert 135 zugewiesen. Diese spezielle Programmanweisung nennt man eine *Zuweisung*.

Wenn das Objekt ein einfacher Wert ist, zum Beispiel ein Zahlenwert wie 60, spricht man auch von einer *Wertzuweisung*.

Die Form einer einfachen Zuweisung ist:

```
Name = Objekt
```

Die allgemeine Form einer Zuweisung ist:

```
Name = Ausdruck
```

Wir sagen dann: *Name verweist auf Objekt*. Oder auch: Der Wert von *Name* ist *Objekt*.

Der IPI ist sehr gut geeignet, um mit Wertzuweisungen zu experimentieren. Wir verwenden ihn daher jetzt, um zu erforschen, wie das genau zu verstehen ist.

➤ Mach mit!

Einfache Wertzuweisung:

```
>>> breite = 30
```

Der Name `breite` verweist auf das Objekt 30. Oder: Der Name `breite` verweist auf den Wert 30. Oder: Der Wert von `breite` ist 30.

Wie groß ist `breite`? Oder: Welchen Wert hat `breite`? Oder: Auf welches Objekt verweist `breite`?

Dinge brauchen Namen

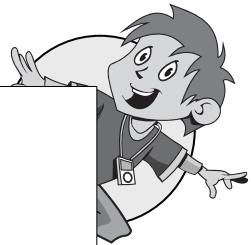
```
>>> breite
```

```
30
```

Noch eine einfache Wertzuweisung:

```
>>> 3fach = 90
```

```
SyntaxError: invalid syntax (<pyshell#24>, line 1)
```



In Python gilt folgende *Syntax-Regel*: Namen dürfen nur die Buchstaben A bis Z und a bis z, Ziffernzeichen 0 bis 9 und den Unterstrich _ enthalten. Das erste Zeichen in einem Namen darf kein Ziffernzeichen sein! Es soll kein Unterstrich sein, außer es gibt ganz besondere Gründe dafür. Außerdem dürfen die reservierten Wörter von Python nicht als Namen verwendet werden.

Obwohl es möglich wäre, ist es besser, Umlaute und das ß nicht in Namen zu verwenden. Leser deiner Programme, die in fernen Ländern wohnen, könnten eventuell damit nicht zurande kommen.

In Python wird meistens die Vereinbarung eingehalten, Namen für Objekte mit einem *Kleinbuchstaben* zu beginnen.

Weil 3fach mit einem Ziffernzeichen beginnt, kann es also nicht als Name verwendet werden.

Sehen wir uns noch eine Wertzuweisung der *allgemeinen Form* an:

```
>>> laenge = 2 * breite + 122.5 / 5
```

Hier steht rechts vom Zuweisungsoperator = ein (arithmetischer) Ausdruck.

```
>>> laenge
```

```
84.5
```

Der Ausdruck rechts vom Zuweisungsoperator darf auch Namen enthalten:

```
>>> umfang = 2 * laenge + 2 * breite
```

```
>>> umfang
```

```
229.0
```

Er darf sogar denselben Namen enthalten, der links vom Zuweisungsoperator steht:

```
>>> irgendwas = laenge - breite
```

```
>>> irgendwas
```

```
54.5
```

```
>>> irgendwas = 10 * irgendwas
```

```
>>> irgendwas
```

```
545.0
```

3



Wie funktioniert das? Zuerst wird der Ausdruck rechts vom = ausgewertet: `10 * irgendwas`. Das heißt bei einem arithmetischen Ausdruck: Er wird einfach ausgerechnet. Diese Auswertung ergibt einen Wert – in unserem Beispiel `545.0`. Dieser Wert wird dann dem Namen, der links vom Zuweisungsoperator = steht, zugewiesen, auch wenn dieser Name selbst im rechten Ausdruck vorkommt. Damit ist `irgendwas` nun der Name für `545.0` und nicht mehr für `54.5`.

```
>>> dasselbe = irgendwas  
>>> dasselbe  
545.0
```

Damit wird einem weiteren Namen, `dasselbe`, der Wert von `irgendwas`, also ebenfalls `545.0`, zugewiesen.

Ein bestimmter Fehler kommt sehr häufig vor – nicht nur bei Programmieranfängern:

```
>>> wasanderes = 10 * wasanderes  
Traceback (most recent call last):  
  File "<pyshell#17>", line 1, in <module>  
    wasanderes = 10 * wasanderes  
NameError: name 'wasanderes' is not defined
```

Auf den ersten Blick schaut das ja genauso aus wie die vorige Eingabe mit `irgendwas`. Doch hatte dort `irgendwas` vorher den Wert `54.5` zugewiesen bekommen. Hier aber hat `wasanderes` auf der rechten Seite noch keinen Wert und deshalb gibt es auch kein Objekt mit diesem Namen.

```
>>> wasanderes  
Traceback (most recent call last):  
  File "<pyshell#18>", line 1, in <module>  
    wasanderes  
NameError: name 'wasanderes' is not defined
```

Wenn es `wasanderes` nicht gibt, kann auch der Ausdruck `10 * wasanderes` nicht ausgewertet werden.

In Python wird ein Name dadurch erzeugt, dass ihm ein Objekt zugewiesen wird!

```
>>> wasanderes = -1  
>>> wasanderes = 10 * wasanderes  
>>> wasanderes  
-10
```



Dinge brauchen Namen



Achtung! Jetzt zeige ich dir noch etwas, was nie und nimmer funktionieren kann, aber doch ein häufig vorkommender Anfängerfehler ist:

```
>>> laenge * breite = 50 * 30      # FEHLER !!!  
SyntaxError: can't assign to operator (<pyshell#27>, line 1)
```

Auch dies ist ein Syntaxfehler. Hier steht links vom Zuweisungsoperator = ein Ausdruck (und zwar einer mit einem *-Operator). Einem Ausdruck kann nichts zugewiesen werden. Nur einem Namen kann etwas zugewiesen werden. Deshalb darf links vom = immer nur *ein Name* stehen. (Na ja, es gibt da schon noch was anderes; aber bis auf weiteres ist das für dich ein sehr guter Rat!)

Da die Objekte, auf die ein Name verweist, wechseln können, werden Namen häufig auch als Variablennamen oder kurz *Variable* bezeichnet. Variable heißt ja nichts anderes als veränderlich.

Zuweisungen sind so wichtige und häufige Anweisungen in Computerprogrammen, dass sich dafür auch eine grafische Darstellung eingebürgert hat:

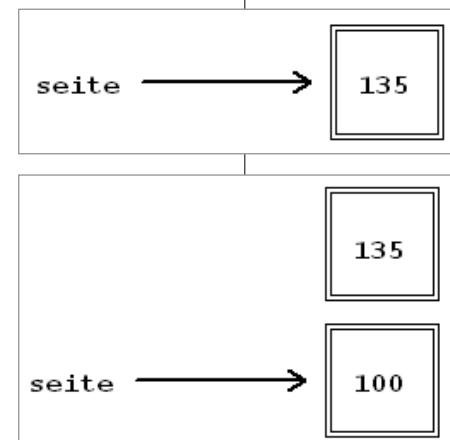
```
seite = 135
```

Wenn während des Programmablaufs der Name `seite` einen anderen Wert zugewiesen bekommt, entsteht folgende Situation:

```
seite = 100
```

Auf das Objekt 135 verweist nun kein Name mehr. Wird neuerlich 135 gebraucht, muss es neu erzeugt werden. Der Name `seite` verweist ja jetzt auf das Objekt 100.

Wir nehmen Zuweisungen wegen ihrer großen Bedeutung in unsere Muster-Sammlung auf, obwohl sie sozusagen nur ein ganz kleines Muster darstellen:



Muster 2: Einfache Zuweisung, auch: Wertzuweisung

(1.) `name = objekt`

← Objekt wird dem Namen zugewiesen,
Name verweist auf Objekt

(2.) `name = Ausdruck`

↓ Ausdruck wird ausgewertet. Die Auswertung ergibt ein
Objekt. Dieses Objekt wird dem Namen zugewiesen.

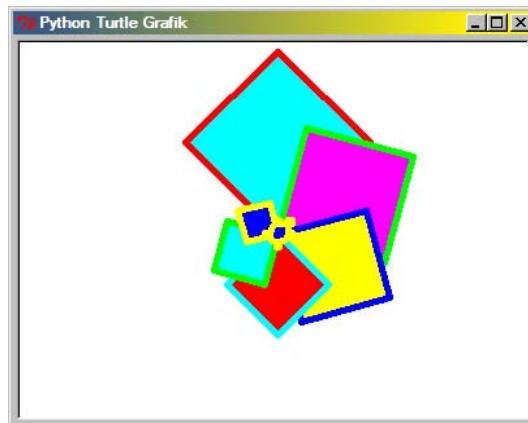
Das Zeichen = heißt Zuweisungsoperator.



3

Übung: Verschieden große Quadrate

- ⇒ Lade quadrat_arbeit.py aus Kapitel 2 und speichere es im Ordner py4kids\kap03.
- ⇒ Ändere es so ab, dass das Programm die folgende Zeichnung erzeugt. Die Seitenlängen der Quadrate sollen 100, 85, 70, 55, 40 und 25 Einheiten groß sein. Verwende dazu wieder einen Namen `seite` für die Seitenlängen der sechs verschiedenen Quadrate. Wenn das Programm fertig ist, speichere eine Kopie unter dem Namen `quadrat04.py`.



Findest du nicht auch, dass das letzte Quadrat etwas klein geraten ist? Nehmen wir lieber die Seitenlängen 100, 90, 80, 70, 60 und 50!

Wenn du vor dem Code für jedes Quadrat eine Zuweisung `an seite` geschrieben hast, dann brauchst du jetzt nur sechs Anweisungen zu ändern, um die Quadratspirale schöner zu machen. Das ist dir immer noch zu viel?

Schau dir einmal folgenden Vorschlag an!

Vor das erste Quadrat schreiben wir:

```
seite = 100
```

Die fünf anderen Zuweisungen an den Namen `seite` ersetzen wir alle durch

```
seite = seite - 10
```

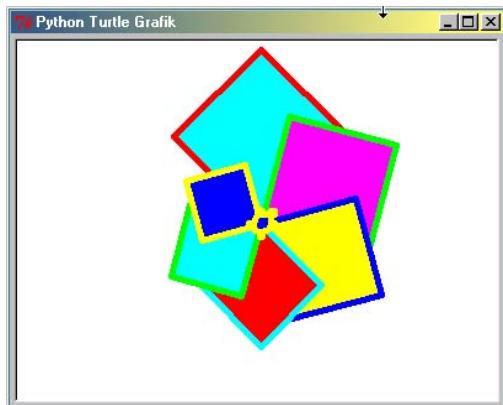
Unser Code beginnt nun so, wie in der langen Abbildung ganz am Rand auf der nächsten Seite. →

Übung: Verschieden große Quadrate



⇒ Führe alle nötigen Änderungen für das Programm aus, teste es und speichere eine Kopie als quadrat05.py.

Das Ergebnis sollte nun so aussehen:



```
Python Turtle Grafik
File Edit Format Run Options
Windows Help
# Python für Kids --
# Autor: Gregor Lingl
# Datum: 5. 8. 2009
# quadrat_arbeit.py:
#           und v

from turtle import *

reset()
pensize(5)

right(45)

seite = 100
pencolor("red")
fillcolor("cyan")
begin_fill()
forward(seite)
left(90)
forward(seite)
left(90)
forward(seite)
left(90)
forward(seite)
left(90)
end_fill()

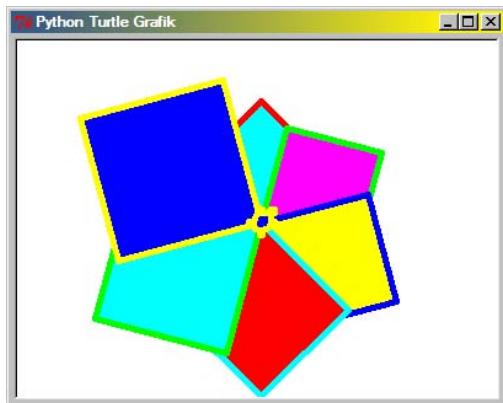
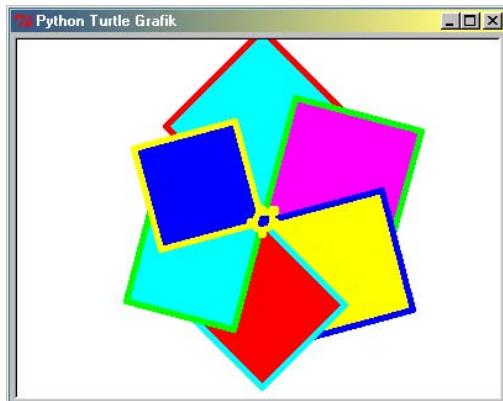
right(60)
seite = seite - 10

pencolor("green")
fillcolor("magenta")
begin_fill()
forward(seite)
left(90)
forward(seite)
left(90)
forward(seite)
left(90)
forward(seite)
left(90)
end_fill()

right(60)
seite = seite - 10

pencolor("blue")
fillcolor("yellow")
Ln: 80 Col: 17
```

Oder hätte dir eines von diesen beiden besser gefallen?



Du wirst staunen: Durch Einführung von zwei Namen können alle diese Zeichnungen mit ganz wenigen Programmänderungen erzeugt werden.

3



Im Programm kannst du das so bewerkstelligen:

- Gleich unter die import-Anweisung schreibst du zwei Zuweisungen (die Zahlen sind die für das linke Bild oben):

```
startseite = 110
```

```
aenderung = -5
```

- Vor das erste Quadrat schreibst du:

```
seite = startseite
```

- Vor alle anderen Quadrate schreibst du anstelle von

```
seite = seite - 10:
```

```
seite = seite + aenderung
```

- Wenn das Programm fertiggestellt ist und richtig funktioniert, speichere eine Kopie davon als quadrat06.py ab!

Du solltest nun noch ein wenig mit quadrat_arbeit.py experimentieren. Dabei wird es dir vielleicht mit der Zeit auf die Nerven gehen, dass sich die Turtle so langsam bewegt.

Im turtle-Modul gibt es eine Funktion, mit der du die Geschwindigkeit der Turtle-Animation steuern kannst. Ihr Name ist speed(), und sie verlangt eine ganze Zahl im Bereich von 0 bis 10 als Argument.

Die Voreinstellung der Turtle-Geschwindigkeit ist speed(3). Auf diesen Wert wird die Geschwindigkeit auch mit reset() zurückgestellt.

speed(1) stellt die Turtle auf die langsamste Geschwindigkeit.

speed(10) stellt die Turtle auf die schnellste Geschwindigkeit.

speed(0) stellt die Turtlebewegung überhaupt ab – die Turtle springt dann vom Anfangs- zum Endpunkt und der Programmablauf ist damit *am schnellsten*, aber die Bewegung kann nicht mehr gut verfolgt werden.

Wähle stets eine langsame Turtle-Geschwindigkeit, wenn du durch Beobachtung der Turtle versuchst, einen Programmablauf zu durchschauen oder einen Programmfehler zu finden.

Wähle eine schnelle Turtle-Bewegung, wenn du mit einem fertigen Programm experimentierst.

- Füge in quadrat_arbeit.py gleich nach der reset()-Anweisung die Anweisung speed(10) ein. Führe das Programm aus. Vergleiche den Programmablauf auch mit der Variante speed(0).



Und nun zu etwas ganz anderem

- » Ändere ein paar Mal die Werte von startseite und aenderung ab und betrachte die unterschiedlichen Ergebnisse! Beachte den Unterschied zwischen positiven und negativen Werten von aenderung! Was geschieht, wenn aenderung gleich 0 gesetzt wird?



Und nun zu etwas ganz anderem

Im letzten Abschnitt hast du gesehen, dass man Dingen Namen geben kann. Als Dinge hatten wir aber immer bloß Zahlenwerte verwendet, die für unsere Zeichnung wichtig waren.

Jetzt wollen wir eine kleine Pause von der Grafik machen und ein ganz einfaches Dialogprogramm, dialog01.py, schreiben. Dieses soll etwa so ablaufen:

```
*IPI Shell / Turtle Grafik*
File Edit Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>>
Hallo! Schönen Tag heute!
Wie heißt du? |
```

Hier muss der Benutzer seinen Namen eingeben. Der Benutzer deines Programms kann also erstmals nicht nur andächtig dem Ablauf zuschauen, sondern muss in den Programmablauf eingreifen.

```
*IPI Shell / Turtle Grafik*
File Edit Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>>
Hallo! Schönen Tag heute!
Wie heißt du? Buffi|
```

Nachdem er auf gedrückt hat, läuft das Programm weiter.

```
*IPI Shell / Turtle Grafik*
File Edit Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>>
Hallo! Schönen Tag heute!
Wie heißt du? Buffi
Fein, dich hier zu sehen, Buffi
Hoffe, du hast Spaß mit mir!
>>> |
```

3



Um das programmieren zu können, brauchen wir etwas Neues: Eine Funktion, die die Zeichen, die der Benutzer oder die Benutzerin auf der Tastatur tippt, im Computer speichert.

Python hat eine solche Funktion eingebaut. Sie heißt `input()` und funktioniert so:

» Starte die IDLE (PYTHON GUI) und mach mit!

```
>>> input()
```

```
|
```

Merkwürdig, nichts geschieht. Der Cursor blinkt. Da fällt uns ein, dass `input()` ja für die Eingabe von Zeichen gedacht ist. Gut, geben wir einen Gruß ein:

```
>>> input()
```

```
Moin moin!
```

```
'Moin moin!'
```

```
>>>
```

The screenshot shows the Python IDLE shell window titled "IPI Shell / Turtle Grafik". The menu bar includes File, Edit, Debug, Options, Windows, Help, and a language selection dropdown. The main window displays the Python interpreter's prompt and output. The user has run the command `>>> input()`, which resulted in the string 'Moin moin!' being printed back to the console. The Python version shown is 3.1.1, build r311:74483, from August 17, 2009.

```
IPI Shell / Turtle Grafik
File Edit Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>> input()
Moin moin!
'Moin moin!'
>>>
```

`input()` ohne Parameter aufgerufen.

Jetzt siehst du, dass `input()` ein Ergebnis erzeugt hat: den String 'Moin moin!'.

Der Python-Interpreter hat es, wie jedes Ergebnis (denke an `sqrt(4)`), gleich ausgegeben.

Falls du Unklarheiten über »Moin« hast (und einen Internet-Zugang), beseitige sie gleich mit Hilfe von <http://de.wikipedia.org/wiki/Moin>.

Ich fasse zusammen, was geschehen ist:

Wir haben die Funktion `input()` aufgerufen. `input()` hat die Zeichen, die der Benutzer eingegeben hat, als String ausgegeben. Der Python-Interpreter hat diese Ausgabe ins IDLE-Fenster geschrieben.

Und nun zu etwas ganz anderem



Wir können mit diesem String leider nichts weiter tun. Wenn wir das aber wollen, dann müssen wir dem Ergebnis von `input()` einen Namen geben:
Dazu verwenden wir – wie schon gehabt – eine Wertzuweisung:

```
>>> gruss = input()  
Moin moin!  
>>> gruss  
'Moin moin!'
```

Der Name `gruss` verweist jetzt auf den String "Moin moin!". Der Name kann weiter verwendet werden:

```
>>> print(gruss, "So eine nette Begrüßung!")  
Moin moin! So eine nette Begrüßung!
```

```
>>>
```

Vielleicht ist dir bei der Eingabe der letzten Anweisung Folgendes aufgefallen:

```
Moin moin!  
'Moin moin!'  
>>> gruss = input()  
           ↑  
           input([prompt]) -> string
```

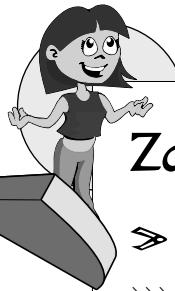
Ein Hinweis, dass in `input()` ein Argument eingesetzt werden kann – aber nicht muss. Man sagt, das Argument ist optional. Das wird durch die eckigen Klammern um das Wort `prompt` ausgedrückt. `prompt` heißt etwa Bereitschaftsanzeige. Außerdem zeigt der Hinweis auch an, dass das Ergebnis des Funktionsaufrufs ein `string` ist. (Nach `sqrt()` und `print()` ist dies nun die dritte Funktion, die du kennen lernst, die ein Ergebnis ausgibt.)

Probieren wir das mit "Hallo?" als Argument für `prompt` aus. Denke daran, dass du `Hi!` oder Ähnliches eingeben musst, wenn `Hallo?` am Bildschirm erscheint:

```
>>> gruss = input("Hallo? ")  
Hallo? Hi!  
>>> print(gruss, "Wie bitte? Nicht gerade gesprächig!")  
Hi! Wie bitte? Nicht gerade gesprächig!
```

➤ Erstelle ein Programm `dialog01.py`, mit dem der Dialog ausgeführt werden kann, der in den drei Screenshots am Anfang dieses Abschnitts dargestellt ist.

3



Zahleneingaben

» Mach mit:

```
>>> knete = input("Knete rüber! ")
Knete rüber! 35
>>> knete
'35'
```

Sieht nicht schlecht aus. Aber knete ist ein String. Kann man damit rechnen? (War da nicht mal was in Kapitel 1?)

```
>>> knete * 12
'353535353535353535353535'
```

Sieht noch besser aus! Ist aber nicht das, was wir erwartet haben! Warum erhältst du dieses Ergebnis? Weil man mit Strings nicht rechnen kann. Nur mit Zahlen kann man rechnen.

In Python gibt es Funktionen, mit denen man Dinge eines bestimmten Typs erzeugen kann. Eine davon heißt `float()` und erzeugt Kommazahlen. Das ist genau das, was wir hier brauchen:

```
>>> float(knete)
35.0
>>> float(knete) * 12
420.0
```

Nach der Eingabe mit `input()` ist der Wert des Namens knete ein String:

```
>>> knete
'35'
```

Wir wollen, dass knete auf die entsprechende Kommazahl verweist:

```
>>> knete = float(knete)
>>> knete
35.0
```

Nun können wir damit rechnen:

```
>>> knete * 12
420.0
```

Beachte, dass man in die Funktion `float()` als Argument nur Dinge einsetzen kann, aus denen man sinnvoll eine Kommazahl erzeugen kann:

```
>>> float(1)
1.0
>>> float("1")
1.0
```

Grafik-Programm mit Dialog

```
>>> float("3.06")
3.06
>>> float(4.99)  # nicht sehr sinnvoll, da 4.99
                 # selbst schon Kommazahl ist.
4.99
>>> float("zwei")
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    float("zwei")
ValueError: could not convert string to float: zwei
>>>
```



Aha! Wörter kann `float()` nicht in Kommazahlen umwandeln!

Jetzt weißt du schon genug, um diesen Programmentwurf zu codieren:

Programmentwurf: Dialog mit Taschengeldberechnung

Ausgabe: "Hallo schönen Tag heute!"
user \Rightarrow Eingabe: "Wie heißt du?"
Ausgabe: "Fein, dich hier zu sehen," user
geld \Rightarrow Eingabe: "Wie viel Euro Taschengeld bekommst du monatlich?"
geld \Rightarrow Kommazahl(geld)
geldProJahr \Rightarrow geld * 12
Ausgabe: "Wow! Das sind ja" geldProJahr "€ im Jahr!"

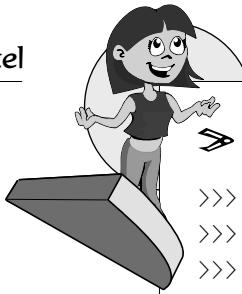
Aufgabe: Codiere diesen Programmentwurf in Python, speichere das Programm als `dialog02.py` ab und teste es, bis es korrekt läuft!

Grafik-Programm mit Dialog

Wir wollen jetzt noch unser Quadrat-Programm so erweitern, dass es vor der Zeichnung den Benutzer fragt, wie lang die erste Quadratseite sein soll und um wie viel sie sich in einem Schritt ändern soll.

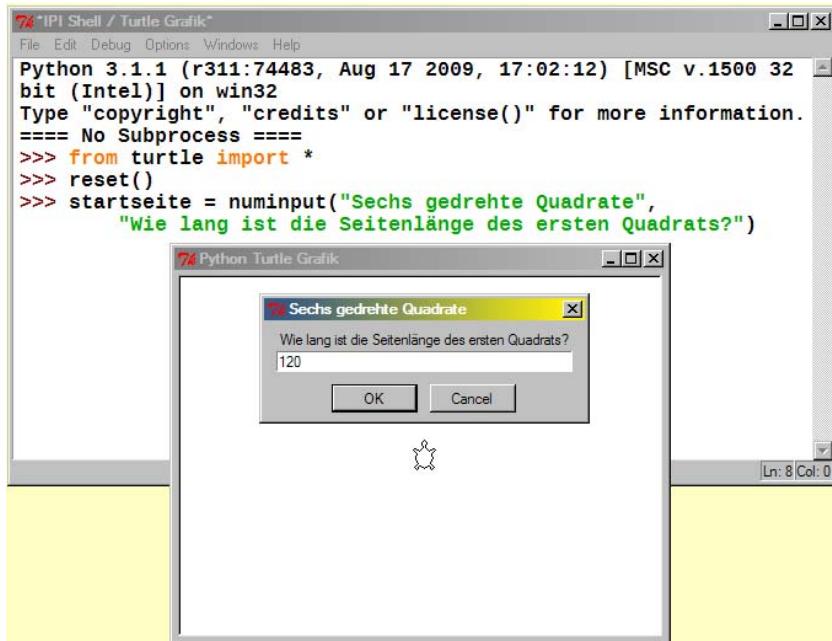
Wir brauchen dazu Dialoge, mit denen wir den Namen `startseite` und `aenderung` Zahlenwerte zuweisen können. Das Turtle-Modul stellt dafür einen grafischen Eingabedialog `numinput()` bereit. Probieren wir den gleich aus:

3



➤ Mach mit:

```
>>> from turtle import *
>>> reset()
>>> startseite = numinput("6 gedrehte Quadrate",
    "Wie lang ist die erste Seite?")
```



`numinput()` benötigt zwei Strings als Argumente: Der erste ist der Titel des Popup-Fensters, der zweite ein »Prompt«, wie wir es schon mehrmals gehabt haben. Der erscheint als Text im Fenster.

Du kannst nun eine Zahl eingeben und auf den OK-Knopf drücken! Das Eingabedialogfenster geht zu. Was ist nun der Wert von `startseite`?

```
>>> startseite
120.0
```

Du siehst, der Eingabedialog hat eine Kommazahl geliefert. Somit ersparen wir uns die Umwandlung von einem String in eine Zahl.

Hast du Sorge, dass die Frage für den Benutzer des Programms vielleicht nicht klar genug gestellt ist? Was könnten wir dagegen machen? Eine etwas längere Erklärung? Fragen wir Clara!



Clara Pythias Python-Special

In Python kann man auch mehrzeilige Strings verwenden: Man muss dazu die Strings nur beiderseitig mit einer Folge von drei Anführungszeichen begrenzen.

➤ Mach mit:

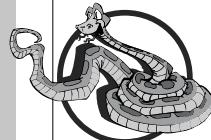
```
>>> mehrzeilig = """Ein String mit  
zwei Zeilen."""
```

Wenn du am Ende der ersten Zeile auf drückst, geht die Schreibmarke einfach nur in die nächste Zeile – du kannst in der nächsten, übernächsten usw. Zeile weiterschreiben, so lange, bis du den String mit drei weiteren Anführungszeichen abschließt. Danach kann er ebenso mehrzeilig, wie er eingegeben wurde, ausgegeben werden:

The screenshot shows a Python IDLE window titled "IPI Shell / Turtle Grafik". The code in the shell is as follows:

```
>>> zweizeilig = """Ein String  
mit zwei Zeilen!"""  
>>> print(zweizeilig)  
Ein String  
mit zwei Zeilen!  
>>> print("""Oder auch mehr:  
a  
be  
bu  
und  
raus  
bist du!""")  
Oder auch mehr:  
a  
be  
bu  
und  
raus  
bist du!  
>>> |
```

The output of the print statements is displayed below the shell. A vertical scroll bar is visible on the right side of the window. The status bar at the bottom shows "Ln: 23 Col: 4".

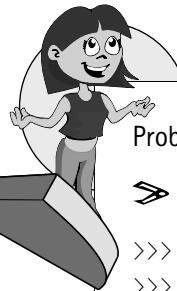


Gruppen von drei Anführungszeichen können mehrzeilige Strings begrenzen.

Mit dieser Technik lassen sich oft viele print-Anweisungen mit herkömmlichen Strings durch eine einzige print-Anweisung ersetzen.

Aber auch für das prompt-Argument in numinput() kannst du mehrzeilige Strings verwenden.

3



Probieren wir das auch noch rasch aus:

➤ Mach mit:

```
>>> from turtle import *
>>> reset()
>>> prompt = """Dieses Programm zeichnet sechs
gegeneinander verdrehte Quadrate.
```

Wie lang ist die Seitenlänge des ersten Quadrats?"""

```
>>> startseite = numinput("6 gedrehte Quadrate", prompt)
```

The screenshot shows the IPI Shell/Turtle Grafik interface. The shell window displays Python code and its output. A numeric input dialog box is overlaid on the shell window, asking for the side length of the first square. The user has entered '111'.

```

IPI Shell / Turtle Grafik
File Edit Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32
bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>> from turtle import *
>>> reset()
>>> prompt = """Dieses Programm zeichnet sechs
gegeneinander verdrehte Quadrate.

Wie lang ist die Seitenlänge des ersten Quadrats?""""
>>> startseite = numinput("6 gedrehte Quadrate", prompt)

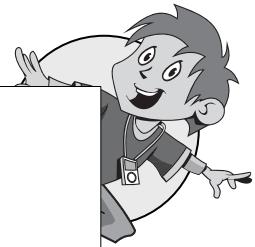
```

Nun kannst du daran gehen, für die `startseite` und die änderung numerische Eingabedialoge in das Programm `quadrat_arbeit.py` einzubauen.

- Starte IPI-TURTLEGRAFIK und öffne `quadrat_arbeit.py` in einem Editor-Fenster.
- Füge Eingabedialoge mit schön erdachten Titeln und Prompts als ersten Programmteil ein. Sie sollten so funktionieren, wie wir das eben interaktiv ausprobiert haben.

Noch ein Schmankerl gefällig? Das Turtle-Modul hat auch eine Funktion `title()`, mit der man den Titel des Grafik-Fensters einstellen kann. Damit sieht dein Programm schon fast professionell aus:

Zusammenfassung



» Mach mit:

```
>>> from turtle import *
>>> reset()
>>> title("Sechs gegeneinander gedrehte farbige Quadrate.")
```

A screenshot of the Python IDLE shell. The title bar of the window says "Sechs gegeneinander gedrehte farbige Quadrate.". The code in the shell window is:

```
74 IPI Shell / Turtle Grafik
File Edit Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32
bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>> from turtle import *
>>> reset()
>>> title("Sechs gegeneinander gedrehte farbige Quadrate.")
>>>
```

The status bar at the bottom right shows "Ln: 7 Col: 4".

Mit `title()` kann man dem Grafikfenster einen passenden Titel verpassen.

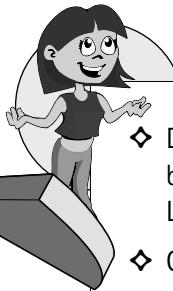
- » Füge gleich am Anfang des Programms eine Anweisung zur Gestaltung des Titels des Grafik-Fensters ein.
- » Speichere eine Kopie deines Programms als `quadrat07.py` ab.

Zusammenfassung

In diesem Kapitel hast du Folgendes erfahren:

- ❖ Python arbeitet mit verschiedenen Dingen: Zahlenwerten, Zeichenketten, Funktionen.
- ❖ Du kannst mit diesen Dingen arbeiten, indem du ihnen Namen gibst.
- ❖ Dinge nennt man in der Computer-Fachsprache besser Objekte.
- ❖ Man sagt: Namen verweisen auf Objekte.
- ❖ Man sagt auch: Der Wert eines Namens ist ein bestimmtes Objekt.
- ❖ Die Python-Anweisung, die einem Namen ein Objekt zuordnet, heißt Zuweisung oder ausführliche Wertzuweisung.
- ❖ Während des Ablaufs eines Programms können die Werte von Namen geändert werden. Daher spricht man auch von Variablen.
- ❖ Namen können in Ausdrücken und Anweisungen wie konstante Werte verwendet werden.

3



- ❖ Die eingebaute Python-Funktion `input()` ermöglicht Benutzereingaben über die Tastatur. Sie liefert immer einen String, allenfalls einen Leerstring.
- ❖ Geeignete Strings können mit der Funktion `float()` in Zahlen umgewandelt werden.
- ❖ Durch Benutzereingaben kann der Ablauf von Programmen beeinflusst werden.
- ❖ Beidseitig mit drei Anführungszeichen begrenzte Strings können sich über mehrere Zeilen erstrecken.
- ❖ Turtle-Grafik: Mit der Funktion `speed()` kannst du die Bewegungsgeschwindigkeit der Turtle einstellen.
- ❖ Das Modul `turtle` stellt einen graphischen Eingabe-Dialog für Zahleneingaben bereit: `numinput()`. Er verlangt zwei Strings als Argumente: den Dialogfenster-Titel und einen Prompt-Text. Er gibt einen Zahlenwert zurück.
- ❖ Das Modul `turtle` hat eine Funktion `title()`. Als einziges Argument verlangt sie einen String, den gewünschten Titel des Grafik-Fensters.

Einige Aufgaben ...

Aufgabe 1: Schreibe ein Programm, das folgende Ausgabe erzeugt (Benutzereingaben sind *kursiv* geschrieben):

```
Notendurchschnittsberechnung:  
=====
```

Name: *Angela*

Gegenstand: *Italienisch*

Noten für

1. Klassenarbeit 2

2. Klassenarbeit 3

3. Klassenarbeit 1

Der Notendurchschnitt von Angela in Italienisch ist 2.0

Mache dir einen Programmentwurf und bedenke, dass du sechs Variablennamen erfinden musst! Für den Namen, den Gegenstand, die Noten – nimm dafür vielleicht `note1`, `note2`, `note3` – und den Durchschnitt.

Wie man den berechnet? So:

```
durchschnitt = (note1 + note2 + note3) / 3
```

... und einige Fragen:

Aufgabe 2, eine Forschungsaufgabe für einen Kreativitätsanfall:

In unseren Quadrat-Programmen wurde jeweils die Quadratseite von Quadrat zu Quadrat um einen bestimmten Betrag verändert.

Andere Bilder ergeben sich, wenn man die Seite stets mit einem Faktor multipliziert. Man ersetzt

seite = seite + aenderung

durch

seite = seite * faktor

Ändere das Programm in dieser Weise ab und spiele damit ein wenig. Vergleiche die jetzt entstehenden Grafiken mit den früher erzeugten. Welche Werte für faktor ergeben schöne Bilder?

... und einige Fragen:

1. Was ist der Unterschied zwischen

>>> 3 * 1.8

und

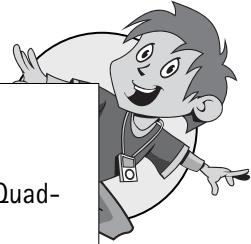
>>> print(3 * 1.8)?

2. Wovon hängt es ab, ob folgende Anweisung funktioniert?

taschengeld = taschengeld + taschengeld

Und welche Wirkung hat sie, wenn sie funktioniert?

3. Um wie viel Grad muss sich die Turtle nach jedem forward() -Schritt drehen, wenn sie ein regelmäßiges Fünfeck zeichnen soll?



4



Wir erzeugen unsere eigenen Funktionen

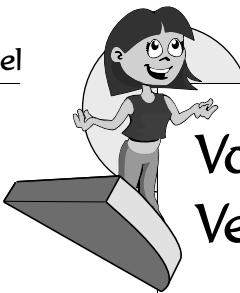
In den ersten Kapiteln hast du schon eine ganze Anzahl von Programmen geschrieben. Dabei wird dir aufgefallen sein, dass die Erstellung mancher Grafiken schon recht lange Programme erfordert hat, in denen sich manche Abschnitte mehrmals wiederholt haben.

Du wirst nicht überrascht sein zu erfahren, dass es verschiedene Techniken gibt, lange und eintönige Programmtexte kürzer zu formulieren. In den nächsten Kapiteln wird es hauptsächlich um diese Techniken gehen.

In diesem Kapitel lernst du ...

- ◎ wie man Programmteile zu selbst geschriebenen Funktionen zusammenfasst und warum das sinnvoll ist.
- ◎ was globale und was lokale Variablen sind und warum man sie bei der Arbeit mit Funktionen unterscheiden muss.
- ◎ wie du Entscheidungen programmieren kannst.

4



Vorbereitung – eine kleine Vereinfachung

In diesem Abschnitt gehen wir vom Programm dreieck05.py aus dem vorigen Kapitel aus. Lade es in ein Editor-Fenster und speichere es in C:\py4kids\kap04\ unter dem Namen dreieck_arbeit.py.

In diesem Programm kommen dreimal zwei Zeilen der Form

```
pencolor("red")
fillcolor("cyan")
```

vor. Im Modul turtle gibt es die Funktion color(), die es gestattet, diese beiden Anweisungen in einer kürzeren zusammenzufassen:

```
color("red", "cyan")
```

➤ Mach mit – im IPI-SHELL-Fenster:

```
>>> from turtle import *
>>> pensize(10)
>>> color("red", "blue")
```

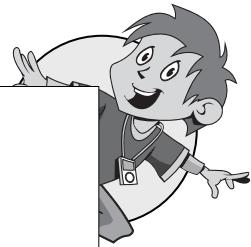
Nun kannst du an der Färbung der Turtle erkennen: Die Farbe, die im color()-Aufruf als erstes Argument eingesetzt wird, ist die Stiftfarbe, und die andere die Füllfarbe. (Wenn du's nicht glaubst, dann probiere es aus: Zeichne ein gefülltes Dreieck. ;-)

➤ Ersetze im Programm dreieck_arbeit.py dreimal die beiden Zeilen mit den Farbeinstellungen durch jeweils eine Zeile mit einem Aufruf der Funktion color().

Außerdem erkennst du, dass sich in diesem Programm die Dreiecksseite – ausgehend von der startseite 65 – von Dreieck zu Dreieck auch um denselben Wert ändert, nämlich um 35. Wir können daher unser Verfahren mit startseite und der aenderung vom Programm quadrat06.py ebenfalls anwenden.

➤ Führe diese nun auch in dreieck_arbeit.py durch.

➤ Führe das Programm aus und überzeuge dich, dass es die gleiche Ausgabe erzeugt wie vorher.



Wir (er)finden die Funktion dreieck

Nach diesen Abänderungen hat unser Programm `dreieck_arbeit.py` den folgenden Code:

```
from turtle import *

startseite = 65
aenderung = 35

reset()
pensize(10)
right(90)

seite = startseite

color("red", "cyan")
begin_fill()
forward(seite)
left(120)
forward(seite)
left(120)
forward(seite)
left(120)
end_fill()

left(120)
seite = seite + aenderung

color("green", "magenta")
begin_fill()
forward(seite)
left(120)
forward(seite)
left(120)
forward(seite)
left(120)
end_fill()
```

4



```

left(120)
seite = seite + aenderung

color("blue", "yellow")
begin_fill()
forward(seite)
left(120)
forward(seite)
left(120)
forward(seite)
left(120)
end_fill()

left(120)
hideturtle()

```

An diesem Code wird dir auffallen, dass darin jene sechs Zeilen, die ein gleichseitiges Dreieck zeichnen, dreimal vorkommen.

Im Direktmodus Funktionen definieren

In Python kannst du eine Folge von Anweisungen zu einer so genannten Funktion zusammenfassen. Wir verwenden weiter den IPI, um zu sehen, wie man das macht und warum das nützlich ist.

➤ Mach mit (turtle ist schon importiert)!

```

>>> showturtle(); clear(); home()
>>> seite = 135
>>> right(90)

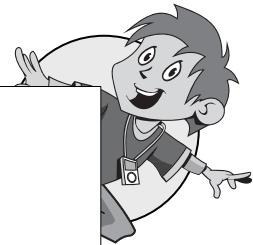
```

➤ Nun kommt etwas ganz Neues, nämlich die Definition einer Funktion.

Wir denken uns für sie einen Namen aus: Nahe liegend ist der Name `drei`. Tippe nun Folgendes ein und achte dabei auf zwei Dinge:

1. Die erste Zeile der Funktionsdefinition muss mit *einem Paar runder Klammern und einem Doppelpunkt* abgeschlossen werden. Diese Zeile nennt man den Kopf der Funktionsdefinition.
2. Die übrigen sechs Zeilen der Funktionsdefinition müssen alle um gleich viele Stellen nach rechts eingerückt werden und stehen dann schön aufgereiht untereinander. Der IPI macht das ganz automatisch für dich – du darfst nur nichts an dieser Einrückung ändern. Diese Folge von sechs Anweisungen nennt man den Körper der Funktionsdefinition.

Im Direktmodus Funktionen definieren



```
>>> def dreieck():
    forward(seite)
    left(120)
    forward(seite)
    left(120)
    forward(seite)
    left(120)
```

>>>

Nach der letzten Zeile drückst du einfach nochmals auf – und diesmal passiert, wenn du keinen Tippfehler im Funktionstext gemacht hast, ungewohnterweise nichts. Es erscheint nur unser bekanntes Bereitschaftszeichen für die Eingabe.

Wie sollen wir das verstehen! So viel Mühe und dann nichts!

➤ Gib doch einmal ein:

```
>>> dreieck()
```

Na bitte! Das Turtle-Grafik-Fenster geht auf und die Turtle zeichnet uns ein Dreieck! Wenn das kein Erfolg ist! Wir probieren gleich weiter aus:

```
>>> left(120)
>>> seite=100
>>> dreieck()
>>> left(120)
>>> seite=65
>>> dreieck()
```

Es gibt jetzt also eine neue Funktion: `dreieck()`.

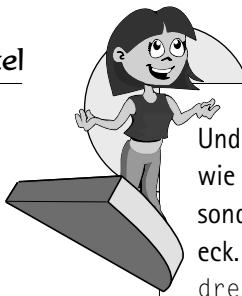
```
>>> dreieck
<function dreieck at 0x00ADFB10>
>>>
```

Beachte hier den Unterschied von `dreieck()` und `dreieck`.

`dreieck()` ist ein *Aufruf* der Funktion `dreieck`. Darauf antwortet der IPI mit der Ausführung der Funktion.

`dreieck` ist das gewöhnliche Anschreiben eines Namens. Darauf antwortet der IPI – wie immer – mit der Ausgabe einer Beschreibung des Objekts, auf das der Name verweist. Hier ist dieses Objekt eben unsere neu definierte Funktion.





Und die macht nicht, was Programmierer vor uns bereits festgelegt haben, wie zum Beispiel der Programmierer des turtle-Moduls mit `penup()`, sondern die macht, was *wir selbst* festgelegt haben: Sie zeichnet ein Dreieck. Wir können also nun mit einer einzigen Python-Anweisung, `dreieck()`, dem Aufruf der Funktion `dreieck()`, ein Dreieck zeichnen.

Nachträglich müssen wir also sagen: An dem Punkt, an dem uns der IPI nach der Eingabe der Funktionsdefinition angeschwiegen hat, hat er in Wirklichkeit viel getan: Er hat »gelernt«, wie er ein Dreieck zeichnen kann.

Oder in Python-Speak: Der IPI hat eine `def`-Anweisung ausgeführt. An der orangen Färbung hast du wahrscheinlich schon erkannt, dass das Wort `def`, mit dem die Funktionsdefinition eingeleitet wird, ein reserviertes Wort ist.

Dreieck-Programm, heute neu

```

# dreieck_arbeit.py - C:/py4kids/kap0...
File Edit Format Run Options Windows Help
# dreieck_arbeit.py: 3 gle
from turtle import *
def dreieck():
    """Zeichne Dreieck mit
    ....
    forward(seite)
    left(120)
    forward(seite)
    left(120)
    forward(seite)
    left(120)

startseite = 65
aenderung = 35

reset()
pensize(10)
right(90)

seite = startseite
color("red", "cyan")
begin_fill()
dreieck()
end_fill()

left(120)
seite = seite + aenderung
color("green", "magenta")
begin_fill()
dreieck()
end_fill()

left(120)
seite = seite + aenderung
color("blue", "yellow")
begin_fill()
dreieck()
end_fill()

left(120)
hideturtle()

```

Ich zeige dir jetzt, wie wir die oben beschriebene Funktionsdefinition in unserem Dreiecksprogramm benutzen und es dabei gleichzeitig kürzer und klarer machen können.

- Wir arbeiten weiter mit `dreieck_arbeit.py` in einem Editor-Fenster.
- Nach der `import`-Anweisung schreibst du unsere Definition der Funktion `dreieck()` hinein. Der Funktionskörper rückt automatisch um vier Zeichen ein.
- Nun kannst du dreimal die jeweils sechs Zeilen im Programm, die das Dreieck erzeugen, durch eine einzige ersetzen, nämlich durch `dreieck()`.

Du wirst bald feststellen, dass das Schreiben von Python-Programmen zum Großteil aus dem Schreiben von Funktionsdefinitionen besteht. Da ist es sehr zweckmäßig festzuhalten, was man mit einer bestimmten Funktion bezweckt. Das macht man mit so genannten Dokumentations-Strings (kurz Doc-Strings), und zwar so:

Wie wird »dreieck07.py« ausgeführt?



```
def dreieck():
    """Zeichne Dreieck mit Seitenlänge seite.
    """
    forward(seite)
    left(120)
    forward(seite)
    left(120)
    forward(seite)
    left(120)
```

Der Doc-String steht gleich unter dem Funktionskopf und ist zweckmäßigweise ein String mit dreifachen Anführungszeichen, damit er, falls nötig, auch mehrere Zeilen umfassen kann.

➤ Füge den Dokumentations-String wie oben gezeigt in die Funktionsdefinition für `dreieck()` ein.

So, und was hast du davon? Wenn du das Programm liest, kannst du dir rasch einen Überblick verschaffen, was die einzelnen Funktionen tun.

➤ Führe das Programm aus und überzeuge dich, dass es keine Fehler enthält!

Du kannst nun auch im IPI die Funktion `dreieck()` direkt ausführen. Dabei bemerkst du Folgendes:

```
>>> dreieck()
O
Zeichne Dreieck mit Seitenlänge seite.
```

Die erste Zeile des Doc-Strings einer Funktion erscheint im »Call-Tipp«.

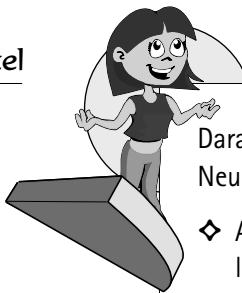
Die erste Zeile des Doc-Strings erscheint auch im so genannten »Call-Tipp«, der Hilfe, die dir Python gibt, wenn du einen Funktionsaufruf schreibst. Nützlich!

➤ Speichere eine Kopie des Programms `dreieck_arbeit.py` unter dem Namen `dreieck07.py` im Ordner `C:\py4kids\kap04\` ab.

Wie wird »dreieck07.py« ausgeführt?

Die Ausführung unserer bisherigen Python-Programme zu verstehen, war eigentlich ganz leicht. Es wurde eine Anweisung nach der anderen ausgeführt, wie wenn du sie der Reihe nach dem IPI eingegeben hättest.

4



Daran hat sich jetzt mit dem Programm dreieck07.py nichts geändert.
Neu ist nur:

- ❖ Als zweite Anweisung wird eine def-Anweisung ausgeführt. Damit lernt Python ein neues Wort. Es lernt, *wie* die Anweisung dreieck() auszuführen ist, *ohne sie tatsächlich* an dieser Stelle *auszuführen*.
- ❖ An drei Stellen wird die Funktion dreieck() aufgerufen: Um sie auszuführen, »springt« die Programmausführung zu den Anweisungen ganz am Anfang des Programms, die in der Funktionsdefinition für dreieck() stehen, und führt diese der Reihe nach aus. Ein Dreieck wird gezeichnet. Dann springt sie wieder zurück, genau hinter die Stelle, an der der Aufruf von dreieck() steht, und fährt dort mit der Ausführung der weiteren Anweisungen fort.
- ❖ *Wichtig!* Ein Punkt ist hier besonders zu beachten: Vor jedem Aufruf von dreieck() muss sichergestellt sein, dass der Name seite auf den gewünschten Wert verweist. Du erreichst das, indem du vor jedem Aufruf von dreieck() dem Namen seite diesen Wert zuweist.

Im nächsten Kapitel wirst du sehen, dass es dafür eine noch elegantere Lösung gibt.

Noch ein Schritt weiter ...

Mit der Definition von dreieck() wird der Funktionsaufruf dreieck() zu einer vollwertigen Python-Anweisung, so wie jeder andere Funktionsaufruf, z. B. penup() oder forward(50) auch. Das heißt aber, dass man ihn auch in anderen Funktionsdefinitionen verwenden darf. Das wollen wir hier noch probieren. (Wenn es dir zu schwierig wird, überspringe diesen kurzen Abschnitt und kehre vielleicht später nochmals hierher zurück.)

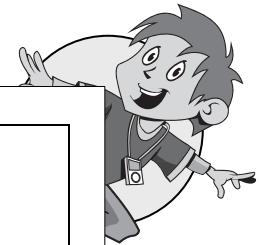
Im Programm dreieck.py kommt eine Folge von drei Anweisungen drei Mal vor, nämlich:

```
begin_fill()  
dreieck()  
end_fill()
```

Diese drei Anweisungen zeichnen ein gefülltes Dreieck. Daraus machen wir jetzt eine neue Funktion.

- Schreibe die folgende Funktionsdefinition in das Programm dreieck_arbeit.py direkt unter die Funktionsdefinition von dreieck():

Noch ein Schritt weiter ...



```
def fuelle_dreieck():
    begin_fill()
    dreieck()
    end_fill()
```

In dieser neuen Funktion kommt im Funktionskörper ein Aufruf der von uns selbst definierten Funktion `dreieck()` vor. Kein Problem!

- Ersetze drei Mal die jeweils drei Zeilen durch den Funktionsaufruf:

```
fuelle_dreieck()
```

Nun sieht das Programm so aus:

- Speichere eine Kopie von `dreieck_arbeit.py` unter dem Namen `dreieck08.py` ab.

Die Funktionen, die wir bisher definiert haben, werden ohne Argumente aufgerufen – also ohne etwas zwischen die runden Klammern zu schreiben – genauso wie `penup()`. Man sagt: Solche Funktionen haben keine Parameter. Über dieses neue Fremdwort wirst du bald mehr erfahren.

```
# dreieck_arbeit.py - C:/py4kids/kap04/dreieck_arbeit.py
File Edit Format Run Options Windows Help
# dreieck_arbeit.py: 3 gleichseitige 1
from turtle import *
def dreieck():
    """Zeichne Dreieck mit Seitenlänge
    """
    forward(seite)
    left(120)
    forward(seite)
    left(120)
    forward(seite)
    left(120)

def fuelle_dreieck():
    """Zeichne gefülltes Dreieck
    mir Seitenlänge seite.
    """
    begin_fill()
    dreieck()
    end_fill()

startseite = 65
aenderung = 35

reset()
pensize(10)
right(90)

seite = startseite
color("red", "cyan")
fuelle_dreieck()

left(120)
seite = seite + aenderung
color("green", "magenta")
fuelle_dreieck()

left(120)
seite = seite + aenderung
color("blue", "yellow")
fuelle_dreieck()

left(120)
hideturtle()

Ln: 51 Col: 12
```

Zusammenfassend hier nochmals das Muster einer Funktionsdefinition:

Muster 3: Definition einer Funktion ohne Parameter

```
def Funktionsname():
    """...Beschreibung...
    """
    Anweisung 1
    Anweisung 2
    ...

```

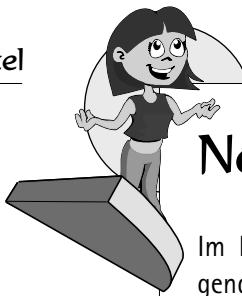
Funktionskopf

Doc-String

Funktionskörper

Alle Anweisungen des Funktionskörpers gleich weit eingerückt

4



Noch eine Idee ...

Im Programmcode von `dreieck_arbeit.py` kommt mehrfach das Folgende vor:

```
fuelle_dreieck()
left(120)
seite = seite + aenderung
```

Hast du da vielleicht die Idee gehabt, die beiden Anweisungen

```
left(120)
seite = seite + aenderung
```

die ja auch für jedes Dreieck gemacht werden müssen, in die Definition der `fuelle_dreieck()`-Funktion hineinzunehmen? (Nach der Zeichnung des letzten Dreiecks würde dann `seite` nochmals geändert – unnötig zwar, hat aber keine sichtbare Wirkung.) Das sähe dann so aus:

```
def fuelle_dreieck():
    """zeichne gefülltes Dreieck
    mit Seitenlänge seite.
    """
    begin_fill()
    dreieck()
    end_fill()
    seite = seite + aenderung
    left(120) # Seite und Richtung
              # fürs nächste Dreieck!
```

Und im Programm stünde dann nur noch:

```
color("red", "cyan")
fuelle_dreieck()

color("green", "magenta")
fuelle_dreieck()

color("blue", "yellow")
fuelle_dreieck()
```

Wirkt schlau! Funktioniert aber nicht! Auf der CD findest du dieses Programm im Ordner `py4kids\kap04` unter dem Namen `dreieck08a.py`.

Noch eine Idee ...

➤ Lade dreieck08a.py in ein Editor-Fenster. Sieh dir den Code im Editor an und führe das Programm aus.

Es erzeugt folgende Fehlermeldung:

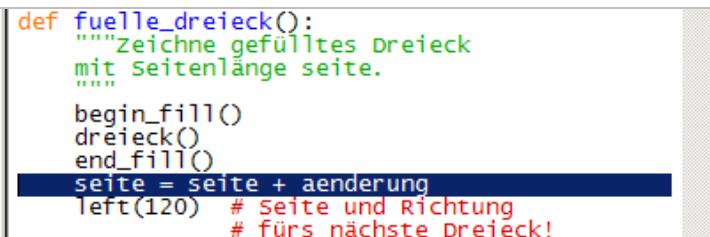


```
Traceback (most recent call last):
  File "C:\py4kids\kap04\dreieck08a.py", line 40, in <module>
    fuelle_dreieck()
  File "C:\py4kids\kap04\dreieck08a.py", line 26, in fuelle_dreieck
    seite = seite + aenderung
                                                Go to file/line
UnboundLocalError: local variable 'seite' referenced before assignment
>>>
```

Ln: 8 Col: 47

Mit der rechten Maustaste auf eine Zeilenummer klicken, führt zu dieser Zeile im Programm.

Das klingt chinesisch! Bis auf 'seite'. Ins Deutsche übersetzt steht da, dass die »lokale« Variable seite verwendet wurde, bevor ihr ein Wert zugewiesen wurde – und zwar in Zeile 26 in der Anweisung seite = seite + aenderung. Gehe mit der Maus auf diese Zeilenummer (kann bei dir eine andere sein), klicke mit der rechten Maustaste drauf und dann auf Go To FILE/LINE. Nun wird im Programm die fehlerhafte Zeile markiert.



```
def fuelle_dreieck():
    """Zeichne gefülltes Dreieck
    mit Seitenlänge seite.

    begin_fill()
    dreieck()
    end_fill()
    seite = seite + aenderung
    left(120) # Seite und Richtung
               # fürs nächste Dreieck!
```

Die Anweisung, die zur Fehlermeldung geführt hat, wurde markiert.

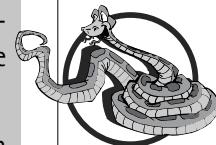
Was ist hier das Problem?

Namen, denen innerhalb einer Funktion ein Wert zugewiesen wird, sind *lokale* Variablen. Um zu verstehen, was das bedeutet, musst du Folgendes wissen:

Clara Pythias Python-Special

Wenn der Python-Interpreter im Code einen Namen antrifft, sucht er das Objekt, auf das der Name verweist. Wo sucht er das? In einem Wörterbuch! Das ist tatsächlich so: Python führt Wörterbücher, in denen die Namen mit ihren Werten verzeichnet sind.

Aus Gründen, die später noch klar werden, haben die Konstrukteure von Python Funktionen so eingerichtet, dass sie ihr eigenes Wörterbuch führen! In diesem »lokalen« Wörterbuch stehen die Namen, denen *im Code der Funktion* Werte zugewiesen werden. Durch die Anweisung seite = seite + aenderung kommt der links stehende Name seite in dieses lokale Wörterbuch. In früheren Versionen von dreieck.py gab es *innerhalb* von Funktionen keine Wertzuweisungen an den Namen seite.



4



Deshalb gab es auch keinen Eintrag ins lokale Wörterbuch. Es gilt folgende Regel: Alle Namen in Funktionskörpern, die links von = stehen, kommen ins lokale Wörterbuch.

Wenn nun der Code der Funktion ausgeführt wird, trifft der Python-Interpreter auf die Anweisung `seite = seite + aenderung` und hat nun zuerst die Addition auszuführen. Dazu sucht er den Wert von `seite`. Er findet, dass `seite` im *lokalen* Wörterbuch vorkommt – aber `seite` hat noch keinen Wert – und daher kann der Interpreter die Anweisung nicht ausführen –, und gibt eine Fehlermeldung aus.

Wir wissen, dass der Name `seite` außerhalb von `fuelle_dreieck()` bereits einen Wert hat. Denn wenn `fuelle_dreieck()` zum ersten Mal aufgerufen wird (Zeile 40, siehe obige Fehlermeldung!), ist die erste Zuweisung eines Wertes an `seite` schon erfolgt (Zeile 37).

Wo stehen die Variablennamen, die in einem Script außerhalb von Funktionen erzeugt werden? Sie stehen im *globalen* Wörterbuch. Dieses wird bei der Ausführung eines Scripts von Python sofort angelegt. Die Funktion `dreieck()` verwendet den Wert dieser globalen Variablen `seite`, denn sie hat keine *lokale* Variable namens `seite`.

Wenn wir unserer Funktion `fuelle_dreieck()` mitteilen könnten, dass mit `seite` ein Name aus dem globalen Wörterbuch gemeint ist, wäre dann das Problem gelöst?

- ⇒ Gehe ins Editor-Fenster mit `dreieck08a.py` und speichere die Datei unter `dreieck_arbeit.py` ab.
- ⇒ Füge als neue erste Zeile der Funktion `fuelle_dreieck()` (nach dem Doc-String) folgende Anweisung ein:

```
def fuelle_dreieck():
    """Zeichne gefülltes Dreieck
    mit Seitenlänge seite.
    """

    global seite
    begin_fill()
    dreieck()
    end_fill()
    seite = seite + aenderung
    left(120) # Seite und Richtung
              # fürs nächste Dreieck!
```

Welche ist die bessere Variante?

Dir wird auffallen, dass `global orange` eingefärbt erscheint – ein neues reserviertes Wort!

➤ Speichere das Programm und führe es aus!

Nun funktioniert es! Also halten wir fest:

Wenn innerhalb einer Funktion eine Zuweisung an einen Variablenamen vorkommt, der außerhalb der Funktion definiert wurde, dann muss diese Variable in der Funktion als `global` deklariert werden. Das geschieht mit der `global`-Anweisung:

`global varname`

die als erste Anweisung im Funktionskörper stehen soll.

Das Wörterbuch für lokale Variablenamen wird von Python zu Beginn der Ausführung einer Funktion erstellt, während der Ausführung der Funktion benutzt und danach gleich wieder gelöscht. Anweisungen außerhalb einer Funktion können auf lokale Namen der Funktion nicht zugreifen!

Der Hauptgrund dafür ist, dass man bei Verwendung von mehreren Funktionen manchmal die gleichen Namen verwenden möchte, ohne Gefahr zu laufen, dass diese einander in die Quere kommen.

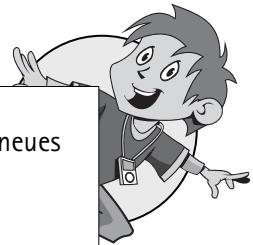
`global varname` stellt sicher, dass der Name `varname` im globalen Wörterbuch gesucht wird.

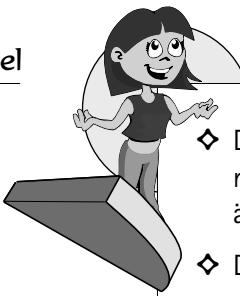
➤ Aktualisiere noch den Kopfkommentar und speichere eine Kopie des Programms unter dem Namen `quadrat08b.py`.

Welche ist die bessere Variante?

Beide Programme `dreieck08.py` und `dreieck08b.py` funktionieren richtig. Doch das erste stellt eindeutig die bessere Variante dar. Das ist diejenige, in der die Funktion `fuelle_dreieck()` nur die drei Anweisungen enthält, die das gefüllte Dreieck zeichnen. Dafür sprechen folgende Gründe:

❖ Der Name der Funktion beschreibt, was die Funktion tut. Nämlich: Ein gefülltes Dreieck zeichnen.





- ❖ Die Funktion `fuelle_dreieck()` kann damit leichter auch zum Zeichnen anderer Dreiecke herangezogen werden, wo etwa `seite` nicht geändert werden soll.
- ❖ Die Turtle zeigt vor und nach dem Funktionsaufruf in dieselbe Richtung.
- ❖ Dies wiegt bei weitem auf, dass im zweiten Programm weniger Anweisungen erforderlich sind. Du wirst aber in Kapitel 6 lernen, wie man sich die zwei Zeilen doch noch sparen kann.

Es gilt als Regel: So sparsam wie möglich globale Variablen in Funktionen ändern!

Mini-Quiz

Du hast nun gelernt, dass ein Block von Anweisungen, der in einem Programm immer wieder gebraucht wird, in eine Funktion gepackt werden kann. Ein Aufruf dieser Funktion bewirkt, dass alle Anweisungen des Blocks ausgeführt werden. Das wird doch nicht nur für Turtle-Grafik-Programme nützlich sein?

Wir werden das nun gemeinsam an einer ganz anders gelagerten Fragestellung untersuchen. Wir wollen ein kleines Quizprogramm erstellen. Es könnte etwa folgendermaßen arbeiten. Beachte, dass *kursiv* gedruckte Wörter Benutzereingaben sind und die Antworten natürlich von den Eingaben abhängen müssen.

Hallo! Du kannst hier ein paar Quizfragen beantworten, um dein Wissen zu überprüfen.

Wie heißt du denn? *Eric*

Also viel Glück, *Eric*, es geht los!

Welche Programmiersprache lernst du gerade? *Python*
Richtig!

Mit welchem reservierten Wort beginnt eine Funktionsdefinition? *fun*

Leider falsch!

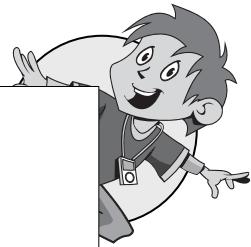
Richtig ist: *def*

Wie viele reservierte Worte hat Python? *101*

Leider falsch!

Richtig ist: 33

Mini-Quiz



Du hast 1 von 3 Punkten erreicht!

Fein, du hast schon einiges gelernt, Eric !

Sieh dir doch mal das Python-Video auf der CD an!

Wenn wir das programmieren wollen, müssen wir zuerst das Problem analysieren und einen Programmennwurf schreiben. Man sieht gleich, dass der Programmablauf drei Phasen hat:

Programmentwurf: MiniQuiz

Begrüßung

Quizfrage 1

Quizfrage 2

Quizfrage 3

Beurteilung

Die Begrüßung scheint ja nichts Neues zu enthalten – so etwas hast du schon gemacht.

Doch bei den Quizfragen tritt etwas Neues auf: Das Programm muss entscheiden können, ob die Frage richtig beantwortet wurde oder nicht. Wenn sie richtig beantwortet wurde, soll das Programm mit "Richtig!" antworten.

Wenn wir für die Frage und die Lösung jeweils die Variablenamen `frage` und `loesung` einführen, können wir diesen Programmteil etwa so beschreiben:

`frage` \Rightarrow Fragentext

`loesung` \Rightarrow Lösungstext

`antwort` \Rightarrow Eingabe(`frage`)

Wenn `antwort` gleich `loesung`:

Ausgabe: »Richtig!«

Was geschieht aber, wenn nicht die richtige Antwort gegeben wurde? Dann gibt das Programm die Worte "Leider falsch!" aus und dazu die richtige Lösung:

4



Wenn antwort nicht gleich loesung:

Ausgabe: »Leider falsch!«

Ausgabe: »Richtig ist:« loesung

Wir haben hier in beiden Fällen dasselbe Muster vor uns: Es wird eine Bedingung abgefragt. Wenn diese Bedingung wahr ist, dann werden ein oder mehrere Anweisungen ausgeführt.

Weil dieses Muster so häufig beim Programmieren gebraucht wird, gibt es dafür eine spezielle Python-Anweisung: die `if`-Anweisung.

Sie heißt so, weil sie mit dem reservierten Wort `if` eingeleitet wird. Die Bedingung, die dann folgt, kann viele verschiedene Formen haben. Hier brauchen wir nur den Vergleich zweier Objekte: Einmal, ob zwei Objekte (hier: Strings) gleich sind, das andere Mal, ob zwei Objekte ungleich sind. Diese beiden Vergleiche schreibt man in Python so:

`antwort == loesung`

beziehungsweise

`antwort != loesung`

Beides sind so genannte *logische Ausdrücke*. (`==` bedeutet: »ist gleich«; `!=` bedeutet: »ist nicht gleich«).

Was kann bei so einem Vergleich herauskommen? Entweder ergibt der Vergleich wahr, oder er ergibt falsch. Wie geht Python damit um?

Das probieren wir gleich mit dem IPI aus:

➤ Mach mit:

```
>>> loesung = "Python"
>>> antwort = "Python"
>>> antwort == loesung
True
>>> antwort != loesung
False
```

`True` und `False` sind reservierte Wörter. Sie bezeichnen die zwei Wahrheitswerte: wahr und falsch.

```
>>> antwort = "Java"
>>> antwort == loesung
False
```

Falsch, `antwort` und `loesung` sind nicht gleich.

Mini-Quiz



```
>>> loesung != antwort  
True
```

Wahr, tatsächlich ist `loesung` nicht gleich `antwort`.

```
>>> loesung = "33"  
>>> antwort = "33"  
>>> loesung == antwort  
True  
>>> loesung = 33  
>>> loesung == antwort  
False
```

Falsch? Ach ja, für Python ist eben der String "33" *nicht gleich* der Zahl 33!

Wir sehen, der IPI wertet diese logischen Ausdrücke – sie werden auch oft *boolesche Ausdrücke* genannt – aus. Das Ergebnis ist stets entweder wahr oder falsch, True oder False.

Es ist ein häufiger Fehler, beim Vergleich zweier Objekte zu schreiben: `a = b` anstatt `a == b`. Das Gleichheitszeichen stellt eine Wertzuweisung dar, das doppelte Gleichheitszeichen einen Vergleich. `a = b` als Bedingung in einer `if`-Anweisung führt zu einem Syntaxfehler.



Je besser du mit logischen Ausdrücken umgehen kannst, desto leichter werden dir `if`-Anweisungen fallen.

Der erste Teil des obigen Programm-Entwurfs wird so codiert (gib das Folgende nicht sofort ein, sondern verwende es später als ersten Teil deines Miniquiz-Programmes):

```
frage = "Welche Programmiersprache lernst du gerade? "  
loesung = "Python"  
  
antwort = input(frage)  
if antwort == loesung:  
    print("Richtig!")
```

Der zweite Teil geht ganz ähnlich:

```
if antwort != loesung:  
    print("Leider falsch!")  
    print("Richtig ist: ", loesung)
```

4



Muster 4: Bedingte Anweisung

Im Programmentwurf:

Wenn Bedingung :

Anweisung 1

Anweisung 2

...

Im Programmcode:

```
if Bedingung : ← Anweisungskopf
    Anweisung1
    Anweisung2 } Anweisungskörper
    ...
    ...
```

Was ist an diesem Muster charakteristisch?

Die Form der if-Anweisung

Die `if`-Anweisung besteht aus zwei Teilen: aus dem Anweisungskopf und dem Anweisungskörper.

Der Anweisungskopf beginnt mit dem reservierten Wort `if`. Danach folgen ein logischer Ausdruck (z. B. ein Vergleich zweier Objekte) und dahinter ein Doppelpunkt. Der gehört zur Syntax der `if`-Anweisung. Vergisst du ihn, erhältst du einen Syntaxfehler.

Der Anweisungskörper besteht aus einer Folge von einer oder mehreren Anweisungen, die gemeinsam gleich weit gegenüber dem Anweisungskopf eingerückt sein müssen – ich empfehle wieder vier Zeichen.

Du siehst, das Muster der `if`-Anweisung hat Ähnlichkeit mit dem Muster der Funktionsdefinition. Viele zusammengesetzte Python-Anweisungen sind so aufgebaut.

Wir können mit dem IPI auch mit der `if`-Anweisung spielen und dabei beobachten, dass der IPI uns die Einrückungen selbstständig richtig vorschlägt.

Bei Eingaben längerer zusammengesetzter Anweisungen im IPI beachte:

Wenn du einen Syntaxfehler machst, bricht die Eingabe ab. Du kannst dann mit `Alt`+`P` die Eingabe zurückholen, ausbessern und mit der Eingabe fortfahren.



Zunächst nur eine Frage



Wie funktioniert die if-Anweisung?

Trifft der Python-Interpreter auf eine `if`-Anweisung, dann wertet er zuerst die Bedingung aus. Wenn diese Auswertung `True` ergibt, werden die Anweisungen des Körpers der `if`-Anweisung ausgeführt. Andernfalls geschieht nichts weiter.

Du kannst die bedingte Anweisung auch interaktiv mit dem IPI untersuchen (die Namen `antwort` und `loesung` sollten noch dieselben Werte haben wie oben):

» Mach mit:

```
>>> if antwort == loesung:  
     print("Ja!")
```

Nichts ausgegeben! Warum?

```
>>> antwort  
'33'  
>>> loesung  
33
```

Ah ja, das hatten wir oben.

```
>>> loesung = "33"  
>>> if antwort == loesung:  
     print("Ja!")
```

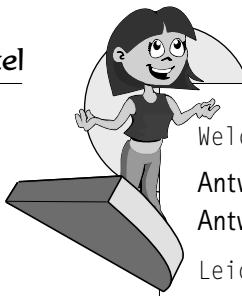
Ja!
>>> if antwort == loesung:
 print("Ja!")
 print("Ja, sicher!")
 print("SICHER!!!")

Ja!
Ja, sicher!
SICHER!!!
>>>

Zunächst nur eine Frage

» Jetzt hast du genug gelernt, um einen ersten Teil unseres Mini-Quiz programmieren zu können. Stellen wir vorerst nur eine einzige Frage:

4



Welche Programmiersprache lernst du gerade?

Antwortet der Benutzer *Python*, dann soll die Antwort richtig lauten.

Antwortet der Benutzer mit irgendeinem anderen Wort, dann:

Leider falsch!

Richtig ist: Python

- ⇒ Starte den IPI, öffne ein Editor-Fenster und schreibe einen Kopfkommentar für das Programm `miniquiz_arbeit.py`!
- ⇒ Schreibe den Programmcode darunter, wie wir ihn weiter oben entwickelt haben!
- ⇒ Sichere das Programm und führe es zum Testen zweimal aus: einmal mit einer falschen Antwort und einmal mit einer richtigen Antwort!
- ⇒ Zur Kontrolle vergleiche deinen Code mit dem Listing, das oben vor dem Muster 4 abgedruckt ist.
- ⇒ Speichere eine Kopie des Programms als `miniquiz01.py` ab.

Mini-Quiz erweitern

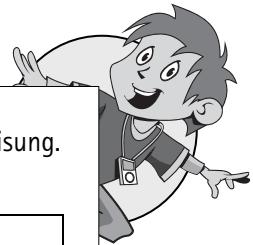
Bevor wir das Mini-Quiz ausbauen, möchte ich dir eine erweiterte Form der `if`-Anweisung vorstellen. Im obigen Programm wird doch zuerst geprüft, ob `antwort` gleich `loesung` ist, und etwas weiter unten, ob `antwort` nicht gleich `loesung` ist. Dabei ist doch klar: Wenn der erste Vergleich wahr ergibt, muss der zweite falsch ergeben und umgekehrt, so dass wir unsere Problemlösung einfacher auch so formulieren können:

`frage` ⇒ Fragentext
`loesung` ⇒ Lösungstext

`antwort` ⇒ Eingabe(`frage`)
Wenn `antwort` gleich `loesung` :
 Ausgabe: »Richtig!«
sonst:
 Ausgabe: »Leider falsch!«
 Ausgabe: »Richtig ist:« `loesung`

So etwas nennen wir eine *Programmverzweigung*: Eine Bedingung wird überprüft und je nachdem, ob das Ergebnis wahr oder falsch ist, wird der entsprechende Block von Anweisungen ausgeführt.

Mini-Quiz erweitern



Auch dafür ist in Python vorgesorgt, mit der if...else-Anweisung.
Beispiel:

```
if antwort == loesung:  
    print("Richtig!")  
else:  
    print("Leider falsch!")  
    print("Richtig ist: ", loesung)
```

Achte darauf, dass else genau unter if steht. Nur so kann der Python-Interpreter die Struktur der Verzweigung erkennen.

Somit sparst du im Programm einen Vergleich ein und der Python-Interpreter muss einen Vergleich weniger ausführen.

Auch diese sehr wichtige Form der Verzweigung wollen wir als Muster festhalten:

Muster 5: Programm-Verzweigung

Im Programmentwurf:

Wenn Bedingung :
 AnweisungA1
 AnweisungA2
 ...
Sonst:
 AnweisungB1
 AnweisungB2

(Bedingung ist ein boolescher Ausdruck)

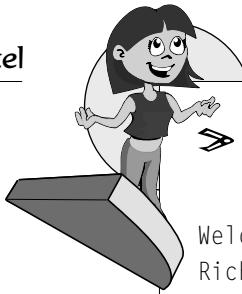
Im Programmcode:

```
if Bedingung:  
    AnweisungA1  
    AnweisungA2  
    ...  
else:  
    AnweisungB1  
    AnweisungB2  
    ...
```

} if-Zweig (falls Bedingung wahr ist)
} else-Zweig (falls Bedingung falsch ist)

➤ **Aufgabe:** Ändere miniquiz_arbeit.py so, dass darin eine Verzweigungsanweisung anstelle zweier bedingter Anweisungen benutzt wird! Speichere eine Kopie als miniquiz02.py ab!

4



➤ **Aufgabe:** Erweitere miniquiz_arbeit.py so, dass es folgende Ausgabe erzeugen kann:

Welche Programmiersprache lernst du gerade? *Python*
Richtig!

Mit welchem reservierten Wort beginnt eine
Funktionsdefinition? *fun*
Leider falsch!
Richtig ist: *def*

Wie viele reservierte Worte hat Python? *101*
Leider falsch!
Richtig ist: *33*

Du hast 1 von 3 Punkten erreicht!

Anleitung:

- 1) Du musst den Code von Miniquiz kopieren und noch zwei Mal (für die beiden anderen Fragen) darunter anfügen. Danach kannst du die Strings *frage* und *loesung* entsprechend der Vorlage ändern.
- 2) Du musst eine Variable für das Mitzählen der richtigen Antworten einführen! Als Variablennamen schlage ich *punkte* vor. Weise ihr am Anfang des Programms den Wert 0 zu!
- 3) In jedem der drei Code-Abschnitte für die Quizfragen muss im *if*-Zweig der Wert von *punkte* um 1 erhöht werden:

punkte = punkte + 1

Das wär's. Teste nun dein Programm!

Wie viele Punkte hast du erreicht? Steht nicht da? Dann hast du vergessen, die Ausgabe der Punkte zu programmieren.

➤ Sobald du das Programm erfolgreich getestet hast, speichere eine Kopie unter dem Namen *miniquiz03.py* ab.

Wie immer sind auch die miniquiz-Programme auf der Buch-CD enthalten, und zwar im Verzeichnis *py4kids-material\kap04\programme*.





Eine Funktion verwenden

Auch in diesem Programm tritt ein Code-Abschnitt mehrfach auf:

```
antwort = input(frage)
if antwort == loesung:
    print("Richtig!")
    punkte = punkte + 1
else:
    print("Leider falsch!")
    print("Richtig ist:", loesung)
print()
```

Die letzte `print`-Anweisung ist nicht Bestandteil der Verzweigung. Du und der Interpreter erkennen das an ihrer Position: Weil sie nicht eingerückt ist, gehört sie nicht mehr zum `else`-Zweig. Die durch dieses `print` erzeugte Leerzeile wird auf jeden Fall ausgegeben.

Wir können diesen Abschnitt in einer Funktion `quizfrage()` zusammenfassen. Wir müssen aber überlegen, welche lokalen Variablennamen in dieser Funktion erzeugt werden. Das sind alle Namen, denen ein Objekt zugewiesen wird: `antwort` und `punkte`. Für `antwort` ist das perfekt. `punkte` aber wird schon außerhalb der Funktion definiert. Der Wert von `antwort` wird nur innerhalb der Funktion verwendet. Um zu verhindern, dass bei der Zuweisung `punkte = punkte + 1` – wie bei `dreieck08a.py` gehabt – eine Fehlermeldung erzeugt wird, müssen wir für `punkte` eine global-Anweisung einfügen.

```
def quizfrage():
    global punkte
    antwort = input(frage)
    if antwort == loesung:
        print("Richtig!")
        punkte = punkte + 1
    else:
        print("Leider falsch!")
        print("Richtig ist:", loesung)
    print()
```

Damit können wir unser `miniquiz_arbeit.py` weiter umarbeiten und gelangen zu einer kompakteren Lösung:

4



```
# Autor: Gregor Lingl
# Datum: 6. 8. 2008
# miniquiz04.py : Python für Kids, Kapitel 4
#                   Quiz aus drei Fragen.

def quizfrage():
    global punkte
    antwort = input(frage)
    if antwort == loesung:
        print("Richtig!")
        punkte = punkte + 1
    else:
        print("Leider falsch!")
        print("Richtig ist:", loesung)
    print()

punkte = 0

frage = "Welche Programmiersprache lernst du?"
loesung = "Python"
quizfrage()

frage = """Mit welchem reservierten Wort
beginnt eine Funktionsdefinition?"""
loesung = "def"
quizfrage()

frage = "Wie viele reservierte Worte hat Python?"
loesung = "33"
quizfrage()

print("Du hast", punkte, "von 3 möglichen Punkten
erreicht!").
```

- Führe die nötigen Änderungen in `miniquiz_arbeit.py` aus. Wenn du es erfolgreich getestet hast, speichere eine Kopie des Programms als `miniquiz04.py` ab.



Mehrfachverzweigung

Wenn wir uns die geplante Programmausgabe auf Seite 116 ansehen, bemerken wir: Nun fehlt uns neben dem Begrüßungsteil nur noch die abschließende Beurteilung. In unserem kleinen Beispiel soll das so vor sich gehen: Wenn man drei Punkte erreicht hat, soll die Beurteilung schlicht »Super!« sein. Wenn man ein oder zwei Punkte erreicht hat, dann soll die Beurteilung wohlwollend sein, etwa: »Fein, du hast schon einiges gelernt!« Wenn man 0 Punkte erreicht hat, dann soll die Beurteilung realistisch sein, nach dem Motto: »Du stehst noch ziemlich am Anfang!«

Da haben wir drei Fälle zu unterscheiden und kommen mit einer gewöhnlichen Verzweigungsanweisung nicht aus. Wir haben es also mit einer Mehrfachverzweigung zu tun. Auch dafür hat Python vorgesorgt. Schreiben wir uns zunächst wieder auf, was wir brauchen:

Wenn `punkte` gleich 3 ist:

Ausgabe: »Super!«

sonst, wenn `punkte` größer als null ist:

Ausgabe: »Fein, du hast schon einiges gelernt!«

sonst:

Ausgabe: »Du stehst noch ziemlich am Anfang!«

Der Interpreter prüft der Reihe nach die Bedingungen. Wenn er auf eine trifft, die wahr ergibt, dann führt er den zugehörigen Anweisungsblock aus und beendet die Mehrfachverzweigung, das heißt, er führt keine weiteren Überprüfungen mehr durch. Wenn keine angegebene Bedingung wahr ist, dann führt er den Code aus, der im `else`-Zweig steht (falls ein solcher vorhanden ist).

In Python wird das so codiert:

```
if punkte == 3:  
    print("Super!")  
elif punkte > 0:  
    print("Fein, du hast schon einiges gelernt!")  
else:  
    print("Du stehst noch ziemlich am Anfang!")
```

Du entdeckst hier ein neues reserviertes Wort: `elif`. Dieses reservierte Wort entstand aus der Zusammenziehung von `else` mit `if` (sonst – wenn).

4



Wenn du dem IPI zusammengesetzte Anweisungen, wie z. B. die `if-elif-else`-Anweisung eingibst, musst du speziell auf die Einrückungen achten!

Nach einem Anweisungskopf wie `if punkte == 3:` rückt der IPI die Anweisungen des folgenden Blocks, des `if`-Zweigs, automatisch ein. Wenn du nun diesen Block abschließen und mit `elif punkte > 0:` fortfahren willst, musst du zuerst die Einrückung mit der -Taste rückgängig machen und dann mit `elif` forsetzen.

Beachte weiter: Auch mehrzeilige Anweisungen lassen sich mit + zurückholen und anschließend – wenn gewünscht – abändern.

Mach mit!

```
>>> punkte = 3
>>> if punkte == 3:
        print("Super!")
    elif punkte > 0:
        print("Na ja!")
    else:
        print("Schlück!")
```

Super!

```
>>> punkte = 0
```

Drücke + , nochmals + und dann .

Schlück!

Hier tritt eine neue Art von Bedingung auf: `punkte > 0`. Sie prüft, ob der Wert von `punkte` größer als null ist. Neben `==` und `!=` können als Vergleichsoperatoren auch folgende verwendet werden: `<`, `>`, `<=`, `>=`. Die letzten beiden prüfen auf »kleiner oder gleich« bzw. »größer oder gleich«:

```
>>> a = 4; b = 5; c = 5
>>> print(a < b, a <= b, a >= b, b >= c, b > c)
True True False True False
```

Entwickle aus `miniquiz04.py` das Programm `miniquiz05.py`, das am Ende die beschriebene Beurteilung enthält. Am Anfang des Programms soll ein Begrüßungsabschnitt eingefügt werden, in dem auch nach dem Namen des Benutzers gefragt wird. Am Ende wird dieser Name bei der Beurteilung nochmals ausgegeben.



Clara Pythias Python-Special: Eigenwerbung

Schließlich folgt noch ein »Werbebanner« für das Python-Video »IntroducingPython« auf der Buch-CD. In dem Video reden junge Leute und Python-Gurus, auch Guido, über Python. Python ist keine kommerzielle, sondern freie Software. Dabei bezieht sich »frei« auf die Freiheiten für den Nutzer der Software. Sieh' mal in der Wikipedia nach: http://de.wikipedia.org/wiki/Freie_Software. Außerdem wird Python nicht verkauft, sondern ist kostenlos erhältlich. Daher macht auch keine Firma Werbung dafür, sondern es wird durch die Mundpropaganda der Benutzer und durch Kommunikation im Internet bekannt gemacht.

Mehrfachverzweigungen werden häufig gebraucht – daher lohnt es sich, sie im Kopf zu verankern – als Muster!

Muster 6: Mehrfache Verzweigung

Im Programmentwurf:

Wenn Bedingung 1:

AnweisungA1

AnweisungA2

...

sonst wenn Bedingung2:

AnweisungB1

AnweisungB2

...

Mehrere sonst-wenn-Zweige möglich

...

sonst:

AnweisungN1

AnweisungN2

...

Im Programmcode:

```
if Bedingung1:
```

```
    AnweisungA1
```

```
    AnweisungA2
```

```
    ...
```

Bedingung ist ein boolescher Ausdruck

if-Zweig (falls *Bedingung1* wahr ist)

4



```

elif Bedingung2:
    AnweisungB1
    AnweisungB2
    ...
    ...
else:
    AnweisungN1
    AnweisungN2
    ...

```

... weitere elif-Zweige

... falls alle Bedingungen falsch sind

Jeder Zweig enthält einen Anweisungsblock – das heißt eine Folge von Anweisungen. Ich werde im Weiteren dafür auch *Block* schreiben. Das ist kürzer und wegen der Notwendigkeit der Nummerierung auch klarer:

Muster 6: Mehrfache Verzweigung (Alternative Darstellung):

Im Programmcode:

```

if Bedingung1:
    Block1
elif Bedingung2:
    Block2
elif ... : ...
else:
    Block

```

Bedingung ist ein boolescher Ausdruck

... falls Bedingung1 wahr ist

... falls Bedingung2 wahr ist

... falls nötig, weitere elif-Zweige

... falls alle Bedingungen falsch sind

Und merke dir dazu: Jeder Block besteht aus einer oder mehreren Anweisungen.

Zusammenfassung

- ◊ Eine Folge von Anweisungen lässt sich zu einer Funktion zusammenfassen.
- ◊ Eine Funktion wird durch eine `def`-Anweisung erzeugt.
- ◊ Eine Funktionsdefinition besteht aus dem Funktionskopf und dem Funktionskörper. Der Funktionskörper ist ein Block von Anweisungen.
- ◊ Beim Aufruf einer Funktion werden die Anweisungen des Funktionskörpers ausgeführt.

Einige Aufgaben ...



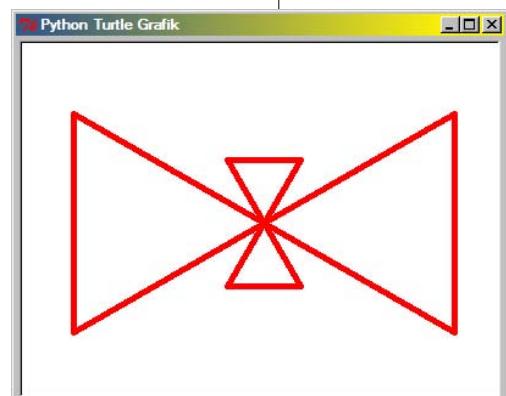
- ❖ Variablen, denen im Funktionskörper ein Wert zugewiesen wird, sind normalerweise lokale Variablen der Funktion.
- ❖ Soll einer globalen Variablen im Code einer Funktion ein Wert zugewiesen werden, dann muss sie im Funktionskörper mit einer `global`-Anweisung als global deklariert werden.
- ❖ Die bedingte Anweisung ist in Python die `if`-Anweisung.
- ❖ Im Anweisungskopf der `if`-Anweisung steht nach dem reservierten Wort `if` ein logischer Ausdruck, dessen Auswertung einen der beiden Wahrheitswerte *wahr* oder *falsch* ergibt. In Python wird *wahr* durch `True` und *falsch* durch `False` dargestellt.
- ❖ Logische Ausdrücke (boolesche Ausdrücke) können mit den Vergleichsoperatoren `==`, `!=`, `<`, `>`, `<=`, `>=` gebildet werden.
- ❖ Für einfache Programmverzweigungen wird die `if...else`-Anweisung verwendet.
- ❖ Für mehrfache Verzweigungen wird die `if...elif...else`-Anweisung verwendet.
- ❖ In diesem Kapitel hast du folgende neue reservierte Wörter kennengelernt: `def` `global` `if` `elif` `else`.
- ❖ In diesem Kapitel hast du die Turtle-Grafik-Funktion `color` kennengelernt. `color(farbe1, farbe2)` stellt die Stiftfarbe auf `farbe1` ein und die Füllfarbe auf `farbe2`.

Einige Aufgaben ...

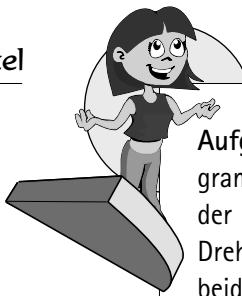
Aufgabe 1: Schreibe `wabe.py` zu `wabe2.py` um, wobei du eine Funktion `sechseck` verwenden sollst. Versuche, das innere Sechseck mit einem Aufruf dieser Funktion zu zeichnen.

Aufgabe 2: Erstelle ein Programm, das eine Funktion `dreieck()` benutzt, um diese Grafik zu erzeugen. Die kleinen Dreiecke haben Seitenlänge `kurz = 60`, die großen Seitenlänge `lang = 180`.

Aufgabe 3: Erstelle ausgehend von `quadrat06.py` aus Kapitel 3 ein Programm `quadrat09.py`, in dem du eine Funktion `quadrat` und `fuelle_quadrat` definierst und verwendest. Es soll die gleiche Zeichnung erstellen wie `quadrat06.py`.



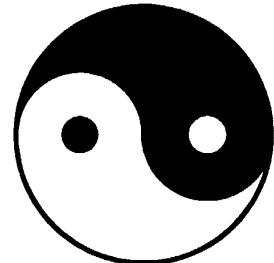
4



Aufgabe 4: Erstelle ausgehend von `quadrat07.py` aus Kapitel 3 ein Programm `quadrat10.py`, mit einer weiteren Benutzereingabe für die Anzahl der Quadrate. Die eingegebene Zahl soll zwischen 2 und 6 liegen. Der Drehwinkel der Quadrate soll der Anzahl angepasst werden. Die ersten beiden Quadrate werden unbedingt gezeichnet. Die weiteren nur bedingt: zum Beispiel wird das fünfte Quadrat nur gezeichnet, wenn die Anzahl der Quadrate größer als vier ist.

Aufgabe 5: Erstelle ein Programm `yinyang.py`, das mittels Turtle-Grafik ein Yinyang-Symbol zeichnet.

Aufgabe 6: Schreibe ein Programm `meinquiz.py`, das Fragen zu deinem Lieblingshobby stellt. Es soll mehr als drei Fragen stellen und eine fantasievolle Beurteilung ausgeben.



... und einige Fragen

1. Was ist der Unterschied zwischen `a = 3` und `a == 3`?
2. Was gibt die `print`-Anweisung im folgenden IPI-Dialog aus:

```
>>> x = 999
>>> y = 4
>>> def testfun():
    x = 2 * y

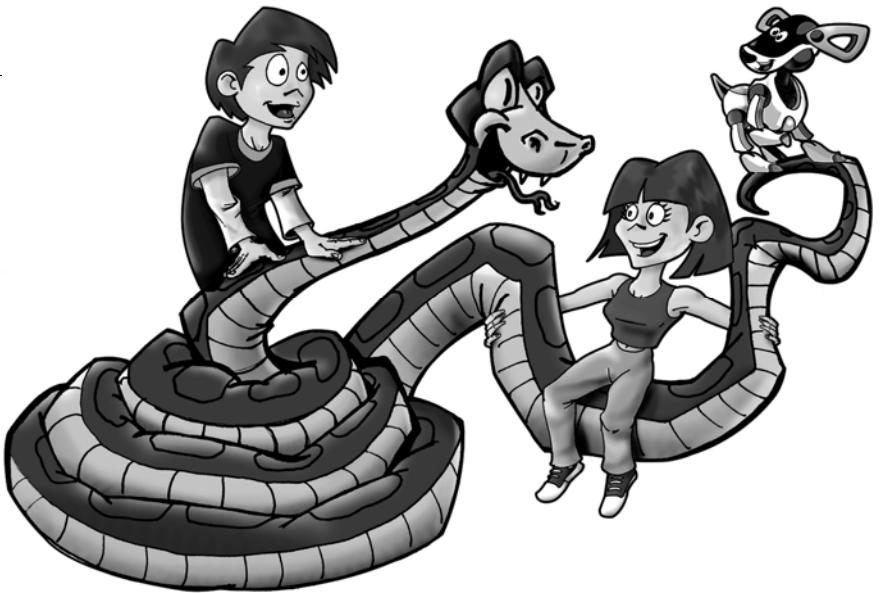
>>> testfun()
>>> print(x)
```

3. Durch *Einfügen* einer Zeile kannst du im Dialog der vorigen Frage erreichen, dass 8 ausgegeben wird. Welche Zeile muss wo eingefügt werden?
4. Was gibt der IPI auf folgende Eingabe als Antwort:

```
>>> "drei" == 3
```

5. ... und was auf diese Eingabe:

```
>>> drei == 3
```



5

Funktionen mit Parametern

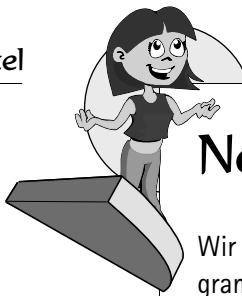
In den ersten Kapiteln dieses Buches hast du gelernt, eine Reihe von Funktionen des `turtle`-Moduls anzuwenden. Im vierten Kapitel hast du erfahren, wie du eigene Funktionen definieren kannst. Die meisten Turtle-Grafik-Anweisungen – wie zum Beispiel `forward()` – verlangen Argumente, um ihre Aufgabe ausführen zu können. Es wäre sehr praktisch, wenn unsere eigenen Funktionen das auch könnten. Zum Beispiel wäre es fein

```
>>> dreieck(100)  
schreiben zu können, anstatt  
>>> seite = 100  
>>> dreieck()
```

Wie das geht und welche Vorteile es bringt, darum geht es in diesem Kapitel. In diesem Kapitel lernst du ...

- Ⓐ wie man Funktionen mit Parametern definiert.
- Ⓑ dass Parameter lokale Variablen der Funktionen sind.
- Ⓒ wie man beim Aufruf solcher Funktionen Argumente für die Parameter einsetzt.
- Ⓓ schließlich, wie du mit dem Aufbau deiner eigenen Werkzeugbibliothek beginnst.
- Ⓔ einiges über das Kombinieren von Strings.

5



Noch einmal Dreiecke

Wir gehen von einer besonders einfachen Version unseres Dreiecksprogramms aus.

➤ Lade dreieck_arbeit.py aus dem Ordner C:\py4kids\kap05 in ein Editor-Fenster. Führe es aus und überlege, wie es funktioniert.

```
dreieck_arbeit.py - C:\py4kids\kap05\dreieck_arbeit.py
File Edit Format Run Options Windows Help
# Autor: Gregor Lingl
# Datum: 6. 8. 2009
# dreieck_arbeit.py
# Zum Einstieg in 'Funktionen mit Parametern'.
from turtle import *

def dreieck():
    """Zeichne Dreieck mit Seitenlänge seite."""
    forward(seite)
    left(120)
    forward(seite)
    left(120)
    forward(seite)
    left(120)

    reset()
    pensize(10)
    right(90)

    pencolor("red")
    seite = 65
    dreieck()
    left(120)

    pencolor("green")
    seite = 100
    dreieck()
    left(120)

    pencolor("blue")
    seite = 135
    dreieck()
    left(120)

hideturtle()

Ln: 4 Col 16
```

Dort wollen wir dreimal jeweils die zwei Zeilen der Form

seite = 65

dreieck()

durch einen Aufruf der Form

dreieck(65)

ersetzen. Dazu müssen wir die Definition der Funktion dreieck() ändern. Die Funktion muss die Information, wie lang die Dreiecksseite sein soll – als Argument – übernehmen können. Wir erreichen das, indem wir einen Variablenamen bereitstellen, dem beim Funktionsaufruf das passende Objekt zugewiesen wird. Ein solcher Variablenname heißt Parameter der Funktion.

Ich zeige das an einem ganz einfachen Beispiel mit dem IPI:

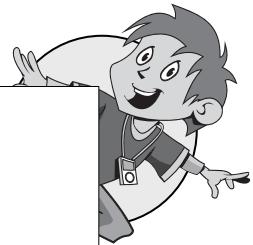
➤ Mach mit!

Definiere zunächst eine Funktion, die eine Zahl und das Zweifache der Zahl ausgeben soll:

```
>>> def zweifach(zahl):
        print("Die Zahl ist:", zahl)
        print("Das Doppelte ist:", 2 * zahl)
```

Dann rufe die Funktion auf:

Noch einmal Dreiecke



```
>>> zweifach(3)
```

Die Zahl ist: 3

Das Doppelte ist: 6

Welchen Wert hat zahl jetzt für den IPI?

```
>>> zahl
```

Traceback (most recent call last):

```
  File "<pyshell#5>", line 1, in <module>
```

```
    zahl
```

NameError: name 'zahl' is not defined

```
>>>
```

Aha, ein Namensfehler! Das zeigt uns, dass der Name zahl offenbar ein lokaler Name der Funktion zweifach() ist. Außerhalb der Funktion existiert er nicht.

Du bist jetzt sicher neugierig, ob wir das mit unserer Funktion dreieck() genauso machen können. Also probieren wir das rasch mit dem IPI aus, wobei wir als Namen für die Länge der Dreiecksseite den Buchstaben a wählen:

➤ Mach weiter mit!

```
>>> from turtle import *
```

```
>>> reset()
```

```
>>> pensize(3)
```

```
>>> def dreieck(a):
```

```
    forward(a)
```

```
    left(120)
```

```
    forward(a)
```

```
    left(120)
```

```
    forward(a)
```

```
    left(120)
```

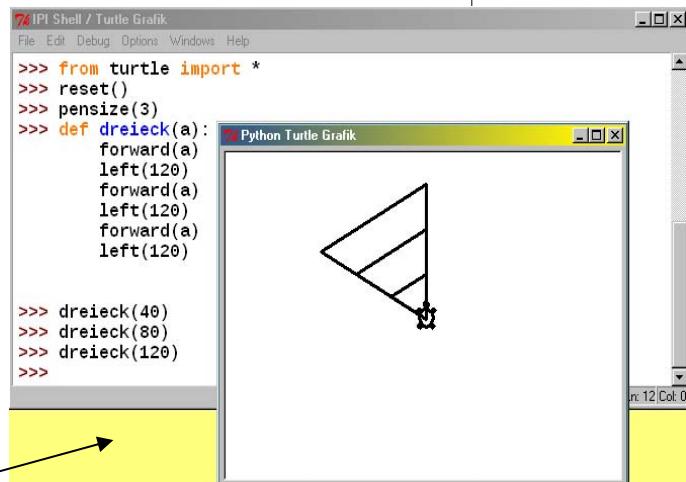
Nun übernimmt die Funktion dreieck() beim Aufruf ein Argument. Probiere das mit drei Aufrufen von dreieck() aus.

```
>>> dreieck(40)
```

```
>>> dreieck(80)
```

```
>>> dreieck(120)
```

Damit ergibt sich folgendes Bild:



Durch Einfügen der Anweisung `print("a:", a)` in den Code von dreieck() kannst du sichtbar machen, welchen Wert der Name a bei einer Ausführung von dreieck() hat.

5



```
>>> reset()
>>> def dreieck(a):
    print("a:", a)
    forward(a)
    left(120)
    forward(a)
    left(120)
    forward(a)
    left(120)

>>> dreieck(40)
a: 40
>>> seite = 60
>>> dreieck(seite)
a: 60
>>> a = 90
>>> dreieck(a)
a: 90
>>> dreieck(2 * seite)
a: 120
>>> a
90
```

Diese Übung zeigt bemerkenswerte und auch merkenswerte Tatsachen über Variablen auf:

- ❖ Wir haben in dieser Übung willkürlich als Parameter der Funktion `dreieck()` den Namen `a` verwendet.
- ❖ Beim Funktionsaufruf muss an der Stelle von `a` ein Objekt eingesetzt werden. Dies kann ein Zahlenwert sein (`40`), ein Name (`seite`) oder ein Ausdruck (`2*seite`).
- ❖ Der eingesetzte Name kann gleichlautend mit dem Parameternamen sein. Dennoch sind es zwei verschiedene Namen. Wie die letzte Ermittlung des Werts von `a` zeigt, behält der *globale* Name `a` seinen Wert `90` auch, nachdem der Parameter `a`, eine lokale Variable, den Wert `120` hatte.
- ❖ Im Allgemeinen ist es keine gute Idee, Parameter genauso zu benennen wie globale Variablen.
- ❖ Es ist auch keine gute Idee, Parameternamen so nichtssagend wie `a` zu wählen. In einer interaktiven Sitzung geht das gerade noch an, denn dabei vergisst du wohl kaum die Bedeutung des Parameters. In einem Programm, das du Tage oder Wochen später wieder verwenden möch-

Noch einmal Dreiecke



test, sind aussagekräftige Namen besser. Sie tragen dazu bei, dass »sich das Programm selbst dokumentiert«!

Parameter sind lokale Variablen der Funktion, zu der sie gehören. Sie sind daher in anderen Programmabschnitten unbekannt.

Wir verbessern nun unser Programm `dreieck_arbeit.py`, indem wir die Funktionsdefinition von `dreieck()` mit einem Parameter versehen. Als Namen des Parameters wähle `laenge`. Damit ist klar, was gemeint ist, und der Parametername ist ein anderer als der globale Name `seite`.

- Solltest du inzwischen das Editor-Fenster mit `dreieck_arbeit.py` geschlossen haben, lade die Datei erneut!
- Füge in der Definition von `dreieck()` den Parameter `laenge` ein! Der Parameter gehört zwischen die runden Klammern im Funktionskopf.
- Ersetze in den drei `forward()`-Anweisungen von `dreieck()` das Argument `seite` durch `laenge`! Tue dasselbe auch noch im Doc-String.
- Trage die gewünschten Argumente in die drei Aufrufe von `dreieck()` ein, z. B. `dreieck(65)`!
- Die Wertzuweisungen an die Variable `seite` sind nun überflüssig und können gelöscht werden.
- Speichere das Programm und teste, ob es fehlerfrei funktioniert!
- Speichere eine Kopie als `dreieck09.py` ab.

Wenn du alle meine Vorschläge befolgt hast, dann kennt Python nun die Funktion `dreieck()` mit dem Parameter `laenge`:

```
# Datum: 6. 8. 2009
# dreieck09.py
# Zum Einstieg in 'Funktionen mit Parametern'.

from turtle import *

def dreieck(laenge):
    """Zeichne Dreieck mit Seitenlänge seite."""
    forward(laenge)
    left(120)
    forward(laenge)
    left(120)
    forward(laenge)
    left(120)

reset()
pensize(10)
right(90)

pencolor("red")
dreieck(65)
left(120)

pencolor("green")
dreieck(100)
left(120)

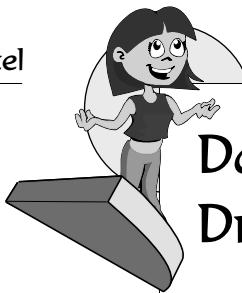
pencolor("blue")
dreieck(135)
left(120)

hideturtle()

Ln: 35 Col: 12
```

```
>>> dreieck()
    (laenge)
    Zeichne Dreieck mit Seitenlänge laenge.
Ln: 29 Col: 12
```

5



Das geht auch mit gefüllten Dreiecken

Wir wollen nun noch unser Programm dreieck08.py aus Kapitel 4 so abändern, dass es eine Funktion `fuelle_dreieck()` mit dem Parameter `seitenlaenge` benutzt.

Dabei ist jedoch Folgendes zu beachten: In diesem Programm kommen die beiden Namen `seite` und `aenderung` vor. Vor jedem Aufruf von `fuelle_dreieck()` muss weiterhin `seite` geändert werden. Der geänderte Wert kann dann in die Funktion `fuelle_dreieck()` als Argument eingesetzt werden. Das heißt, dass du drei Mal eine Passage folgender Bauart im Programm haben wirst:

```
color("red", "cyan")
fuelle_dreieck(seite)
left(120)
seite = seite + aenderung # fehlt beim letzten Mal
```

- Lade `dreieck08.py` aus Kapitel 4, ändere den Kopfkommentar und speichere es unter dem Namen `dreieck_arbeit.py` im Ordner `C:\py4kids\kap05`.
- Ändere die Funktion `dreieck()` so, dass sie einen Parameter `laenge` hat! (Gerade so wie in `dreieck09.py`.)
- Ändere die Funktion `fuelle_dreieck()` so, dass sie einen Parameter `seitenlaenge` hat! Beim Aufruf von `dreieck()` im Funktionskörper von `fuelle_dreieck()` muss dann `seitenlaenge` als Argument eingesetzt werden.

```
def fuelle_dreieck(seitenlaenge):
    """Zeichne gefülltes Dreieck
    mit Seitenlänge seitenlaenge.
    """
    begin_fill()
    dreieck(seitenlaenge)
    end_fill()
```

- Ändere die drei Funktionsaufrufe von `fuelle_dreieck()` derart, dass als Argument `seite` eingesetzt wird!



```
color("red", "cyan")
fuelle_dreieck(seite)

left(120)
seite = seite + aenderung

color("green", "magenta")
fuelle_dreieck(seite)

left(120)
seite = seite + aenderung

color("blue", "yellow")
fuelle_dreieck(seite)
```

- Speichere das Programm und teste es!
- Speichere eine Kopie des Programms als dreieck10.py.

Vielleicht denkst du jetzt: Was bringt denn das? Wir haben keine einzige Anweisung eingespart und das Programm ist eher komplizierter geworden.

Immerhin bleibt der Vorteil, dass wir nicht wissen müssen, wie die Funktionen `dreieck()` und `fuelle_dreieck()` programmiert sind, insbesondere, welcher Variablenname für die Länge der Dreieckseite verwendet wird.

Mache dir das am Beispiel der Funktion `forward()` klar: Sicherlich wäre es sehr unpraktisch, wenn wir `forward()` in der Form

```
distanz = 50
forward()
```

verwenden müssten. Wir müssten dazu wissen und ständig daran denken, dass innerhalb der Funktion `forward()` der Variablenname `distanz` verwendet wird.

Funktionen mit mehreren Parametern

Oft ist es nötig, einer Funktion mehr als eine Information zu übergeben. Denke beispielsweise an die Turtle-Grafik-Funktion `color()`.

5



Stelle dir vor, du kennst von einem Rechteck die Länge, zum Beispiel 54.5 und die Breite, zum Beispiel 36.8. Du möchtest eine Funktion schreiben, die für jedes Rechteck auf den Bildschirm schreibt, wie groß sein Flächeninhalt ist.

Um der Funktion beide Werte übergeben zu können, werden wir sie mit zwei Parametern, `laenge` und `breite`, versehen. Um genau zu sehen, wie die Funktion arbeitet, wollen wir auch die Werte, auf die die Parameternamen `laenge` und `breite` verweisen, mit einer `print()`-Funktion ausgeben.

Das können wir ganz leicht mit dem IPI ausprobieren:

➤ Mach mit!

```
>>> def rechteckflaeche(laenge, breite):
    print("Länge:", laenge, " Breite:", breite)
    flaeche = laenge * breite
    print("Die Fläche beträgt:", flaeche)
```

```
>>> rechteckflaeche(16, 5)
Länge: 16   Breite: 5
Die Fläche beträgt: 80
>>> rechteckflaeche(5, 16)
Länge: 5   Breite: 16
Die Fläche beträgt: 80
```

Beachte: Der erste Parameter ist immer die Länge, der zweite die Breite unabhängig von ihrer Größe! Daher ist im letzten Beispiel die `laenge` kleiner als die `breite`.

```
>>> a = 54.5
>>> b = 36.8
>>> rechteckflaeche(a,b)
Länge: 54.5   Breite: 36.8
Die Fläche beträgt: 2005.6
>>>
```

Sehen wir uns nochmals den Code von `dreieck_arbeit.py` an. Vor jedem Aufruf unserer Funktion `fuelle_dreieck()` steht ein Aufruf der Turtle-Grafik-Funktion `color()`. Wir können diesen Aufruf in die Funktion `fuelle_dreieck()` hineinnehmen, wenn wir die Stiftfarbe und Füllfarbe für das Dreieck als zweiten und dritten Parameter festlegen:

Funktionen mit mehreren Parametern



```
def fuelle_dreieck(seitenlaenge, stiftfarbe, fuellfarbe):
    """Zeichne gefülltes Dreieck mit Seite seitenlaenge.
    Umrandung in stiftfarbe, Füllung mit fuellfarbe.
    """
    color(stiftfarbe, fuellfarbe)
    begin_fill()
    dreieck(seitenlaenge)
    end_fill()
```

Damit vereinfachen sich die Passagen, wo die Dreiecke gezeichnet werden:

```
seite = startseite
fuelle_dreieck(seite, "red", "cyan")

left(120)
seite = seite + aenderung
fuelle_dreieck(seite, "green", "magenta")

left(120)
seite = seite + aenderung
fuelle_dreieck(seite, "blue", "yellow")
```

➤ Führe diese beiden Änderungen in `dreieck_arbeit.py` aus. Teste das Programm und speichere danach eine Kopie unter dem Namen `dreieck11.py` ab.

Warum wir die beiden anderen Anweisungen nicht in die Funktion `fuelle_dreieck()` hineinnehmen, haben wir schon im vorigen Kapitel geklärt!

Funktionen mit Parametern gehören zu den wichtigsten Elementen von Python-Programmen. Daher hier zusammenfassend ein Muster dafür:

Muster 7: Definition einer Funktion mit Parametern

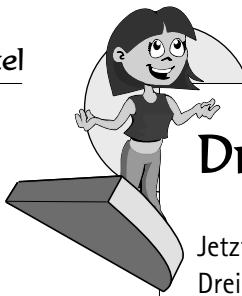
```
def Funktionsname(param1, param2, ...):      Funktionskopf
    Anweisung 1                                       }
    Anweisung 2                                       }
```

Funktionskörper

Anweisungen verwenden die lokalen Namen param1, param2, ...

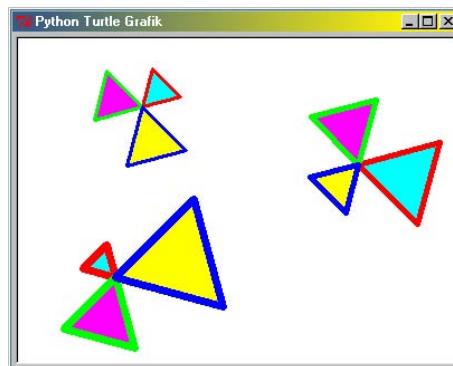
Alle Anweisungen des Funktionskörpers gleich weit eingerückt

5



Dreiecksmuster

Jetzt sind wir schon ganz wagemutig und wollen ein Muster aus solchen Dreierpacks von Dreiecken erzeugen, wie hier abgebildet:



Es sind nur noch zwei kleine Schritte, um dahin zu gelangen. Der erste besteht darin, das Zeichnen eines solchen Dreierpacks selbst in eine Funktion zu verpacken. Dazu müssen wir einfach die Anweisungen, die das tun, zum Funktionskörper einer Funktion `dreierpack()` machen. Um unterschiedlich gestaltete Dreierpacks damit zeichnen zu können, verwenden wir zwei Parameter: `seite` und `aenderung`:

```
def dreierpack(startseite, aenderung):
    """Zeichne Muster aus drei Dreiecken.
    Erstes Dreieck hat Seitenlänge startseite.
    """
    seite = startseite
    fuelle_dreieck(seite, "red", "cyan")

    left(120)
    seite = seite + aenderung
    fuelle_dreieck(seite, "green", "magenta")

    left(120)
    seite = seite + aenderung
    fuelle_dreieck(seite, "blue", "yellow")

    left(120)
```

➤ Füge also in `dreieck_arbeit.py` vor dem ersten `fuelle_dreieck()`-Aufruf den Funktionskopf und den Doc-String ein und rücke

jump()



die Anweisungen des Funktionskörpers ein. (Wie gehabt: markieren und [←].)

- ⇒ Die paar Anweisungen im Programm, die direkt ausgeführt werden sollen, verschiebe unter die Funktionsdefinitionen. Außerdem ist noch ein Aufruf der Funktion dreierpack() nötig, damit überhaupt etwas gezeichnet wird.

```
reset()  
pensize(10)  
right(90)  
dreierpack(65, 35)  
hideturtle()
```

In diesem Code-Abschnitt kommen vier Zahlen-Argumente vor. Alle diese Zahlen können abgeändert werden, wodurch andere Grafiken entstehen.

- ⇒ Speichere das Programm und führe es aus. Es sollte zunächst dieselbe Zeichnung entstehen wie vor der Programmänderung.
⇒ Wenn es korrekt funktioniert, speichere eine Kopie unter dem Namen dreieck12.py.

Experimentiere ein wenig mit dem Programm. Entweder, indem du in diesem letzten Abschnitt des Programmtexes die Zahlen änderst und das Programm wiederholt mit verschiedenen Werten ausführst. Oder, indem du in der IPI-Shell die Zeichnung immer wieder mit reset() löscht und derartige Anweisungen direkt eingibst.

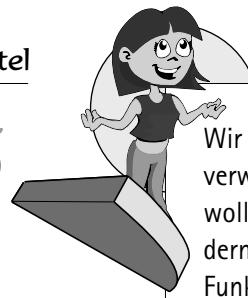
- ⇒ Mach mit! Zum Beispiel:

```
>>> reset()  
>>> pensize(3)  
>>> right(30)  
>>> dreierpack(100, -28)
```

jump()

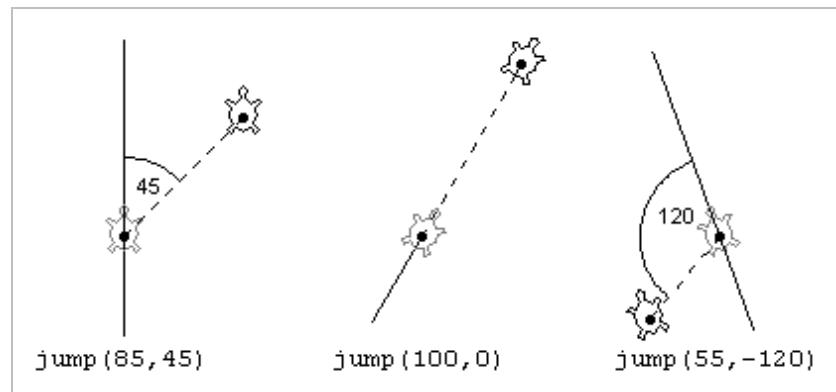
Wir wollen jetzt solche unterschiedlichen Dreierpacks an verschiedene Stellen im Zeichenfenster zeichnen. Zu diesem Zweck müssen wir mit der Turtle, ohne zu zeichnen, zu diesen Stellen springen. Das erreichen wir am besten mit einer Funktion jump(). Ich finde, das englische Wort *jump* passt besser zu all den übrigen Turtle-Grafik-Funktionen und gefällt mir auch einfach besser als etwa *springe()* oder Ähnliches ...

5



Wir wollen diese Funktion gleich so entwerfen, dass sie möglichst vielseitig verwendbar ist. Denn so etwas kann man sicher oft brauchen. Und zwar wollen wir, dass die Turtle damit nicht nur geradeaus springen kann, sondern auch unter einem bestimmten Winkel zur Seite. Die Turtle-Sprung-Funktion soll so funktionieren:

Wie `jump()` funktioniert.



Die Sprungstrecke, entlang der die Turtle mit `jump()` *nicht* zeichnet, ist hier *gestrichelt* dargestellt.

Mit welchen Angaben wollen wir festlegen, wie die Turtle springen soll? Naheliegend ist, die Länge der Sprungstrecke durch die Zahl der Einheiten festzulegen, etwa wie bei der Funktion `forward()` und die Richtung durch den Winkel relativ zur Ausgangsrichtung der Turtle. Dabei vereinbaren wir, dass die Turtle positive Winkel im Uhrzeigersinn misst – wie in der Turtle-Grafik gebräuchlich. Eine Drehung um einen positiven Winkel ist daher als Rechtsdrehung auszuführen. Für Linksdrehungen sind negative Winkel zu verwenden.

Außerdem wollen wir dafür sorgen, dass die Turtle nach dem Sprung wieder in dieselbe Richtung schaut wie vorher und dass nach dem Sprung der Zeichenstift stets unten ist.

➤ Sehen wir uns das mit dem IPI an. Lassen wir die Turtle zunächst nach rechts oben springen. Mach mit!

```
>>> reset(); dot()
>>> distanz = 85
>>> winkel = 45
>>> penup()
>>> right(winkel)
>>> forward(distanz)
>>> left(winkel)
>>> pendown()
```

Ran ans Dreiecksmuster



Das funktioniert. Jetzt vielleicht flach nach links unten. Nehmen wir `distanz = 180`. Als Winkel sollte vielleicht `250` passen. Oder, interessante Idee: `winkel = -110`.

➤ Führe die obigen sieben Anweisungen nochmals mit *diesen* Angaben aus.

Schaut aus, als ob es funktionierte! Du siehst, für jeden Sprung kommst du mit zwei Parametern aus: `strecke` und `winkel`.

Damit ist der Entwurf von `jump()` nicht mehr schwer.

Funktion `jump()`:

Parameterliste: `distanz, winkel`

Stift hoch

Um `winkel` nach rechts drehen

`Distanz` Einheiten vorwärts gehen

Um `winkel` nach links drehen

Stift runter

Ran ans Dreiecksmuster

➤ Arbeitet weiter mit `dreieck_arbeit.py`.

Codiere den eben erstellten Entwurf von `jump()` als erste Funktion im Programm.

```
def jump(distanz, winkel):  
    penup()  
    right(winkel)  
    forward(distanz)  
    left(winkel)  
    pendown()
```

Und jetzt gibt's kein Halten mehr. Jetzt wird gesprungen. Um zum Beispiel das Bild vom Anfang des Abschnitts *Dreiecksmuster* zu erhalten, muss der Anweisungsteil unter den Funktionsdefinitionen so aussehen:

5



```
reset()
right(90)

jump(120, -135)
pensize(3)
left(15)
dreierpack(35, 10)

jump(200,30)
pensize(5)
right(60)
dreierpack(75,-15)

jump(240,110)
pensize(7)
left(150)
dreierpack(30,35)

hideturtle()
```

➤ Führe diese Änderungen aus. Sobald das Programm zufriedenstellend funktioniert, speichere eine Kopie des Programms unter dem Namen dreieck13.py ab.

Oder erstelle ein ganz anderes Muster mit vier oder fünf dreierpacks. Spiele einfach mit dem Programm herum, bis du eine Zeichnung hast, die dir echt gefällt.

»jump()« ist auch für später nützlich

jump() ist sicherlich eine so nützliche Funktion, dass wir sie noch öfter brauchen werden. Wir werden daher mit ihr beginnen, einen Werkzeugkasten anzulegen, in den wir in Zukunft alle jene Funktionen hineinschreiben, die wir mit einer import-Anweisung importieren wollen – also eine eigene Software-Werkzeug-Bibliothek.

Gehe dazu wie folgt vor:

➤ Öffne vom IPI aus ein neues Editor-Fenster.

»jump()« ist auch für später nützlich



➤ Schreibe als Kopfkommentar Folgendes in die Datei:

```
# Autor : ...
# Datum : ...
# mytools.py : Bibliothek von selbst erstellten
# brauchbaren Funktionen für die Arbeit mit
# Python für Kids
```

➤ Diese Datei soll jetzt nicht in C:\Py4Kids\kap05, sondern im besonderen Verzeichnis C:\Py4Kids\mylib abgespeichert werden, und zwar mit dem Dateinamen mytools.py! Das Unterverzeichnis mylib – abgekürzt für »my library« = meine Bibliothek – kannst du, wenn es nicht schon vorhanden ist, während des Speicherns erzeugen, indem du das Icon NEUEN ORDNER ERSTELLEN anklickst und den voreingestellten Text Neuer Ordner durch mylib ersetzt.

➤ Kontrolliere sicherheitshalber nochmals, dass die Datei mytools.py im richtigen Ordner gelandet ist!

➤ Schreibe unter den Kopfkommentar von mytools.py die import-Anweisung für das turtle-Modul, denn die folgende Definition von jump() benutzt ja Turtle-Grafik-Funktionen.

➤ Kopiere aus der Datei dreieck_arbeit.py den Code der Funktion jump() über die Zwischenablage in die Datei mytools.py oder schreibe ihn einfach neu dazu!

➤ Speichere mytools.py nochmals!

Damit mytools.py von einer import-Anweisung auf deinem Computer auch gefunden wird, muss das Verzeichnis C:\py4kids\mylib in den Suchpfad von Python aufgenommen werden, und zwar am besten bei jedem Start von Python. Dies kann durch einen Eintrag in der Datei sitecustomize.py sichergestellt werden. Diesen Eintrag nehmen wir jetzt vor. Es ist entscheidend, dass du hier alles genau befolgst:

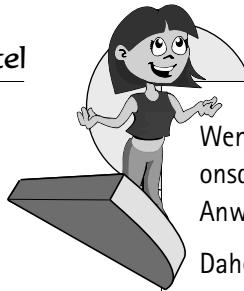
➤ Öffne über den Menüpunkt FILE|OPEN... die Datei c:\Python31\Lib\sitecustomize.py! Wenn sie nicht existiert, öffne ein neues Editor-Fenster.

➤ Stelle sicher, dass am Anfang der Datei die Anweisung import sys steht. Füge ans Ende der Datei folgende Anweisung an:

```
sys.path.append("C:\\\\py4kids\\\\mylib")
```

➤ Speichere die Datei sitecustomize.py im Ordner C:\Python31\Lib ab.

5



Wenn unsere Installation gelungen ist, wird es in Zukunft genügen, Funktionsdefinitionen in die Datei `mytools.py` zu kopieren, um sie per `import`-Anweisung zur Verfügung zu haben.

Daher werden wir uns jetzt überzeugen, dass alles gut gelaufen ist.

➤ Schließe alle offenen IPI- und Editor-Fenster und starte IPI - TURTLE-GRAFIK neu!

➤ Mach mit!

```
>>> from turtle import *
>>> from mytools import jump
```

Wir hätten ebenso `from mytools import *` schreiben können – in diesem Fall macht das gar keinen Unterschied, denn im Modul `mytools.py` ist bisher ohnehin nur diese eine Funktion. Und nun ein paar Anweisungen zum Testen:

```
>>> forward(30)
>>> jump(40, 0)
>>> forward(30)
>>> jump(100,120)
>>> forward(20)
>>>
```

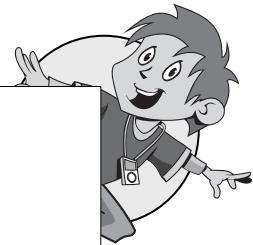
Ich hoffe, es hat auch bei dir geklappt!

Zum Abschluss möchte ich dir hier noch zeigen, wie du feststellen kannst, welche Verzeichnisse im Python-Suchpfad stehen:

```
>>> import sys
>>> sys.path
['C:\\py4kids', 'c:\\Python31\\Lib\\idlelib',
'C:\\WINDOWS\\system32\\python31.zip', 'C:\\Python31\\DLLs',
'C:\\Python31\\lib', 'C:\\Python31\\lib\\plat-win',
'C:\\Python31', 'C:\\Python31\\lib\\site-packages',
'C:\\py4kids\\mylib']>>>
```

Der Pfad für deine Privatbibliothek steht hier auch schon drin. (Je nach Installation kann diese Liste von Verzeichnissen auf deinem Computer von meiner abweichen.)

Nur so zum Drüberstreuen: Hier habe ich `sys` mit einer alternativen Form der `import`-Anweisung importiert. Es ist die `import`-Anweisung ohne `from`. Auch nach *dieser* `import`-Anweisung können alle Objekte aus dem importierten Modul, hier: `sys`, angesprochen werden, doch muss dazu zu jedem Objekt der Modulname angegeben werden. Das Objekt `path` aus dem Modul `sys` wird daher so angesprochen: `sys.path`.



Das geht mit jedem Modul, ist nur nicht immer praktisch:

```
>>> import turtle  
>>> turtle.forward(100)
```

Mit `from turtle import *` sparen wir also sehr viele `turtle` vor unseren Turtle-Grafik-Anweisungen!

Seifenoper

Wir wollen auch diesmal wieder das neu Gelernte in einem Beispiel anwenden, das nicht mit Grafik arbeitet.

Seifenopern verlaufen meistens nach einem gleichbleibenden Muster. Das ist ideal zum Programmieren. Schreiben wir also eine Funktion `seifenoper()`, die uns eine Beschreibung einer Folge einer TV-Serie ausgibt.

Wesentlich für eine Seifenoper ist (abgesehen von der Seifenwerbung), dass es drei Hauptpersonen gibt, die einen Konflikt miteinander haben. Das wollen wir so beschreiben:

Rufen wir

```
>>> seifenoper("Erwin", "Karin", "Kurt", "ein Lottogewinn")
```

auf, dann soll Folgendes ausgegeben werden:

Erwin liebt Karin.

Karin findet Erwin nicht besonders interessant.

Karin ist verrückt nach Kurt.

Erwin mag Kurt gar nicht!

Da verändert ein Lottogewinn alles ...

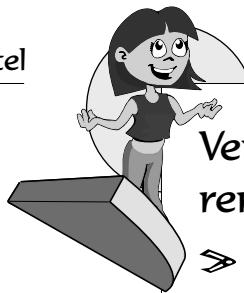
```
>>>
```

Dies sieht nach einer leichten Übung aus: Die Funktion `seifenoper()` hat vier Parameter, gib ihnen zum Beispiel folgende Namen: `mann`, `frau`, `ri-vale`, `ereignis`.

Die gewünschte Ausgabe kann durch fünf `print`-Anweisungen erzeugt werden. Die `print`-Anweisungen müssen Strings ausgeben, in die die Werte der Parameter auf verschiedene Weise eingebaut sind.

Wir wollen die Gelegenheit nutzen, um uns verschiedene Wege anzusehen, wie man das erreichen kann:

5



Verwendung der print()-Funktion mit mehreren Argumenten

⇒ Starte IDLE (PYTHON Gui) und mach mit:

```
>>> mann = "Jens"
>>> frau = "Katja"
>>> print(mann, "liebt", frau)
Jens liebt Katja
>>>
```

Das ist ein Aufruf der `print()`-Funktion mit drei Argumenten. Er schreibt *drei* sehr einfache Ausdrücke an: den Wert der Variablen `mann`, das vorgegebene konstante String-Objekt "liebt" und dann wieder den Wert einer Variablen, diesmal `frau`. Die Werte der Variablen sind ebenfalls Strings.

Der ausgegebene Satz sollte am Ende noch einen Punkt haben. Geben wir also noch den konstanten String ". ." dazu:

```
>>> print(mann, "liebt", frau, ".")
Jens liebt Katja .
>>>
```

Das entspricht nicht ganz unseren Erwartungen. Zwischen Katja und . sollte kein Zwischenraum sein. Aber die `print()`-Funktion schreibt ihre Argumente normalerweise durch Leerzeichen getrennt auf den Bildschirm.

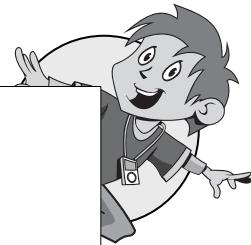
Verkettung von Strings

Python kann Strings auch zusammenfügen beziehungsweise verketten. Dazu verwendet es den + -Operator. Für die Zwischenräume musst du hier selbst sorgen:

```
>>> "eins" + "zwei"
'einszwei'
>>> mann+frau
'JensKatja'
>>> mann + "liebt" + frau
'JensliebtKatja'
>>> mann + " liebt " + frau
'Jens liebt Katja'
>>> mann + " liebt " + frau + "."
'Jens liebt Katja.'
>>>
```

Ein weites Feld für Experimente.

Die String-Verkettung ermöglicht folgende Lösung unseres Problems:



```
>>> print(mann + " liebt " + frau + ".")  
Jens liebt Katja.  
>>>
```

Hier wird die `print()`-Funktion mit nur einem Argument aufgerufen. So mit gibt sie auch nur *einen* Ausdruck aus. Daher enthält der Funktionsaufruf auch keine Kommas. Der Ausdruck allerdings ist komplizierter als die Ausdrücke im ersten Beispiel: Er ist eine Verkettung von vier String-Objekten!

Formatierungsmarken und die `format()`-Methode

Python bietet noch eine weitere elegante Möglichkeit, konstante Strings mit den Werten von Variablen zu mischen: Es verwendet dazu Formatierungsmarken innerhalb von Strings und die Formatierungsmethode `format()`. Formatierungsmarken dienen übrigens auch dazu, Zahlen zu formatieren, doch interessiert uns das hier noch nicht.

In der *einfachsten Form* sind Formatierungsmarken ganze Zahlen, die in geschwungene Klammern eingeschlossen sind, z.B.: `{0}`, `{1}` usw. Die Zahlen beziehen sich auf die Argumente der `format()`-Methode: 0 bezieht sich auf das erste Argument, 1 auf das zweite usw. (Diese Merkwürdigkeit – nämlich dass die Zählung mit 0 beginnt – wird dir später noch häufig begegnen. Und bald auch gar nicht mehr merkwürdig vorkommen.) Bevor ich weitere Erklärungen dazu gebe, zeige ich dir, wie das funktioniert:

➤ Mach weiter mit!

```
>>> "Das ist ein, {0}, Text".format("hick!")  
'Das ist ein, hick!, Text'
```

Beachte, dass nach dem String ein Punkt geschrieben wird, gefolgt von dem Methoden-Namen `format()`. Über den Umgang mit Methoden wirst du im hinteren Teil des Buches noch viel erfahren. Hier nur das Nötigste.

Im obigen Beispiel hätten wir den String "hick!" auch gleich direkt einfügen können:

```
>>> "Das ist ein, hick!, Text"  
'Das ist ein, hick!, Text'
```

Die Verwendung von Formatierungsmarken wird erst richtig nützlich, wenn die Werte von Variablen in einen String eingefügt werden müssen:

5



```
>>> "Heute ist {0} gut drauf!".format(mann)
'Heute ist Jens gut drauf!'
>>> "Kommt morgen {0}?".format(frau)
'Kommt morgen Katja?'
>>> "{0} liebt {1}.".format(mann, frau)
'Jens liebt Katja.'
```

Die `format()`-Methode erzeugt aus dem gegebenen String mit den Formatierungsmarken einen neuen. Der kann natürlich auch als Argument für die `print()`-Funktion verwendet werden:

```
>>> print("{0} liebt {1} nicht.".format(frau, mann))
Katja liebt Jens nicht.
>>> print("{1} liebt {0} nicht.".format(mann, frau))
Katja liebt Jens nicht.
```

Wie erwähnt: 0 bezieht sich immer auf das erste Argument im Aufruf der `format()`-Methode, 1 auf das zweite usw. Die Vorschrift geht so: Wenn in einen konstanten String Werte von Variablen eingesetzt werden sollen, schreibt man an die betreffenden Stellen Formatierungsmarken. Die Werte, die eingesetzt werden sollen, kommen als Argumente in die `format()`-Methode.

```
>>> print("{1} liebt {0} nicht {2}.".format(mann, frau,
                                             "mehr"))
Katja liebt Jens nicht mehr.
>>> print("{0} liebt {1} nicht {2}.".format(mann, frau,
                                             "sehr"))
Jens liebt Katja nicht sehr.
```

Ein letztes Beispiel:

```
>>> rivale = "Kurt"
>>> "Es spielen {0}, {1} und {2} mit!".format(mann, frau,
                                                rivale)
'Es spielen Jens, Katja und Kurt mit!'
>>>
```

So! Jetzt hast du die Wahl! Spiele mit diesen Dingen ein bisschen herum und dann programmiere die Funktion `seifenoper()`.

Erklärende Variablennamen

Ich hatte dir oben vorgeschlagen, die Funktion `seifenoper()` mit folgendem Funktionskopf zu beginnen:

```
def seifenoper(mann, frau, rivale, ereignis):
    ...
```

Zusammenfassung

Meine Idee war, die Namen der Parameter so zu wählen, dass schon beim Lesen des Programmcodes klar wird, welche Rolle die Objekte spielen, auf die diese Namen verweisen. Der Programmcode wird dadurch wenigstens teilweise selbsterklärend und du sparst dir das Einfügen von Kommentaren.

Ich gebe zu, manchmal ist es nicht ganz leicht, Variablennamen treffend zu wählen. Überlege zum Beispiel, welche Argumente an die Parameter von `seifenoper()` übergeben werden müssen, um folgende Ausgabe zu erzeugen:

Doris liebt Paul.

Paul findet Doris nicht besonders interessant.

Paul ist verrückt nach Eva.

Doris mag Eva gar nicht!

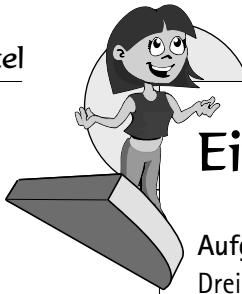
Da verändert eine Krankheit alles ...



Zusammenfassung

- ❖ Funktionen können mit einem oder mehreren Parametern definiert werden.
- ❖ Solche Funktionen müssen dementsprechend mit einem oder mehreren Argumenten aufgerufen werden.
- ❖ Parameter sind lokale Variablen. Beim Ablauf der Funktion verweisen sie auf die Werte, die ihnen beim Aufruf als Argumente übergeben worden sind. Nach der Ausführung der Funktion existieren sie nicht weiter.
- ❖ Du kannst dir eine eigene Software-Werkzeug-Bibliothek mit häufig verwendeten Funktionen erstellen, die du bei Bedarf in dein Programm importieren kannst.
- ❖ Den zugehörigen Suchpfad musst du in der Datei `sitemcustomize.py` eintragen, genauer: an den Wert von `sys.path` anhängen.
- ❖ Strings können mit dem Verkettungsoperator `+` verkettet werden.
- ❖ Sollen in einen Text mehrere Objekte eingefügt werden, kannst du dafür Formatierungsmarken und die `format()`-Methode verwenden.
- ❖ Es empfiehlt sich, Variablennamen selbsterklärend zu wählen.

5



Einige Aufgaben ...

Aufgabe 1: Schreibe eine Funktion `zeichne(figur, laenge)`, die ein Dreieck oder ein Quadrat mit der gewünschten Seitenlänge zeichnet! Ein Aufruf dieser Funktion könnte z. B. lauten: `zeichne("quadrat", 43)`.

Aufgabe 2: Lade die Datei `politiker.py` und stelle das Programm nach Anleitung in den Kommentarzeilen fertig!

Aufgabe 3: Schreibe ein Programm, das den Benutzer auffordert, eine Zahl zwischen 1 und 100 einzugeben und das entsprechende Zahlwort ausgibt!

Tipp: Ab zwanzig setzen sich alle Zahlworte aus den Silben, ein, zwei, drei, vier, fünf, sechs, sieben, acht, neun, und, zwanzig, dreißig, vierzig, fünfzig, sechzig, siebzig, achtzig und neunzig zusammen.

Aufgabe 4: Schreibe eine Funktion, die Texte folgender Art für Urlaubsgrußkarten erzeugt:

_____!

Hier _____ gefällt es mir _____.

Gestern waren wir _____. Das war _____!

Bis bald,

_____.

Die Funktion kann beispielsweise so aufgerufen werden:

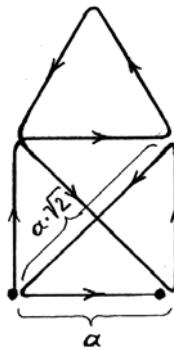
```
gruss( "Lieber Harry", "in Rhodos", "sehr, sehr gut",
      "schwimmen", "traumhaft", "deine Clara-Pythia")
```

Aufgabe 5: (a) Schreibe eine Funktion, `dreieck45(seite)`, die mit Turtle-Grafik ein rechtwinkeliges Dreieck mit zwei gleich langen Seiten zeichnet. Die längere Seite ist »Wurzel aus 2«-mal so lang wie die beiden kürzeren.

(b) Schreibe eine weitere Funktion, `diagquadrat(seite)`, die ein Quadrat mit Diagonalen zeichnet.

(c) Schreibe eine Funktion, `haus(seite)`, in der die Turtle nebenstehendes Haus in einem Zug ohne abzusetzen zeichnet.

(d) Verwende `haus()` und `jump()`, um drei verschieden große Häuser nebeneinander zu zeichnen.





... und einige Fragen

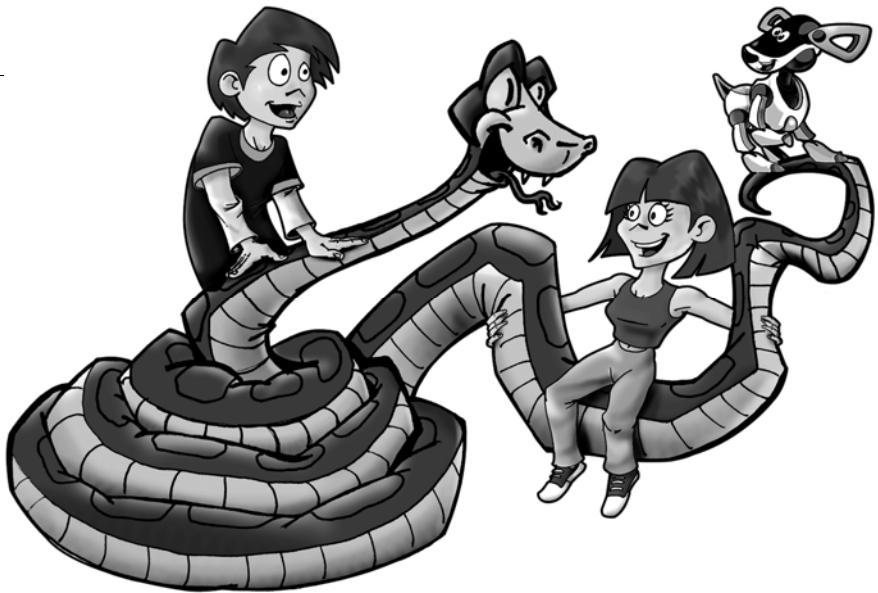
1. Welcher Unterschied ist bei der Verwendung dieser beiden Funktionen zu beachten?

```
def ecke(s):  
    forward(s)  
    right(90)  
def ecke(a):  
    forward(a)  
    right(90)
```

2. Welche der folgenden Anweisungen sind korrekt?

```
print("{0} plus {1} ist {2}".format("eins","zwei","drei"))  
print("{0} plus {1} ist {2}".format("eins","zwei",3))  
print("{0} plus {1} ist {2}".format("eins","zwei","vier"))  
print("{0} plus {1} ist {2}".format ("eins und zwei", "drei"))  
print("{0} plus {1} ist {2}".format ("1 + 2 = 3"))
```


6



Von oben nach unten und zurück

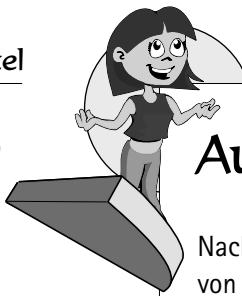
Als Programmierer oder Programmiererin kommst du immer wieder in die Lage, dass dir eine Aufgabe gestellt wird, die dir ganz neu ist. Auf den ersten Blick hast du keine Idee, wie eine Lösung dafür aussehen könnte. Vielleicht ergeht es dir so bei der Vorstellung, das Yinyang-Symbol zu programmieren, das auf der nächsten Seite abgebildet ist.

Das ist aber kein Grund zu verzweifeln. Es gibt nämlich einige allgemeine Verfahren der »Problemlösung«, die man lernen kann. Zwei davon wirst du in diesem Kapitel kennen lernen. Ich werde dir für das Yinyang-Problem zwei Wege zeigen, wie man zu einer Lösung kommen kann: Top-down-Entwurf und Bottom-up-Entwicklung. Es ist ganz wichtig, diese Verfahren zu kennen. Vergiss jedoch nicht: Mindestens ebenso wichtig ist es, sie immer wieder anzuwenden: Programme schreiben, Programme schreiben, Programme schreiben ...

In diesem Kapitel lernst du ...

- ◎ dass es fürs Programmieren verschiedene Verfahren des »Problemlösen« gibt.
- ◎ wie man das Verfahren des Top-down-Entwurfs auf das Yinyang-Problem anwendet.
- ◎ wie man das Yinyang-Problem mit Bottom-up-Entwicklung lösen kann.
- ◎ wie man in Python mit (zu) langen Programmzeilen umgeht.
- ◎ dass Parameter mit Standardwerten versehen werden können.
- ◎ einiges über das Kombinieren von Strings.

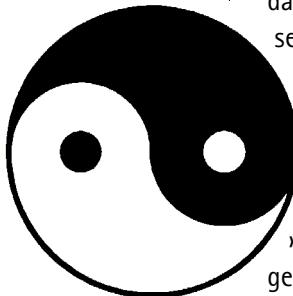
6



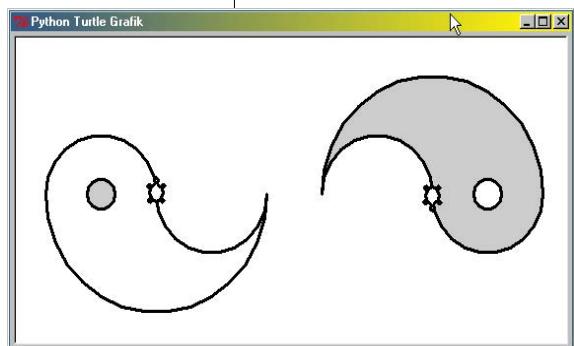
Aufgabenstellung: Yinyang

Nach den Kapiteln 2 bis 5 hast du wahrscheinlich zunächst einmal genug von Dreiecken und Vierecken. Machen wir mal was anderes, was Rundes.

Wir stellen die Aufgabe, ein Yinyang-Symbol zu erzeugen, und zwar so, dass die Größe des Symbols leicht – etwa durch Angabe seines Durchmessers – verändert werden kann. Da bietet es sich an, eine Funktion `yinyang()` zu entwerfen mit `durchmesser` als Parameter.



Wenn du das programmieren willst, musst du das Bild zuerst analysieren. Da fällt zunächst auf, dass der untere, weiße Teil, sagen wir »yin«, eigentlich dieselbe geometrische Figur ist wie der obere, schwarze, »yang«. Nur ist der schwarze um 180° gedreht, also »auf den Kopf gestellt«. Das ist günstig für Turtle-Grafik. Denn um mit Turtle-Grafik zu einer gegebenen Figur eine entsprechende gedrehte zu zeichnen, braucht man nur vor dem Zeichnen die Turtle zu drehen. (Diese Überlegungen haben herzlich wenig mit der Bedeutung von Yin und Yang in der chinesischen Philosophie zu tun. Wenn dich die interessiert, sieh zum Beispiel hier nach: http://de.wikipedia.org/wiki/Yin_und_Yang. Auch ziemlich spannend!)



Ich möchte dir zu dieser Aufgabe zwei unterschiedliche Wege zur Lösung vorstellen. Welchen davon du gehen willst, ist auf der einen Seite eine Frage der Arbeitsweise, die du persönlich bevorzugst. Auf der anderen Seite ist es bei vielen Problemstellungen günstig, beide Wege zu kombinieren. Ich rate dir: Sieh dir beide an. Sollte »Weg 1« zu schwer werden, brich ab. Denke daran, dass »Weg 2« dasselbe Problem von Neuem behandelt, und versuche

diesen. Vielleicht hast du danach Lust, noch mal »Weg 1« zu begehen. Jeder der beiden Wege stellt eine vollständige für sich verständliche Lösung dar.

Weg 1: Top-down. Programmieren ohne Computer

Das Programmier-Verfahren, das ich hier verwende, wird oft als Top-down-Entwurf bezeichnet. Warum und was das Besondere daran ist, erkläre ich dir dann am Ende dieses Abschnitts. Zuerst aber tun wir's.



Wir nehmen uns Bleistift und Papier und schreiben uns auf, welche Schritte zur Lösung des Problems ausgeführt werden müssen. Anfangs werden wir einige Schritte noch nicht im Detail ausführen können. Das macht aber gar nichts. Diese Schritte werden dann in weiterer Folge genauer beschrieben.

Wir versuchen, wie schon in früheren Kapiteln, unsere Entwurfsideen in einer »programmnahen« Sprache zu formulieren. Im ersten Schritt geht es darum, festzuhalten, was `yinyang()` leisten soll:

Programmentwurf: yinyang

Funktionen vom Modul turtle importieren

Funktion `yinyang()`

Parameterliste: `durchmesser`

Strichdicke auf 3 einstellen

`yin zeichnen` (`durchmesser`, Feld weiß, Punkt schwarz)

Turtle um 180 Grad drehen

`yin zeichnen` (`durchmesser`, Feld schwarz, Punkt weiß)

Hauptprogramm

Grafik zurücksetzen

`yinyang` für Durchmesser 200 zeichnen

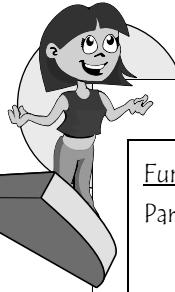
Turtle verstecken

Klar und einfach! Nach der Definition der Funktion `yinyang()` folgen nur noch drei Zeilen, die sozusagen das Hauptprogramm darstellen. Die wichtigste Anweisung in diesem Hauptprogramm ist natürlich der Aufruf der Funktion `yinyang()`. Er bewirkt schließlich das Zeichnen des Yinyang-Symbols.

Du kannst diesen Entwurf sofort korrekt codieren. Doch wäre das Ergebnis kein lauffähiges Programm, denn eine Funktion »`yin zeichnen`« ist noch nicht erklärt. Die Hauptarbeit bleibt also noch zu tun:

Die Funktion »`yin()`«

Wir brauchen eine Funktion `yin()`, die uns sozusagen die Hälfte des Yin-yang-Symbols zeichnen kann. Aus der Zeichnung weiter oben sehen wir, dass `yin()` zwei Teilaufgaben erledigen muss: die Yin-Figur zeichnen und im Mittelpunkt des kleinen Halbkreises einen Punkt einfügen:

Funktion yin():

Parameterliste: durchmesser, feldfarbe, punktfarbe

Yin-Figur in feldfarbe zeichnen

Springe zum Mittelpunkt des Punktes

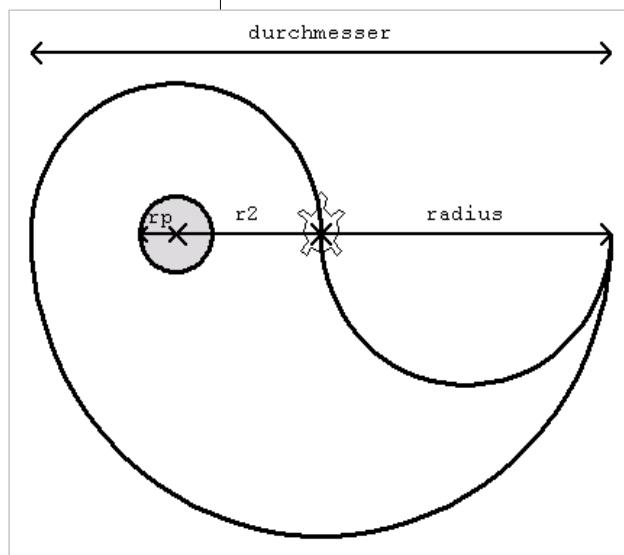
Punkt in punktfarbe zeichnen

Springe zum Mittelpunkt der Yin-Figur zurück

Das ist noch ziemlich ungenau formuliert. Aber es bringt uns weiter. Wir haben drei Teilaufgaben.

- (1) yinfigur() zum Zeichnen der (schwarz oder weiß) gefüllten Form.
- (2) Springen – zum Mittelpunkt und zurück: Dafür haben wir schon jump().
- (3) punkt() soll einen Punkt mit gegebenem Radius zeichnen, wobei die Position der Turtle der Mittelpunkt des Punktes ist.

Die verbleibenden Aufgaben (1) und (3) sind schon recht kleine Aufgaben, aber ganz ohne Nachdenken geht es doch nicht. Betrachten wir zunächst die Größenverhältnisse:



Die ganze Figur besteht aus Kreisen bzw. Halbkreisen in drei verschiedenen Größen. Da wir zum Zeichnen von Kreisen und Kreisbögen die Funktion `circle()` verwenden werden, müssen wir uns die Radien dieser Kreise überlegen. Ausgangspunkt dafür ist, dass wir den **durchmesser** der ganzen Yinyang-Figur als gegeben betrachten.

Aus der Skizze erkennst du: Der **radius** der Figur ist die Hälfte vom **durchmesser**. Der **Radius** der kleinen Halbkreise, r_2 , ist die Hälfte von **radius** und der **Radius** des Punktes, r_p , ist etwa ein Viertel von r_2 . Damit

können wir den Programmentwurf für die Funktion `yin()` schon genauer formulieren. Dazu weisen wir zunächst den Größen `radius`, `r2` und `rp` die passenden Werte zu.



Von mytools jump importieren

Funktion yin():

Parameterliste: durchmesser, feldfarbe, punktfarbe

radius \Rightarrow durchmesser / 2

r2 \Rightarrow radius / 2

rp \Rightarrow r2 / 4

Stiftfarbe schwarz einstellen

Yin-Figur mit radius in feldfarbe zeichnen

Springe Strecke r2 um 90° nach links

Punkt mit Radius rp in punktfarbe zeichnen

Springe Strecke r2 um 90° nach rechts

Jetzt können wir uns an den Entwurf der zwei benötigten Funktionen machen:

Die Funktion »yinfigur()«

(1) **yinfigur()**: Die schwarze Linie, die den Yin-Teil des Yinyang-Symbols begrenzt, besteht aus drei Halbkreisen: *Aus der Sicht der Turtle* sind das: ein »kleiner« Halbkreis nach links und ein »großer« Halbkreis ebenfalls nach links. Dann ist die Turtle an der »Spitze« der Yin-Figur. Dort muss sie sich um 180° drehen und einen weiteren »kleinen« Halbkreis, aber diesmal nach rechts, zeichnen. Danach ist sie an ihrem Ausgangspunkt angelangt.

Funktion yinfigur():

Parameterliste: radius, farbe

radius2 \Rightarrow radius / 2

Füllfarbe farbe einstellen

Füllen beginnen

Halbkreis mit radius2 nach links

Halbkreis mit radius nach links

Turtle um 180 Grad drehen

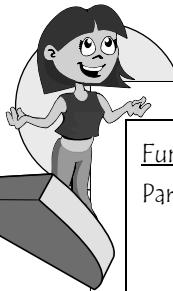
Halbkreis mit radius2 nach rechts

Füllen ausführen

Die Funktion »punkt()«

Jetzt fehlt uns nur noch der Entwurf von `punkt()`. Bei der Gelegenheit können wir die Funktion `jump()` gleich wieder verwenden.

6

Funktion punkt():

Parameterliste: radius, farbe

Springe Strecke radius 90° nach rechts
Stiftfarbe und Füllfarbe farbe einstellen
Füllen beginnen
zeichne kreis mit radius
Füllen ausführen
Springe Strecke radius 90° nach links

Damit ist unser Programmamentwurf fertig und wir können flink ans Kodieren gehen. Das heißt den Programmamentwurf in Python-Code umschreiben. (Oder, wenn wir wollten, in einer anderen Programmiersprache, wenn sie nur ein Turtle-Grafik-Modul bereitstellt.)

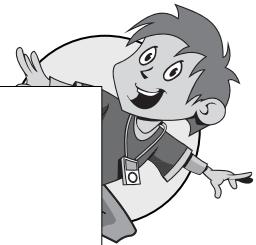
Das vorgeführte Programmier-Verfahren des Top-down-Entwurfs heißt oft auch *Methode der schrittweisen Verfeinerung*. Du beginnst mit einer Beschreibung des Gesamtproblems – in programmnaher Umgangssprache. Man nennt diese Sprache *Pseudocode*. Dann zerlegst du das Gesamtproblem in Teilprobleme, die du dann in weiteren Schritten so lange verfeinerst, bis du auf einer Stufe angelangt bist, die du direkt codieren kannst.

➤ Ich schlage dir vor, das Kodieren – natürlich in Python – zu versuchen. Beginne mit den import-Anweisungen für die Funktionen aus turtle und für jump. Schreibe danach die Funktionsdefinitionen für die Funktionen punkt(), yinfigur() und yin(). Wenn du es in dieser Reihenfolge machst, kannst du jede Funktion gleich nach ihrer Fertigstellung testen: **Strg + S**, **F5** und dann dem IPI einige Aufrufe der neuen Funktion eingeben. Unter die Funktionsdefinitionen ans Ende des Scripts schreibst du die Anweisungen aus dem Programmamentwurf »yinyang«, die schließlich die Grafik zeichnen.

Vielleicht ist dir die Sache aber auch noch etwas zu steil und du möchtest einfach sehen, wie der zugehörige Code aussieht.

Diesen Python-Code findest du auf der Buch-CD im Verzeichnis py4kids-material\kap06\programme als Programm yinyang01.py.

Ich halte es für eine gute Idee, zur Ergänzung jetzt auch noch den Wikipedia-Eintrag zu diesem Thema durchzulesen. Du findest ihn hier: http://de.wikipedia.org/wiki/Top-Down_-und_Bottom-Up-Design. Er führt dich auch gleich zum nächsten Abschnitt:



Weg 2: Bottom-up. Schrittweise von unten nach oben

Diesmal machen wir es umgekehrt. Wir sehen zuerst nach, ob wir an unserer Aufgabenstellung ganz kleine Teilprobleme erkennen können, die wir gleich codieren können. Um Details zu klären, können wir ja den IPI zu Hilfe nehmen.

Beginnen wir damit, die Umrisslinie für den weißen Teil eines Yinyang-Symbols zu zeichnen, das 200 Pixel Durchmesser, also 100 Pixel Radius hat. Er wird von drei Halbkreisen begrenzt, zwei kleinen und einem großen. Du erinnerst dich aus Kapitel 2, dass Halbkreise mit der Funktion `circle()` gezeichnet werden können mit einem `extent` von 180. Du erinnerst dich auch, dass mit positiven Radien Kreise nach links, mit negativen Radien Kreise nach rechts gezeichnet werden.

➤ Starte den IPI-TURTLEGRAFIK neu und mach mit!

```
>>> from turtle import *
>>> reset()
>>> pensize(3)
>>> circle(50, 180)
>>> circle(100, 180)
```

Das waren zwei nach links gedrehte Halbkreise. Sieht schon gut aus. Bevor sie nun den nächsten Halbkreis zeichnet, muss sich die Turtle umdrehen. Oder besser: müssen wir die Turtle umdrehen.

```
>>> left(180)
```

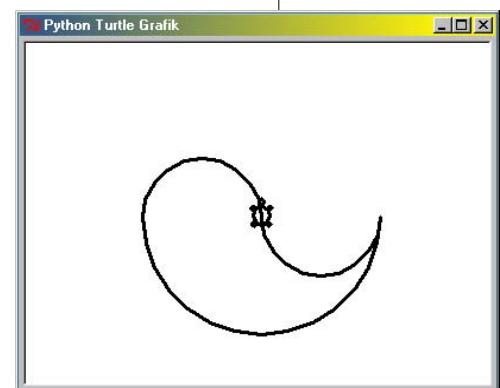
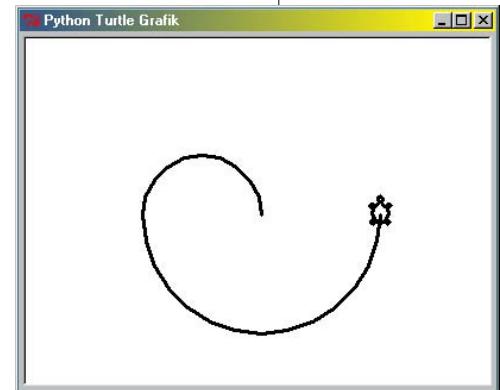
Jetzt Achtung! Der nächste Halbkreis muss nach rechts gedreht werden, also müssen wir einen negativen Radius verwenden:

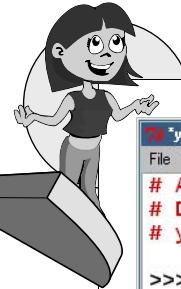
```
>>> circle(-50, 180)
```

Genau, was wir brauchen. Daraus machen wir eine Funktion!

➤ Öffne ein neues Editor-Fenster. Schreibe einen Kopfkommentar und speichere das (entstehende) Programm unter dem Namen `yinyang_arbeit.py` ab.

➤ Weil wir beide, du und ich, nicht so gerne tippen, kopieren wir jetzt die letzten sechs Zeilen aus dem IPI-SHELL-Fenster ins Editor-Fenster.





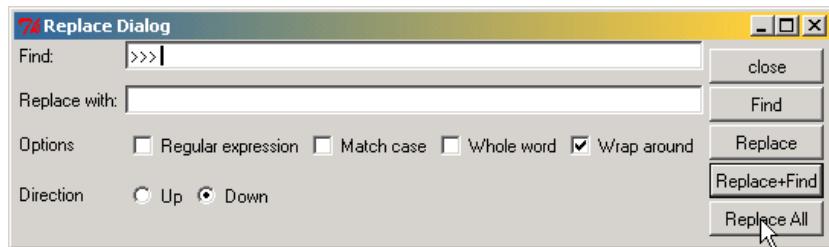
```
*yinyang_arbeit.py - C:/py4kids/kap06/yinyang_arbeit.py*
File Edit Format Run Options Windows Help
# Autor: Gregor Lingl
# Datum: 6. 8. 2009
# yinyang_arbeit.py

>>> from turtle import *
>>> reset()
>>> pensize(3)
>>> circle(50, 180)
>>> circle(100, 180)
>>> left(180)
>>> circle(-50, 180)

Ln: 7 Col: 0
```

Hier stören natürlich die Dreifachpfeile samt Leerzeichen der Prompts, die in einem Programm absolut nichts verloren haben! Aber die sind leicht wegzukriegen.

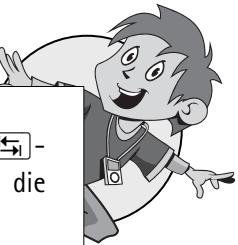
➤ Starte über das Menü EDIT|REPLACE... den REPLACE DIALOG.



- Schreibe in die FIND-Zeile das Python-Prompt, `>>>`, und ein Leerzeichen. Die REPLACE WITH-Zeile lasse leer, denn die obigen vier Zeichen sollen ja durch nichts ersetzt – und damit einfach entfernt – werden. Klicke auf REPLACE ALL. Fertig!.
- Kontrolliere, ob wirklich alle Programmzeilen ganz links beginnen. Wenn sich dort noch Leerzeichen finden, müssen sie weg!
- Speichere und führe das Programm aus. Erstellt es die Yin-Figur richtig?
- Nun wollen wir die letzten vier Anweisungen zu einer Funktion zusammenfassen. Füge vor dem ersten `circle()`-Aufruf zwei Leerzeilen ein. In die zweite schreibe die Kopfzeile einer Funktionsdefinition für die Funktion – sagen wir: `yin()`.

Die vier Zeilen danach bilden den Funktionskörper und müssen eingerückt werden. Das geht am schnellsten so:

Weg 2: Bottom-up. Schrittweise von unten nach oben



- » Markiere diese vier Zeilen mit der Maus und drücke auf die - Taste. Verschiebe zuletzt noch die `reset()`-Anweisung unter die Funktionsdefinition. Das Programm sieht jetzt so aus:

```
from turtle import *

def yin():
    circle(50, 180)
    circle(100, 180)
    left(180)
    circle(-50, 180)

reset()
```

Ist dir klar, warum `reset()` *keinesfalls* im Funktionskörper von `yin()` stehen darf? Wenn nicht, probiere aus, was dann (bei den folgenden »Mach mit!«-Anweisungen) passiert!

- » Speichere das Programm und führe es erneut aus.

Nun hat `reset()` die alte Grafik gelöscht und der IPI sollte die neue Funktion `yin()` kennen. Das wollen wir gleich ausprobieren.

- » Mach mit!

```
>>> yin()
>>> left(180)
>>> yin()
>>> left(180)
```

Das sieht ja schon recht vielversprechend aus. (Die letzte `left()`-Anweisung hat die Turtle wieder in die ursprüngliche Richtung gestellt.)

Unsere Funktion `yin()` muss aber jetzt noch verbessert werden. (1) Sie soll das Symbol in verschiedenen Größen zeichnen können. Und (2) muss die Yin-Figur auch mit einer Farbe gefüllt werden können. Unternehmen wir zuerst die Verbesserung (1) und verpassen wir `yin()` einen Parameter `radius`. Wir müssen aber daran denken, dass die kleinen Halbkreise nur mit dem halb so großen Radius gezeichnet werden dürfen. Also:

- » Setze im Funktionskopf von `yin()` den Parameter `radius` ein. Ersetze in den `circle()`-Aufrufen 100 durch `radius` und 50 durch `radius / 2`. So sieht dann das Ergebnis aus:

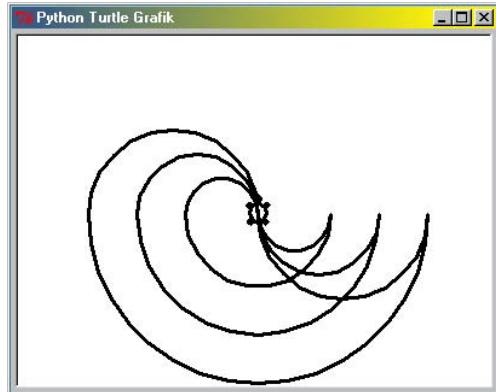


```
def yin(radius):
    circle(radius / 2, 180)
    circle(radius, 180)
    left(180)
    circle(-radius / 2, 180)
```

- Speichere das Programm, führe es aus und teste es mit dem IPI:

```
>>> yin(140)
>>> yin(100)
>>> yin(60)
```

Um nun unsere Yin-Figur auch noch färben zu können, gehen wir wie schon bei den Dreiecken vor:

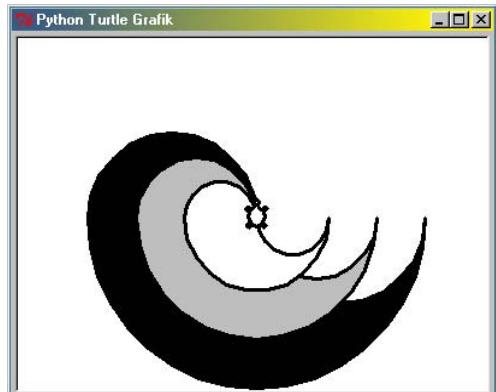


- Setze im Funktionskopf von yin() einen weiteren Parameter farbe ein. Im Funktionskörper muss dann die Füllfarbe auf farbe gesetzt werden und an den passenden Stellen die Anweisungen für das Beenden beziehungsweise Ausführen des Füllens eingefügt werden.

```
def yin(radius, farbe):
    fillcolor(farbe)
    begin_fill()
    circle(radius / 2, 180)
    circle(radius, 180)
    left(180)
    circle(-radius / 2, 180)
    end_fill()
```

- Speichere das Programm, führe es aus und teste es mit dem IPI:

```
>>> yin(140, "black")
>>> yin(100, "gray")
>>> yin(60, "white")
```



Weg 2: Bottom-up. Schrittweise von unten nach oben



Sieht super aus und verlockt – entsprechend dem oben gemachten Experiment –, gleich eine Funktion `yinyang()` zu schreiben:

➤ Füge nach der Definition von `yin()` folgende Funktionsdefinition ein:

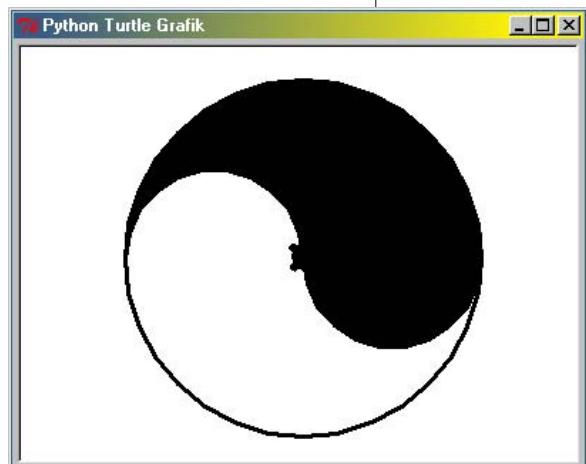
```
def yinyang(radius):
    pensize(3)  # für einen stärkeren Rand!
    yin(radius, "white")
    left(180)
    yin(radius, "black")
    left(180)
```

➤ Nach dem `reset()`-Aufruf füge einen Aufruf von `yinyang()` zum Beispiel mit dem Radius 125 ein. Speichere das Programm und führe es aus.

Da fehlt doch noch was! Ja, jedes Yin braucht noch einen fetten Punkt – einen kleinen Kreis – im Mittelpunkt des kleinen Halbkreises, und zwar in der Gegenfarbe, das heißt der Farbe vom anderen Yin. Sieht auch wieder aus wie Arbeit:

➤ Importiere zunächst `jump` aus `mytools`.

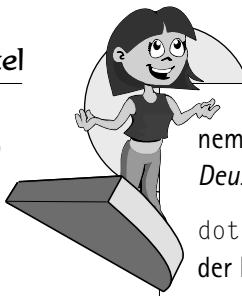
➤ Jetzt müssen wir `yin()` erweitern. Zunächst soll die Turtle zum Mittelpunkt des kleinen Halbkreises springen: unter einem Winkel von 90° nach links gerade so weit, wie der Radius des kleinen Kreises ist, und später dann wieder zurück:



```
def yin(radius, farbe):
    fillcolor(farbe)
    ...
    end_fill()
    jump(radius / 2, -90)
    ## ??? Punkt zeichnen ???
    jump(radius / 2, 90)
```

Aber wie zeichnen wir den Punkt? Hast du eine Idee? (Wenn du den »Weg 1« durchgearbeitet hast, hast du sicher eine Idee!) Doch auch in ei-

6



nem Informatik-Buch kann einmal Folgendes geschehen: Es erscheint ein *Deus ex machina*, die Funktion `dot()` aus dem Turtle-Grafik-Modul: `dot(size, color)` zeichnet einen Punkt mit dem Durchmesser `size` in der Farbe `color`.

(Siehe auch: http://de.wikipedia.org/wiki/Deus_ex_machina)

Der Durchmesser des Punktes dürfte etwa *ein Viertel* so groß sein wie der Durchmesser des kleinen Kreises, `radius`. Also versuchen wir `dot(radius / 4, ???)`. Und bemerken, dass dazu `yin()` die Gegenfarbe kennen muss. Das geht nur mit einem weiteren Parameter:

```
def yin(radius, farbe, gegenfarbe):
    fillcolor(farbe)
    ...
    end_fill()
    jump(radius / 2, -90)
    dot(radius / 4, gegenfarbe)
    jump(radius / 2, 90)
```

Und erfordert auch eine Änderung von `yinyang()`:

```
def yinyang(radius):
    pensize(3) # für einen stärkeren Rand!
    yin(radius, "white", "black")
    left(180)
    yin(radius, "black", "white")
    left(180)
```

- Füge diese Änderungen in den Code von `yinyang_arbeit.py` ein. Speichere ab, teste das Programm und bessere etwaige Fehler aus.
- Abschlussarbeiten: Kopfkommentar? Doc-Strings? Speichere eine Kopie des Programms als `yinyang02.py` ab.

Du hast jetzt schon eine ganze Menge Erfahrung im Gebrauch von Funktionen mit Parametern. Hier ist eine gute Gelegenheit sie anzuwenden:

- Versuche nun, die Funktion `yinyang()` so umzuschreiben, dass sie vier Parameter hat:

```
def yinyang(radius, farbe, gegenfarbe, strichdicke):
    # ... (Funktionskörper)
```

Weg 2: Bottom-up. Schrittweise von unten nach oben

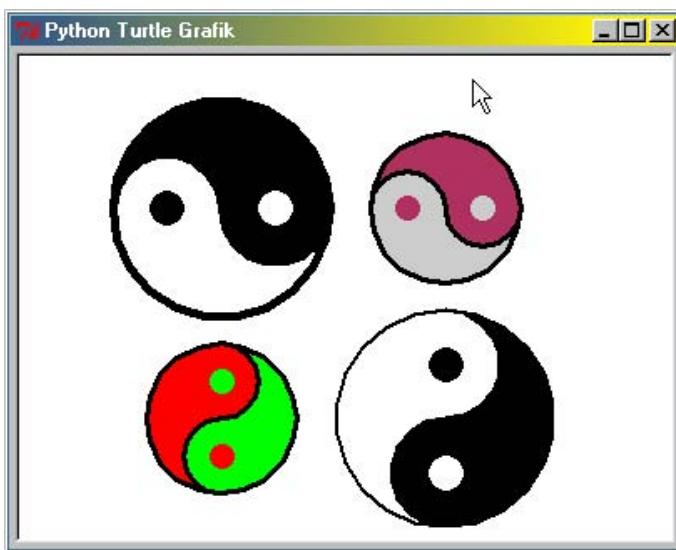


Am Ende sollte sie mit der Anweisungsfolge

```
reset()
jump(80,45)
yinyang(45, "gray80", "maroon", 3)
jump(126,180)
right(90)
yinyang(65, "white", "black", 2)
jump(-134, 0)
yinyang(45, "red", "green", 3)
left(90)
jump(126,0)
yinyang(65, "white", "black", 5)

hideturtle()
```

diese Grafik erzeugen:



- Wenn dein Programm fertig ist, speichere eine Kopie des Programms unter dem Namen `yinyang03.py` ab.

Nach all dieser Mühe willst du vielleicht dein Ergebnis mit meinem Programm vergleichen, das obige Grafik erzeugt hat. Daher hier das vollständige Listing (ohne Kopfkommentar):

6



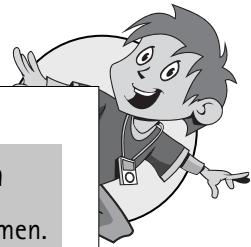
```
from turtle import *
from mytools import jump

def yin(radius, farbe, gegenfarbe):
    fillcolor(farbe)
    begin_fill()
    circle(radius / 2, 180)
    circle(radius, 180)
    left(180)
    circle(-radius / 2, 180)
    end_fill()
    jump(radius / 2, -90)
    dot(radius / 3, gegenfarbe)
    jump(radius / 2, 90)

def yinyang(radius, farbe, gegenfarbe, strichdicke):
    pensize(strichdicke)
    yin(radius, farbe, gegenfarbe)
    left(180)
    yin(radius, gegenfarbe, farbe)
    left(180)

    reset()
    jump(80, 45)
    yinyang(45, "gray80", "maroon", 3)
    jump(126, 180)
    right(90)
    yinyang(65, "white", "black", 2)
    jump(-134, 0)
    yinyang(45, "red", "green", 3)
    left(90)
    jump(126, 0)
    yinyang(65, "white", "black", 5)

hideturtle()
```



Clara Pythias Python-Special: Lange Zeilen aufteilen

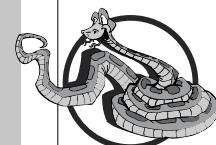
Im Folgenden wirst du es mit langen Programmzeilen zu tun bekommen. In Python muss man mit der Zeileneinteilung und den Einrückungen sehr genau sein. Da »zu lange« Zeilen jedoch häufig vorkommen, ist es nützlich, sich folgende Regel zu merken:

Programmtext, der zwischen einem Paar von Klammern steht, kann auf mehrere Zeilen aufgeteilt werden. Du kennst dieses Verfahren schon, weil wir es bei der `print()` Funktion mehrfach verwendet haben. Nach dem Öffnen einer Klammer weiß der Python-Interpreter, dass die Zeile logisch erst nach dem Schließen der Klammer zu Ende ist.

Auch das kann man mit dem IPI ausprobieren – ich tu's hier nicht mit einer Parameterliste, sondern mit arithmetischen Ausdrücken:

Mach mit:

```
>>> (1 + 2 + 3 + 4 +
   5 + 6)
21
>>> (1 + 2 + 3 + 4 +
   5 + 6) * 10
210
>>> (1 + 2 + 3 + 4 + 5 + 6) *
SyntaxError: invalid syntax
```



Hier ist eine Zeile offenbar unvollständig. Es gibt aber keine offene Klammer, die noch zu schließen wäre. Dagegen geht:

```
>>> (1+
 2+
 3+
 4+
 5)
15
>>> (1 + 2 +
 3) * (7 -
      5)
12
>>>
```

Es ist zweifellos sinnvoll, von dieser Möglichkeit nur dort Gebrauch zu machen, wo dadurch die Lesbarkeit von Programmen verbessert wird!



Clara Pythias Python-Special: Standardwerte für Parameter und Schlüsselwort-Argumente

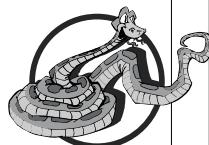
In Python kann man Parametern einer Funktion einen Standardwert (Default-Wert) zuweisen. Dies geschieht bei der Definition der Funktion. Für unsere Funktion `yinyang()` kann das so aussehen:

```
def yinyang(radius, farbe="white",
            gegenfarbe="black", strichdicke=3):
    pensize(strichdicke)
    ...
    ...
```

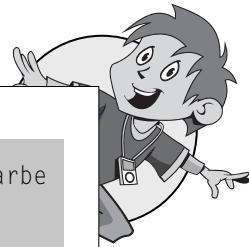
Um zu verstehen, was die Einführung von Standardwerten bringt, ist es günstig, damit zu experimentieren:

- Lade das Programm `yinyang_arbeit.py`!
- Kommentiere alle Anweisungen aus, die *nach* der `reset()`-Anweisung stehen. Das geht so: Markiere sie mit der Maus und wähle den Menüpunkt FORMAT|COMMENT OUT REGION oder die Tastenkombination **Alt**+**3**! Auskommentierte Anweisungen erscheinen rot und werden bei der Ausführung des Programms ignoriert. Sie können nach Bedarf wieder »entkommentiert« werden.
- Ändere in der Definition von `yinyang()` den Funktionskopf nach der oben stehenden Vorlage ab!
- Speichere das Programm und führe es aus! Eine leere Turtle-Grafik-Zeichenfläche erscheint!
- Mach mit! – und betrachte jeweils das Grafik-Fenster.

```
>>> yinyang(50)
>>> jump(120, -90)
>>> yinyang(30, "white", "blue", 1)
>>> jump(240,90)
>>> yinyang(40, "red", 5)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    yinyang(40, "red", 5)
...
TG_Error: bad color arguments: 5
```



Weg 2: Bottom-up. Schrittweise von unten nach oben



Hier wurde der Wert 5 ungünstigerweise dem dritten Parameter gegenfarbe zugewiesen. Die Zahl 5 beschreibt aber keine Farbe. Daher: Fehler!

Bei dieser Schreibweise entscheidet die Position eines Arguments darüber, welchem Parameter es zugewiesen wird. Man nennt derartige Argumente daher Positionsargumente.

➤ Mach weiter mit!

```
>>> jump(90,0)
>>> yinyang(40,"red", "black", 5)
>>> jump(200,-90)
>>> yinyang(50,strichdicke=1)
>>> jump(-180, 0)
>>> yinyang(40, strichdicke=8, gegenfarbe="green")
```

Erklärung: Im ersten dieser drei Aufrufe von `yinyang()` traten nur Positionsargumente auf. Für die gegenfarbe wird der Standardwert "black" verwendet. Der Standardwert für strichdicke wird mit 5 überschrieben.

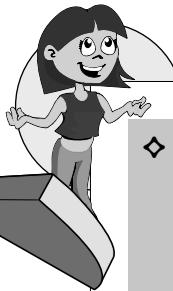
Im zweiten werden für beide Farben die Standardwerte verwendet. Für die strichdicke wird der Wert 1 an Stelle des Standardwertes verwendet: Man sagt: strichdicke wird als *Schlüsselwort-Argument* gebraucht. Das geschieht in Form einer Wertzuweisung an den Parameter strichdicke, die in die Argumentliste geschrieben wird. Deshalb wird 1 nicht als zweites Argument eingesetzt, sondern als Argument für den Parameter strichdicke.

Im letzten Aufruf werden gar zwei Argumente als Schlüsselwert-Argumente verwendet. Dabei siehst du, dass sie nicht einmal in derselben Reihenfolge auftreten müssen wie in der Funktionsdefinition. Welcher Argument-Wert zu welchem Parameter gehört, ist dabei ohnehin eindeutig.

Welche Regeln sind für die Verwendung von Standardwerten beziehungsweise Schlüsselwort-Argumenten zu beachten?

- ❖ Die Standardwerte werden im Funktionskopf den Parametern zugewiesen. Dies geschieht nur einmal, nämlich wenn die Funktionsdefinition ausgeführt wird, und *nicht* bei jedem Funktionsaufruf!
- ❖ Bei der Funktionsdefinition: *Alle Parameter mit Standardwerten müssen nach den Parametern ohne Standardwerte stehen.* (farbe, gegenfarbe und strichdicke stehen nach dem Parameter radius.)

6



- ❖ Beim Aufruf einer Funktion müssen den Parametern mit Standardwerten keine Argumente übergeben werden. In unserem Beispiel sind also zum Beispiel folgende Aufrufe erlaubt:

```
yinyang(75)
yinyang(12, "black", "red")
```

Der erste Aufruf zeichnet ein weiß/schwarzes Standard-Yinyang-Symbol mit Radius 75. Der zweite ein schwarz/rotes. Ein übergebenes Argument überschreibt den Standardwert.

- ❖ Beim Aufruf einer Funktion müssen zuerst alle Positionsargumente angeführt werden. Für sie entscheidet die Position in der Argumentliste, an welche Parameter sie übergeben werden. Soll ein Parameter ein Positionsargument erhalten, dann müssen auch alle links davon stehenden Parameter ein Positionsargument erhalten. Für unser Beispiel heißt das: Wenn an den vierten Parameter strichdicke ein Positionsargument übergeben wird (etwa um einen dickeren Rand zu erhalten), dann müssen auch der zweite und dritte Parameter ein Positionsargument erhalten, selbst dann, wenn du die Standardwerte für sie verwenden willst:

```
>>> yinyang(50, "white", "black", 9)
```

- ❖ Nach den Positionsargumenten können weitere Argumente als Schlüsselwort-Argumente übergeben werden. Dabei brauchen Parameter mit Standardwert dann nicht angeführt zu werden, wenn der Standardwert Verwendung finden soll. Beispiel:

```
>>> yinyang(50, strichdicke=9)
```

Zusammenfassend ergeben sich folgende beiden Muster:

Muster 8: Definition einer Funktion mit Parametern, einige davon mit Standardwerten

```
def Funktionsname(param1, param2, ...,
                  sparam1=wert1, sparam2=wert2, ...):
```

Anweisung 1

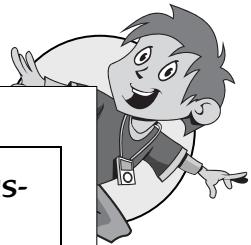
Anweisung 2

...

Funktionskörper

Anweisungen verwenden die lokalen Namen
param1, param2, ..., sparam1, ...

In der Parameterliste kommen die Parameter mit Standardwerten hinter den Parametern ohne Standardwerte.



Muster 9: Funktionsaufruf mit Positions- und Schlüsselwort-Argumenten

Positionsargumente,
Reihenfolge wie in der Funktionsdefinition

Funktionsname(pos-arg1, pos-arg2, ...,
param_s1=sw-arg1, param_s2=sw-arg2, ...):

...

Schlüsselwort-Argumente,
beliebige Reihenfolge

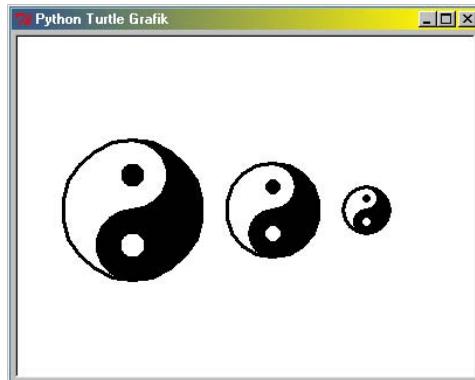
In der Argumentliste kommen zuerst Positionsargumente entsprechend der Reihenfolge der Parameter in der Funktionsdefinition. Danach können Schlüsselwort-Argumente in beliebiger Reihenfolge eingesetzt werden.

»jump()«, revisited

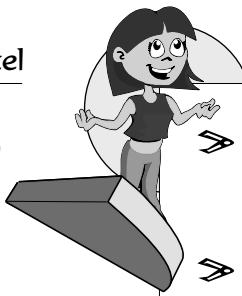
Wir können jetzt unsere Funktion `jump()` etwas bequemer verwendbar machen. Es wird ja oft so sein, dass die Turtle um eine gewisse Strecke geradeaus weiter springen soll.

➤ Falls du inzwischen IPI-TURTLEGRAFIK neu gestartet hast, lade nochmals `yinyang_arbeit.py` und führe es aus. Dann weiter mit dem IPI :

```
>>> right(90)
>>> jump(-100,0)
>>> yinyang(60)
>>> jump(120,0)
>>> yinyang(40)
>>> jump(80,0)
>>> yinyang(20)
>>> hideturtle()
```



6



- Es kommt wahrscheinlich recht oft vor, dass die Turtle nur vorwärts springen muss. Dann hat `winkel` den Wert 0. Es wird daher sinnvoll sein, dem Parameter `winkel` den Standardwert 0 zuzuordnen.
- Ändere in der Datei `c:\py4kids\mylib\mytools.py` die Definition der Funktion `jump` so ab, dass der Parameter `winkel` den Standardwert 0 erhält – für einfacheres Vorwärtshüpfen!
- Entkommentiere die Anweisungen am Ende des Programms `yinyang_arbeit.py` wieder: mit der Maus markieren und Menüpunkt **FORMAT|UNCOMMENT REGION** oder **Alt+4** anwenden. Speichere die Datei und schließe alle Python-Fenster.



Wenn du eine bereits importierte Funktion änderst, wie es im obigen Beispiel mit `jump()` der Fall war, kannst du sie nicht mit einer weiteren `import`-Anweisung nochmals importieren. Jede Funktion wird vom Interpreter nur ein Mal importiert und er merkt sich dann, dass sie schon da ist.

Um mit veränderten Bibliotheksmodulen weiterzuarbeiten, muss Python neu gestartet werden.

- Schließe alle Python-Fenster und starte IPI-TURTLEGRAFIK neu.
- Lade `yinyang_arbeit.py` (**FILE|RECENT FILES**), Führe es aus. Es muss nach wie vor fehlerlos laufen.
- Ändere in der drittletzten Zeile `jump(126,0)` auf `jump(126)` ab. Auch dies muss jetzt wie gehabt funktionieren.
- Füge nach der `reset()`-Anweisung die folgenden beiden Anweisungen ein:

```
hideturtle()
speed(0)
```

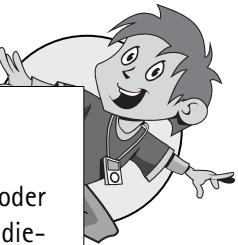
- Beobachte, wie das Programm nun schneller abläuft.

Zusammenfassung

- ❖ Funktionen können mit einem oder mehreren Parameter(n) definiert werden.
Solche Funktionen müssen dementsprechend mit einem oder mehreren Argumenten aufgerufen werden.

Einige Aufgaben ...

- ❖ Programme können mit verschiedenen Verfahren entwickelt werden.
- ❖ Ein Programm-Entwicklungsverfahren ist der Top-down-Entwurf oder die Methode der schrittweisen Verfeinerung. Bei ihm findet das Kodieren als letzter Schritt statt.
- ❖ Ein anderes Verfahren ist die Bottom-up-Methode. Sie beginnt mit dem Kodieren kleiner Teilaufgaben, die dann schrittweise zur Gesamtlösung zusammengesetzt werden.
- ❖ Programmtext, der zwischen Klammern steht, kann auf mehrere Zeilen aufgeteilt werden.
- ❖ Parameter können Standardwerte erhalten.
- ❖ Die Parameter mit Standardwerten müssen hinter denen ohne Standardwerte stehen.
- ❖ Argumente können als Schlüsselwort-Argumente verwendet werden. Schlüsselwort-Argumente müssen in der Argumentliste hinter den Positionsargumenten stehen.
- ❖ Turtle-Grafik: Die Funktion `dot(groesse, farbe)` zeichnet einen Punkt mit dem Durchmesser `groesse` in der Farbe `farbe`.

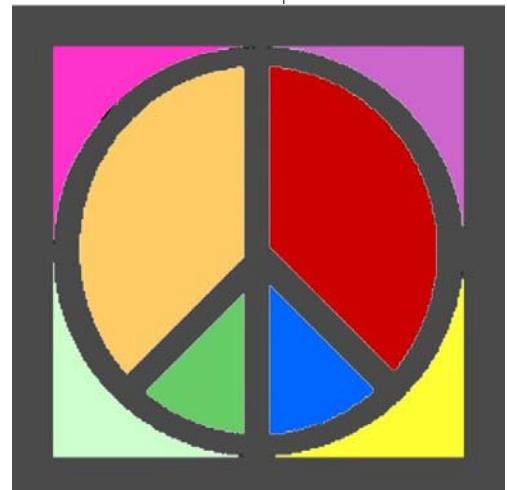


Einige Aufgaben ...

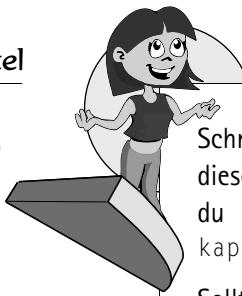
Aufgabe 1: Schreibe ein Bibliotheksmodul `myfigures.py` mit Funktionen für die Erstellung von Quadraten, gleichseitigen Dreiecken, Rechtecken oder auch von Kreisen mit Mittelpunkt am aktuellen Turtle-Standort und von Kreissektoren! Speichere es so ab, dass du anschließend die Möglichkeit hast, diese Funktionen in andere Programme zu importieren.

Aufgabe 2: Schreibe eine neue Version des Programms `radioaktiv.py`, (Kapitel 2, Aufgabe 4). Verwende diesmal Funktionen mit Parametern und schreibe das Programm so, dass mit einer Funktion `radioaktiv()` verschiedene große Radioaktivitätswarnzeichen erzeugt werden können.

Aufgabe 3: In den letzten Jahren tauchte immer wieder an den unterschiedlichsten Orten und in verschiedensten Formen nebenstehendes Symbol auf:



6



Schreibe ein Programm, das diese Grafik erzeugt. Neben Schwarz gibt es in diesem Symbol acht verschiedene Farben. Wie die genau aussehen, siehst du auf der Buch-CD: multi_peace_logo.jpg im Pfad py4kids\kap06\.

Solltest du bei der Bearbeitung von Aufgabe 1 oder Aufgabe 2 eine Funktion `sektor(radius,winkel)` geschrieben haben, wirst du diese hier vorteilhaft verwenden können.

Mehr über dieses Symbol findest du unter
<http://de.wikipedia.org/wiki/Friedenszeichen>.

... und einige Fragen

1. Welche Funktionsköpfe sind korrekt?

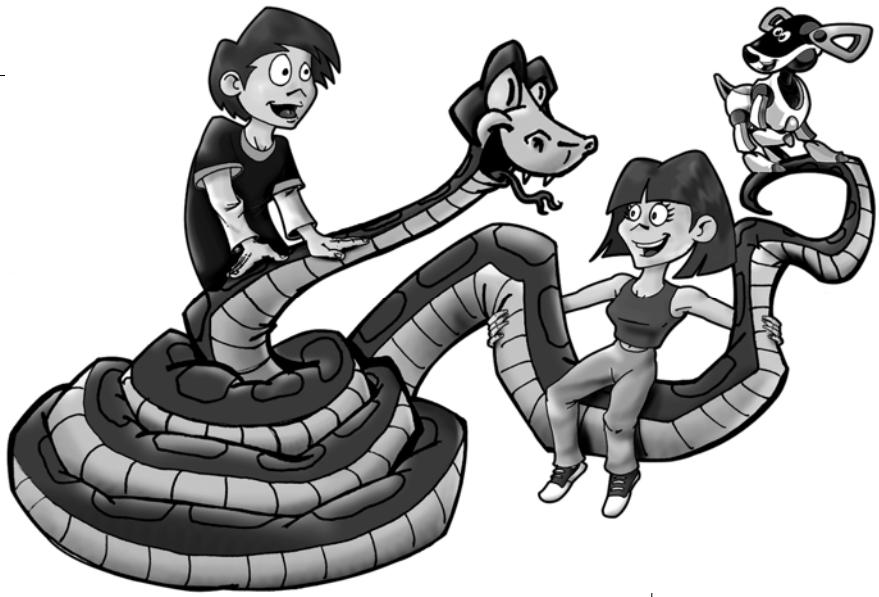
```
def figur(x, y, z):  
def figur(x, y=10, z=""):  
def figur(x=7, y, z):
```

2. Ist Folgendes ein korrekter Python-Ausdruck?

```
>>> (3+  
4)*5  
+6)
```

3. Wenn ja, was ergibt dann seine Auswertung?
4. Du hast eine Folge von Anweisungen aus dem Python-SHELL-Fenster mitsamt den >>>-Prompts in ein Editor-Fenster der IDLE kopiert, weil die entsprechenden Anweisungen Teil eines Programms werden sollen. Was musst du nun tun?

7



Schleifen, die zählen

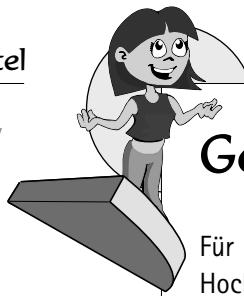
Du hattest in den letzten Kapiteln immer wieder Aufgaben zu lösen, bei denen sich bestimmte Anweisungsfolgen wiederholt haben. Manchmal war es richtig öde, diese Anweisungsfolgen im IDLE-Editor zu kopieren. Außerdem hatten wir auf diese Weise auch nicht die Möglichkeit, die Anzahl der Wiederholungen variabel zu gestalten.

Python bietet, wie alle Programmiersprachen, die Möglichkeit, den Programmablauf so zu kontrollieren, dass Anweisungen oder Anweisungsböcke wiederholt werden können. Diese Form der Programmablaufkontrolle nennt man *Schleife*.

In diesem Kapitel lernst du ...

- ◎ wie man mit Python Schleifen programmiert.
- ◎ einiges über Tupel und `range()` – Objekte, statische und dynamische Wertevorräte..
- ◎ wie man mit Python Tupel entpacken kann.
- ◎ wie man unser Quizprogramm flexibler gestaltet, indem man Daten und Programmablauf voneinander trennt.

7

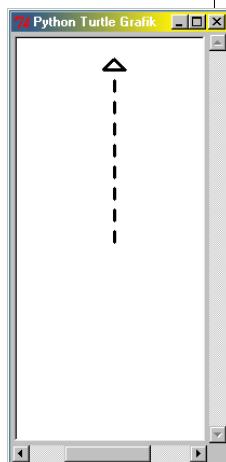


Gestrichelte Linien

Für die folgende interaktive Übung ist es ganz nützlich, ein Fenster im Hochformat zu haben. Um die Größe des Turtle-Grafik-Fensters zu bestimmen, gibt es im Modul `turtle` die Funktion `setup(breite, hoehe)`. Ganz leicht auszuprobieren.

➤ Starte IPI-TURTLEGRAFIK und mach mit!

```
>>> from turtle import *
>>> setup(200,400)
>>> shape("arrow")
>>> pensize(3)
```



Ein einfacher Vorgang, bei dem immer wieder dasselbe wiederholt werden muss, ist das Zeichnen gestrichelter Linien. Wir sehen uns das Arbeiten mit Schleifen an diesem Beispiel an:

➤ Da hier offenbar unsere Turtle ziemlich viel springen muss, importiere auch die Funktion `jump` aus `mytools.py`!

```
>>> from mytools import jump
>>> forward(10)
>>> jump(10)      # und diese zwei Anweisungen
>>> ...           # ein paar Mal
```

Eine gestrichelte Linie, in Handarbeit erzeugt.

So entsteht tatsächlich eine gestrichelte Linie. Aber wie können wir diese Wiederholungen automatisieren?

➤ Mach weiter mit:

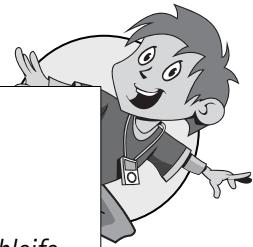
```
>>> reset(); pensize(3)
>>> for i in (1, 2, 3):
    forward(10)
    jump(10)
```

Die Turtle hat drei Striche gezeichnet!

```
>>> reset()
>>> for i in (1, 2, 3, 4, 5):
    forward(10)
    jump(10)
```

Die Turtle hat fünf Striche gezeichnet – und fünf Zwischenräume übersprungen. Eigentlich sollte am Ende noch ein Strich hin:

```
>>> forward(10)
```



Die Zählschleife mit »for«

Was du hier erstmals ausprobiert hast, ist eine so genannte **for-Schleife**. Bevor ich dir erkläre, wie die **for**-Schleife aufgebaut ist und wie sie funktioniert, noch ein ganz kurzes und noch einfacheres Beispiel im IPI:

➤ Mach weiter mit:

```
>>> for i in (1, 2, 3, 4):
    print("*")

*
```

Die **print**-Anweisung wurde vier Mal ausgeführt. Ebenso, aber mit dem optionalen Schlüsselwort-Argument `end=" "`, damit die Ausgabe in einer Zeile erfolgt:

```
>>> for i in (1, 2, 3, 4):
    print(i, end=" ")
```

1 2 3 4

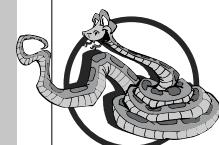
Die **print**-Anweisung wurde vier Mal ausgeführt. Da die **Schleifenvariable** `i` in der **print**-Anweisung ausgegeben wird, sieht man, dass sich ihr Wert bei jedem Schleifendurchgang ändert.

Clara Pythias Python-Special: Die Schlüsselwort-Argumente der `print()`-Funktion

Du weißt schon, dass die `print()`-Funktion eine beliebige Anzahl von Positionsargumenten übernimmt. Im obigen Beispiel wurde ihr auch das Schlüsselwort-Argument `end=" "` übergeben.

Die `print()`-Funktion versteht aber auch drei Schlüsselwort-Argumente, die – wie stets – nach den Positionsargumenten übergeben werden müssen.

Sehen wir uns zunächst das Schlüsselwort-Argument `end` mit dem IPI an (eigentlich haben wir damit ja schon oben begonnen):



7



➤ Mach mit!

```
>>> For wort in ("a", "be", "bu"):
    print(wort)

a
be
bu

>>> for wort in ("a", "be", "bu"):
    print(wort, end=" -+-
")
a -+- be -+- bu -+-

>>> for wort in ("a", "be", "bu"):
    print(wort, end=" "+chr(9786)+"\n")

a ☺
be ☺
bu ☺
```

In allen Beispielen werden drei Aufrufe der `print()`-Funktion ausgeführt, jede mit einem Positionsargument `wort` und dem Schlüsselwort-Argument `end`. Jede schreibt das `wort` auf den Bildschirm und danach den Wert des Parameters `end`. Im letzten Beispiel ist dieser Wert ein String, der aus einem Leerzeichen, dem lustigen *Unicode*-Zeichen mit der Nummer 9786 und einem newline-Zeichen "`\n`" zusammengesetzt ist (mehr dazu auf Seite 289). Daher beginnt nach jeder `print`-Anweisung eine neue Zeile. Wenn `end` nicht angegeben wird, wird der Standardwert "`\n`" verwendet und die Zeile somit beendet. Wird `end=" "` verwendet, dann fügt jede `print`-Anweisung am Ende ein Leerzeichen ein.

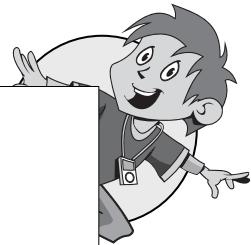
Das Schlüsselwort-Argument `sep` wird verwendet, um ein Trennzeichen für eine `print`-Anweisung mit mehreren Positionsargumenten festzulegen:

```
>>> print("a", "be", "bu")
a be bu
>>> print("a", "be", "bu", sep="?")
a?be?bu
>>> print("a", "be", "bu", sep=" ", ...ähhh... ")
a, ...ähhh... be, ...ähhh... bu
```

Wie du leicht erkennen kannst, hat auch `sep` einen Standardwert, nämlich das Leerzeichen " ".

Die `print()`-Funktion akzeptiert auch noch das Schlüsselwort-Argument `file`. Es wird in diesem Buch nicht benutzt.

Die Zählschleife mit »for«



Wir experimentieren weiter mit for:

```
>>> for element in (1, 2, 3, 4):  
    print(element, end=" ")
```

1 2 3 4

Das ändert offenbar an der Ausführung nichts. Wir haben nur einen anderen Namen für die Schleifenvariable gewählt. Aber:

```
>>> for element in (1, 2, 1, 2):  
    print(element, end=" ")
```

1, 2, 1, 2

Hier werden dem Namen `element` bei den vier Schleifendurchgängen abwechselnd die Werte 1 und 2 zugewiesen.

Die for-Schleife ist so wichtig, dass ich dir genauere Erklärungen geben muss.

Die for-Schleife hat folgende Struktur:

for *ding* in *Wertevorrat*:

Anweisungsblock

Zunächst erkennst du wieder: Die for-Schleife hat einen *Schleifenkopf* und einen *Schleifenkörper*. Der Schleifenkörper ist, wie immer eingekückt, ein Block von Anweisungen, die wiederholt ausgeführt werden.

Im Schleifenkopf spielt der *Wertevorrat* eine wichtige Rolle. Er liefert die Werte, für die der Schleifenkörper ausgeführt werden soll. *Wertevorrat* kann in Python vieles sein, was eine Folge von Werten liefert. Bisher haben wir dafür nur Folgen von Werten kennen gelernt, die direkt hingeschrieben werden. Sehr häufig besteht der Wertevorrat aus einem `range()`-Objekt. Diese und andere Möglichkeiten für *Wertevorrat* wirst du im Weiteren noch kennen lernen.

In unseren ersten Beispielen war der Wertevorrat eine Folge von Zahlenwerten oder Strings. Genau so gut können an Stelle von Zahlen oder Strings auch andere Objekte verwendet werden. Sie sind durch Kommas voneinander getrennt und in runde Klammern eingeschlossen. Eine solche Folge heißt in Python ein *Tupel*.

ding ist in der einfachsten und häufigsten Form der for-Schleife ein Name.

Die Ausführung der Schleife geschieht in der folgenden Weise: Dem Namen *ding* wird das erste Element des *Wertevorrats* zugewiesen und





dann wird der Anweisungsblock ausgeführt. Dann wird `ding` das zweite Element des Wertevorrats zugewiesen und der Anweisungsblock neuerlich ausgeführt. Und so weiter, mit allen Elementen von Wertevorrat.

Unabhängig davon, ob `ding` bei der Ausführung des Schleifenkörpers eine Rolle spielt oder nicht, wird die Schleife so oft ausgeführt, wie Elemente in `Wertevorrat` vorhanden sind.

Im Weiteren werden `for`-Schleifen in vielen Beispielen auftreten. Auch wenn dir diese Erklärungen ein bisschen abstrakt vorgekommen sind, wirst du mit ihnen bald recht vertraut sein.

Wir haben jetzt noch ein Problem: Was ist, wenn wir eine Schleife mit 20 oder 100 oder noch mehr Wiederholungen brauchen? Sollen wir ein Tupel mit 100 Elementen direkt anschreiben? Oft weißt du ja beim Schreiben eines Programms noch gar nicht, wie oft eine Schleife ausgeführt werden soll.

Bei einer Funktion, die eine gestrichelte Linie zeichnen soll, werden wir die Anzahl der Wiederholungen – das ist die Anzahl der Striche – je nach Länge der Linie wählen.

Python hat eine Funktion eingebaut, die einen Wertevorrat vorgegebener Länge erzeugt, die Funktion `range()`. Das englische Wort »range« bedeutet so viel wie Bereich. Diese Funktion ist für unsere Zwecke gut zu gebrauchen.

Faule Typen

Python hat einige spezielle Typen von Objekten eingebaut, die die Elemente eines Wertevorrats erzeugen. Aber nicht statisch alle auf einmal, sondern dynamisch eines nach dem anderen immer erst dann, wenn es gebraucht wird. Im Englischen wird diese Form der Erzeugung von Werten als »lazy evaluation« bezeichnet. Solche Objekte sind also gewissermaßen »faule Typen«, die die Arbeit so lange aufschieben, bis es nicht mehr anders geht. Weil das aber nicht so super sympathisch klingt, will ich diese Typen lieber als »dynamischen Wertevorrat« bezeichnen.

Ich will dir kurz erklären, wozu das gut ist: Stelle dir vor, eine `for`-Schleife soll im Bereich der natürlichen Zahlen von 0 bis 999999 die erste finden, die eine bestimmte Eigenschaft hat. (Zum Beispiel, dass sie auf zwei verschiedene Arten als Summe von zwei dritten Potenzen darstellbar ist.) Nun könnte Python eine Liste dieser Zahlen erzeugen – immerhin eine Million



Zahlen – und diese dann auf die genannte Eigenschaft prüfen. Da die erste Zahl dieser Art aber 1729 ist, wären alle Zahlen ab 1730 – und das sind immerhin 988270 Zahlen – für nichts und wieder nichts erzeugt worden. Dafür wäre eine Menge Speicherplatz und auch Rechenzeit vergeudet worden. Daher ist es viel günstiger, einen dynamischen Wertevorrat zu verwenden.

Python stellt dafür den `range()`-Typ zur Verfügung. Er arbeitet viel ökonomischer, denn selbst wenn alle Werte gebraucht werden, müssen sie nicht alle auf einmal erzeugt werden und brauchen somit viel weniger Speicher.

Wir wollen diesen dynamischen `range()`-Typ nun ein bisschen untersuchen. In den folgenden Beispielen verwenden wir natürlich nur kleine Wertebereiche.

➤ Mach mit:

Wir sehen uns das mit dem IPI an:

➤ Mach mit!

```
>>> range(3)
range(0, 3)
>>> range(10)
range(0, 10)
```

Das ist nicht besonders erhellend. Aber was will man von einem dynamischen Wertevorrat? Die Werte sind eben noch nicht da! Erst wenn man sie braucht! Zum Beispiel in einer `for`-Schleife:

```
>>> for i in range(3):
    print(i)
```

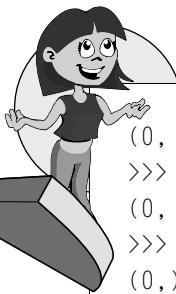
```
0
1
2
```

Da haben wir's!

Wenn diese Bereiche nicht zu groß sind, können wir Python dazu bringen, aus dem dynamischen einen statischen Wertevorrat zu machen, sprich zum Beispiel ein Tupel. Dafür gibt es die eingebaute Funktion `tuple()`. Sie erzeugt aus einem `range()`-Objekt ein Tupel, also ein fixes »statisches« Objekt:

```
>>> tuple(range(3))
(0, 1, 2)
>>> tuple(range(7))
```

7



```
(0, 1, 2, 3, 4, 5, 6)
>>> tuple(range(15))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14)
>>> tuple(range(1))
(0,)
>>> tuple(range(0))
()
```

Du siehst, `tuple(range(zahl))` gibt eine Folge ganzer Zahlen aus. Und wir erkennen, dass `range(zahl)` immer `zahl` Elemente enthält. Und zwar, von 0 ausgehend, alle ganzen Zahlen, die kleiner als `zahl` sind. Das größte Element von `range(zahl)` ist `zahl-1`.



Merke dir: Du kannst nachsehen, welche Elemente in einem dynamischen Wertevorrat, z. B. einem `range()`-Objekt enthalten sind, indem du aus seinen Werten ein statisches Tupel machst:

```
>>> range(5, 20, 4)
range(5, 20, 4)
>>> tuple(range(5, 20, 4))
(5, 9, 13, 17)
```

Anmerkung 1: In Python gibt es auch noch andere statische Objekte, zum Beispiel Listen.

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

Auf den ersten Blick kein berauschender Unterschied. Warte nur ab! Mit Listen wirst du bald mehr zu tun bekommen und dabei lernen, dass sie wenigstens nicht ganz so statisch sind wie Tupel.

Anmerkung 2: Folgendes ist mir schon länger ein Anliegen:

```
>>> print
<built-in function print>
>>> tuple
<class 'tuple'>
>>> list, int, float
(<class 'list'>, <class 'int'>, <class 'float'>)
```

Genau genommen sind `tuple()`, `float()` usw. im Gegensatz zu `print()` keine Funktionen, sondern sie *erzeugen* Objekte, die zu bestimmten Klassen gehören. Für dich macht das aber bei der praktischen Arbeit keinen Unterschied. Also: Vergiss es – und verwende sie weiter wie Funktionen.



Weiter mit der for-Schleife ...

In einer for-Schleife über `range(n)` wird der Schleifenkörper daher immer n Mal ausgeführt:

```
>>> for nummer in range(5):
    print(numero, end=" ")
```

0 1 2 3 4

```
>>> for nummer in range(10):
    print("*", end=" ")
```

* * * * *

Im weiteren Verlauf des Buches wirst du gelegentlich auch noch mit anderen dynamischen Wertevorräten in Berührung kommen:

... und zurück zur gestrichelten Linie

Bevor wir eine Funktion programmieren, die die Turtle gestrichelte Linien zeichnen lässt, müssen wir noch zwei Entscheidungen treffen:

Erstens müssen wir uns einen Namen für die Funktion ausdenken. Da sie gestrichelte Linien zeichnet, ist vielleicht `strichel()` ein passender Name.

Zweitens müssen wir entscheiden, welche Größen als Argumente an die Funktion übergeben werden sollen?

Klar ist, dass die Länge der Linie ein Argument sein muss. Außerdem spielt aber noch die Anzahl der Striche eine Rolle. Es ist daher naheliegend, dafür einen Parameter `striche` einzuführen.

Die gestrichelte Linie soll mit einem Strich beginnen und mit einem Strich enden. Die Anzahl der Zwischenräume zwischen den Strichen nennen wir spruenge. Sie ist um 1 kleiner als die Anzahl der Striche. Wenn wir die Striche und die Zwischenräume gleich lang machen wollen, können wir die Strichlänge leicht so berechnen:

Strichlänge = Länge der Linie / (Anzahl der Striche + Anzahl der Sprünge)

7



Dann muss die Turtle spruenge Mal zeichnen und springen. Wir haben hier eine Schleife mit gegebener Anzahl von Wiederholungen. Anschließend muss sie noch einen letzten Strich zeichnen.

Wir können diese Idee so aufschreiben:

Funktion strichel():

Parameterliste: laenge, striche = 10 # Standardwert

spruenge \Rightarrow striche - 1

strich \Rightarrow laenge / (striche + spruenge)

Wiederhole spruenge Mal:

Turtle um strich vorwärts bewegen

Turtle um strich vorwärts springen lassen

Turtle um strich vorwärts bewegen

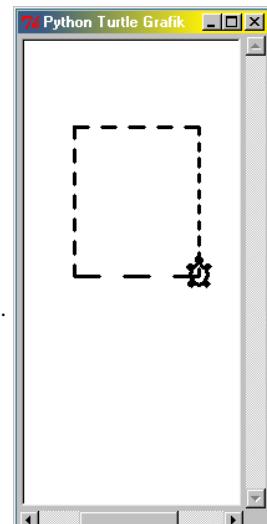
Um eine Schleife mit gegebener Anzahl von Wiederholungen zu programmieren, verwenden wir eine `for`-Schleife mit `range()`, wie in der letzten IPI-Übung:

```
for i in range(spruenge):
    Anweisungen
```

- Kodiere nun die Funktion `strichel()` in einer Datei `strichel.py`.
- Teste sie im IPI, etwa so:

```
>>> reset(); pensize(3)
>>> jump(50,90)
>>> strichel(120)
>>> left(90)
>>> strichel(100, 5)
>>> left(90)
>>> strichel(120,7)      So arbeitet strichel().
>>> left(90)
>>> strichel(100, 3)
>>> left(90)
```

- Die Funktion `strichel()` sieht nützlich aus. Übernehmen wir sie in die Werkzeugdatei `mytools.py`. Vielleicht willst du sie wieder einmal verwenden.
- Öffne mit der IDLE das Modul `C:\py4kids\mylib\mytools.py`



Dreiecke mit Schleife



- »> Schreibe (oder kopiere) den Code von strichel() als letzte Funktionsdefinition in die Datei oder kopiere ihn über die Zwischenablage hinein!
- »> Speichere mytools.py!

Bevor wir uns dem nächsten Problem zuwenden, fassen wir die Verwendung von `for`-Schleifen als Zählschleifen noch einmal zusammen:

Muster 10: »for«-Schleife als Zählschleife:

Zweck: eine Folge von Anweisungen n Mal wiederholen.

Im Programmentwurf:

Wiederhole n Mal:

Anweisung 1

Anweisung 2

...

Schleifenkopf

Im Programmcode:

for i in range(n):

Anweisung1

Anweisung2

...

Schleifenkörper

Dreiecke mit Schleife

In früheren Kapiteln haben wir Programme geschrieben, in denen verschiedene Anweisungsfolgen wiederholt vorgekommen sind.

Es ist eine gute Übung für die Anwendung von `for`-Schleifen, diese Programme nochmals herzunehmen und zu sehen, wo sich der Code durch Anwendung von Schleifen verkürzen lässt. Tun wir das hier nochmals mit Dreiecken.

- »> Lade das Programm `dreieck_arbeit.py` aus dem Verzeichnis `py4kids\kap07`.

7



Hier ist der Code:

```
from turtle import *

def dreieck(laenge, farbe):
    """Zeichne Dreieck mit Seitenlänge laenge.
    Dreieckseiten werden in farbe gezeichnet.
    """
    pencolor(farbe)
    forward(laenge)
    left(120)
    forward(laenge)
    left(120)
    forward(laenge)
    left(120)

    reset()
    pensize(10)

    dreieck(65, "red")
    left(120)

    dreieck(100, "green")
    left(120)

    dreieck(135, "blue")
    left(120)

hideturtle()
```

➤ Gleich in der Funktion dreieck() haben wir zwei Anweisungen, die drei Mal wiederholt werden:

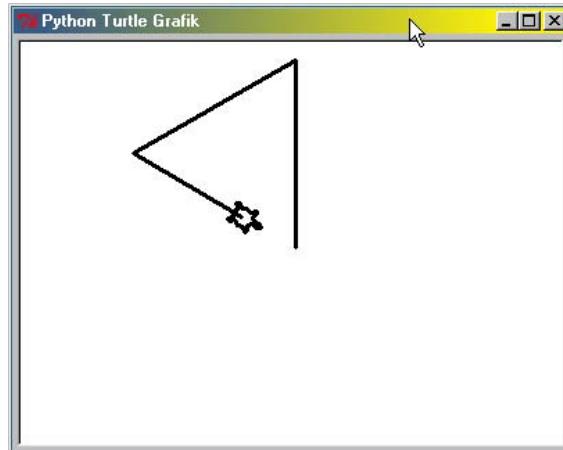
```
forward(laenge)
left(120)
```

Wie das in einer for-Schleife unterzubringen ist, können wir mit dem IPI direkt ausprobieren.

➤ Starte den IPI neu und mach mit!

```
>>> from turtle import *
>>> reset(); pensize(3)
>>> for i in range(3):
        forward(135)
        left(120)
```

Dreiecke mit Schleife



Die Turtle bei der Ausführung der `for`-Schleife.

Das klappt! Also bauen wir das in unser Programm ein.

- Ändere den Code der Funktion `dreieck()`. Die sechs `forward()`/`left()`-Anweisungen sind durch die drei Zeilen der `for`-Schleife, wie im IPI ausprobiert, zu ersetzen. Natürlich muss als Argument im Aufruf von `forward()` der Name `laenge` an Stelle des Wertes 135 eingesetzt werden.
- Speichere das Programm ab und führe es aus!

Der Code der Funktion `dreieck()` sollte jetzt so aussehen:

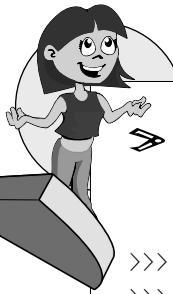
```
def dreieck(laenge, farbe):  
    pencolor(farbe)  
    for i in range(3):  
        forward(laenge)  
        left(120)
```

Die sechs Anweisungen vor dem `hideturtle()`-Aufruf stellen auch eine Wiederholung dar. Der Aufruf von `dreieck()` und der Aufruf von `left()` werden drei Mal wiederholt. Doch ändern sich die Größe von `seite` und die Farbe.

Wir können bei jedem Schleifendurchlauf leicht *eine* der beiden Größen ändern:

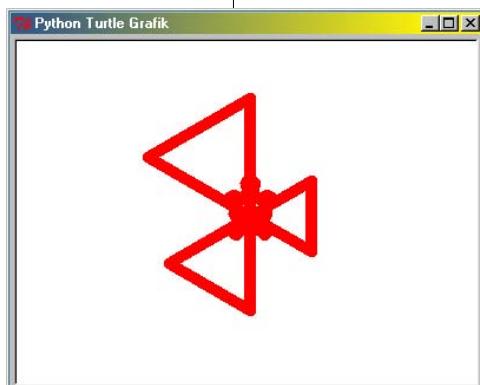
- Mach mit!

7



➤ Überprüfe, dass der IPI die Funktion dreieck() kennt. Das ist sicher der Fall, wenn du eben einen Testlauf von dreieck_arbeit.py ausführst hast:

```
>>> reset()
>>> dreieck( 100, "red")
```

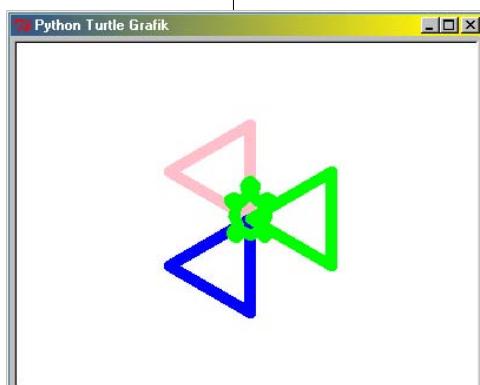


Wenn das funktioniert hat, können wir mit dreieck() experimentieren:

Zunächst zeichnen wir Dreiecke mit unterschiedlichen Seitenlängen:

```
>>> reset(); pensize(10)
>>> for seite in (100, 80, 60):
    dreieck(seite, "red")
    left(120)
```

Jetzt versuchen wir es mit unterschiedlichen Farben:



```
>>> clear()
>>> for farbe in ("pink", "blue", "green"):
    dreieck(80, farbe)
    left(120)
```

Es erweist sich als sehr praktisch, dass eine Liste auch Strings als Elemente enthalten kann.

Doch stellt sich nun die Frage, wie wir es anstellen, dass bei jedem Schleifendurchgang beides, seite und farbe, geändert wird. Dafür gibt es mehrere Lösungen. Die einfachste ist wohl, wie wir es früher

bei Quadraten gemacht haben, den Wert von seite jedes Mal durch eine Zuweisung zu ändern:

➤ Mach weiter mit!

Wir geben der Variablen seite zuerst einen Wert:

```
>>> clear()
>>> seite = 100
```

Dann schreiben wir eine for-Schleife für die verschiedenen Farben, in der wir zusätzlich seite ändern:

Die Funktion »len()«

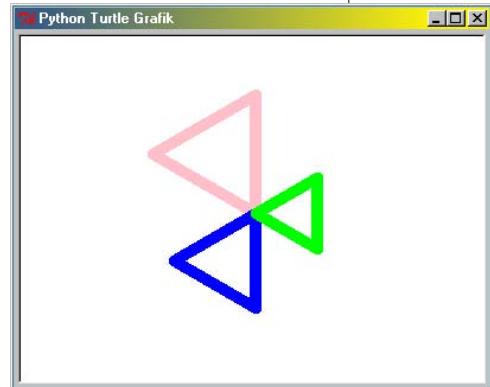


- » Hole mit dem Cursor oder mit **Alt+P** die Schleife in die Eingabezeile! Achte darauf, in `dreieck()` das erste Argument 80 auf `seite` abzuändern.

```
>>> for farbe in ("pink", "blue", "green"):  
    dreieck(seite, farbe)  
    left(120)  
    seite = seite - 20
```

Das Ergebnis entspricht unseren Vorstellungen.

- » Ersetze nun in `dreieck_arbeit.py` die letzten sechs Zeilen durch eine `for`-Schleife entsprechend unserer IPI-Übung! Speichere eine Kopie des Programms als `dreieck14.py` ab.



Die Funktion »len()«

Wir haben jetzt im Programm fünf Zeilen statt sechs, und einfacher ist es sicherlich auch nicht geworden. Was bringt's also?

Es ist wieder einmal flexibler geworden. Ich werde dir gleich zeigen, wie ich das meine. Vorher überzeugen wir uns mit dem IPI davon, dass Tupel auch Dinge, »Objekte«, sind, denen man einen Namen geben kann:

- » Mach mit:

```
>>> farben = ("pink", "blue", "green")  
>>> farben  
('pink', 'blue', 'green')  
>>> for farbe in farben:  
    print(farbe, end=" ")
```

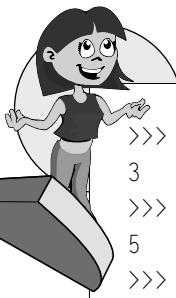
```
pink blue green
```

```
>>>
```

Python hat eine eingebaute Funktion, die von Tupeln (und anderen Sammlungen von Dingen) die Länge ermittelt und zurückgibt: die Funktion `len()`.

- » Mach weiter mit!

7



```
>>> len(farben)
3
>>> len( (4,7,9,13,-4) )
5
>>> len( ('a','be','bu','und','raus','bist','du') )
7
>>> len(())
0
```

Wenn man die Länge eines Tupels für irgendwelche Berechnungen braucht, kann man auch ihr einen Namen geben:

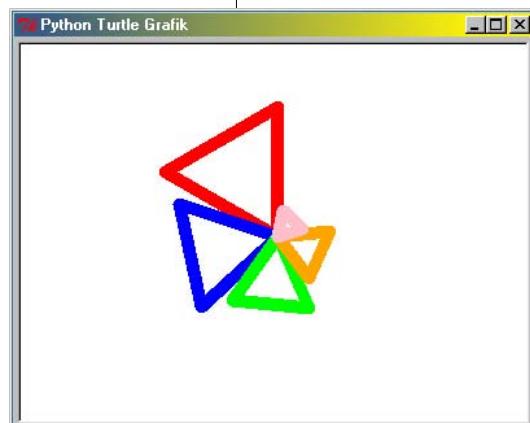
```
>>> anzahl=len(('a','be','bu','und','raus','bist','du'))
>>> anzahl
7
```

Mit diesen Mitteln ausgerüstet ist es ein Leichtes für uns, die Schleife in dreieck_arbeit.py so umzuschreiben, dass sie mehrere bunte Dreiecke zeichnet:

➤ Ändere dreieck_arbeit.py so ab:

```
seite = 100
farben = ("red", "blue", "green", "orange", "pink")
anzahl = len(farben)
for farbe in farben:
    dreieck(seite, farbe)
    left(360.0 / anzahl)
    seite = seite - 20
```

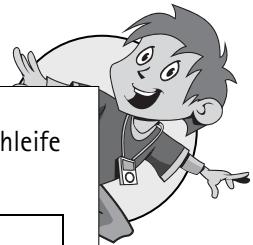
➤ Speichere das Programm und führe einen Testlauf aus!



Du kannst nun verschiedene Dreieck-Folgen zeichnen, indem du einfach die Liste der Farben abänderst, andere Farben hineinschreibst, welche hinzufügst oder entfernst. Vielleicht möchtest du auch für den Wert 20 in der letzten Zeile einen Namen, z.B. aenderung, einführen, wie wir es in anderen Fassungen des Dreiecksprogramms schon gemacht haben. Mit verschiedenen Werten von seite und aenderung stehen dann weitere Variationsmöglichkeiten zur Verfügung.

➤ Speichere eine Kopie dieses Programms als dreieck15.py ab.

Eine Schleife für gefüllte Dreiecke



In den letzten beiden Abschnitten hat sich gezeigt, dass die `for`-Schleife vielseitiger verwendbar ist als für bloßes Zählen:

Muster 11: Allgemeine for-Schleife:

Zweck: eine Folge von Anweisungen für jedes Element aus einem Wertevorrat wiederholen.

Im Programmentwurf:

Wiederhole für jedes element aus Wertevorrat:

Anweisung 1

Anweisung 2

...

Schleifenkopf

Im Programmcode:

```
for element in wertevorrat:  
    Anweisung1  
    Anweisung2  
    ...
```

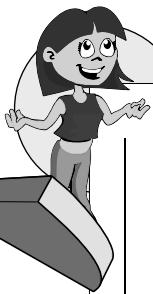
Schleifenkörper

Eine Schleife für gefüllte Dreiecke

Eines der letzten Dreiecksprogramme aus Kapitel 5 zeichnete Dreierpacks von Dreiecken. Wenn wir auch dort die Funktion `dreieck()` mit einer `for`-Schleife codieren, erhalten wir folgenden Code, der als `dreierpack_arbeit.py` im Verzeichnis `py4kids\kap07` enthalten ist:

```
from turtle import *  
  
def dreieck(laenge):  
    """Zeichne Dreieck mit Seitenlänge seitenlaenge.  
    """  
  
    for i in range(3):  
        forward(laenge)  
        left(120)  
  
def fuelle_dreieck(seitenlaenge, stiftfarbe, fuellfarbe):  
    """Zeichne gefülltes Dreieck mit Seitenlänge seitenlaenge.  
    Umrandung in der stiftfarbe, gefüllt mit der fuellfarbe.  
    """
```

7



```
color(stiftfarbe, fuellfarbe)
begin_fill()
dreieck(seitenlaenge)
end_fill()

def dreierpack(seite, aenderung):
    """Zeichne Muster aus drei Dreiecken.
    Erstes Dreieck hat Seitenlaenge seite.
    """
    fuelle_dreieck(seite, "red", "cyan")
    left(120)
    seite = seite + aenderung

    fuelle_dreieck(seite, "green", "magenta")
    left(120)
    seite = seite + aenderung

    fuelle_dreieck(seite, "blue", "yellow")
    left(120)
    seite = seite + aenderung

reset()
pensize(10)
right(90)
dreierpack(65, 35)
hideturtle()
```

Am Ende des Anweisungsblocks der Funktion `dreierpack()` steht hier noch einmal die Anweisung zur Änderung der `seite`, obwohl das keinerlei Auswirkung auf das Ergebnis des Programmablaufes hat. Es macht aber deutlich, dass in `dreierpack()` wieder drei Mal drei gleiche Anweisungen vorkommen. Kann man das nicht auch mit einer Schleife lösen?

Kann man! Und zwar mit einer `for`-Schleife, die über ein Tupel von *Farbenpaaren* läuft. Farbenpaare brauchen wir deshalb, weil hier immer eine Stiftfarbe und eine Füllfarbe zusammengehören.

Tupel eignen sich gut, um solche Paare zu bilden und somit erhalten wir ein Tupel von Tupeln – genauer: ein Dreier-Tupel von Zweier-Tupeln.

➤ Mach mit!

Eine Schleife für gefüllte Dreiecke

```
>>> farbenpaare = (("red","cyan"), ("green","magenta"),
                   ("blue","yellow"))
>>> for paar in farbenpaare:
    print(paar)

('red', 'cyan')
('green', 'magenta')
('blue', 'yellow')
```

Jetzt müssen wir aber die einzelnen Farben aus jedem Paar herauskriegen, um sie in `fuelle_dreieck()` als Argumente einsetzen zu können. Dafür hat Python ein spezielles Verfahren:

Clara Pythias Python-Special: Tupel entpacken

Python bietet die Möglichkeit, ein Tupel in einer besonderen Form der Zuweisungsanweisung auszupacken, das heißt, in seine Bestandteile zu zerlegen.

Du kannst mit dem IPI studieren, wie diese Dinge funktionieren. Zunächst probieren wir das Tupel-Auspicken aus:

» Mach mit!

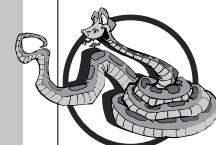
```
>>> name = ("Charles", "Babbage")
```

Beachte genau die Tupel-Schreibweise: zwischen runden Klammern mehrere Objekte, durch Kommas getrennt!

```
>>> vorname, zuname = name
>>> vorname
'Charles'
>>> zuname
'Babbage'
```

Die Objekte können auch von verschiedenem Typ sein, zum Beispiel Zahlen und Strings gemischt:

```
>>> datum = (11, "September", 2001)
>>> tag, monat, jahr = datum
>>> tag
11
>>> monat
'September'
>>> jahr
2001
>>>
```



7



Dieses Auspacken von Tupeln verwenden wir nun für unsere Farbenpaare:

```
>>> for paar in farbenpaare:
    stiftfarbe, fuellfarbe = paar
    print(stiftfarbe, "-", fuellfarbe)
```

red - cyan
green - magenta
blue - yellow

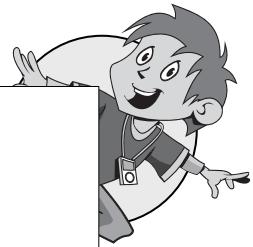
Damit können wir dreierpack() auch mit einer Schleife implementieren:

```
def dreierpack(seite, aenderung, farbpaare):
    """Zeichne Muster aus drei Dreiecken.
    Erstes Dreieck hat Seitenlänge seite.
    """
    for farbpaar in farbpaare:
        stiftfarbe, fuellfarbe = farbpaar
        fuelle_dreieck(seite, stiftfarbe, fuellfarbe)
        left(120)
        seite = seite + aenderung

    reset()
    pensize(10)
    right(90)
    farbenpaare = (("red","cyan"), ("green","magenta"),
                   ("blue","yellow"))
    dreierpack(65, 35, farbenpaare)
    hideturtle()
```

In diesem Code-Abschnitt wurde dreierpack() mit dem dritten Parameter farbpaare ausgestattet. Im »Hauptprogramm« wurde eine Anweisung zur Festlegung der farbenpaare eingefügt. farbenpaare wird im Aufruf von dreierpack() als drittes Argument (an den Parameter farbpaare) übergeben.

- ⇒ Führe diese Änderungen in dreierpack_arbeit.py aus. Teste das Programm – es sollte dieselbe Zeichnung wie vor der Änderung erzeugen.
- ⇒ Wenn alles geklappt hat, speichere eine Kopie des Programms unter dem Name dreierpack.py ab.



Ein bisschen eleganter und ein bisschen vielseitiger

Du kannst jetzt leicht aus der Funktion `dreierpack()` eine vielseitigere Funktion `multipack()` machen, die Muster zeichnen kann, die aus einer beliebigen Anzahl von Dreiecken bestehen. Die sollten dann aber nicht immer um 120° gegeneinander gedreht werden, sondern – wie weiter oben in `dreieck15.py` – um 360° / Anzahl der Dreiecke. Damit werden alle Dreiecke gleichmäßig auf einen vollen Winkel verteilt (wobei sie sich möglicherweise überlappen). Wie viele Dreiecke gibt es? So viele wie Farbenpaare!

➤ Führe diese Änderung selbstständig aus, wobei du die Funktion `dreierpack()` in `multipack()` umbenennst. Schwierig? Auch mit der Vorlage `dreieck15.py`? Macht nichts. Lies weiter, etwas weiter unten steht eine Lösung.

O.K., wenn dir das gelungen ist, ist die Funktion `multipack()` vielseitiger als `dreierpack()`. Und wie wird sie eleganter? Indem wir ausnützen, dass das Entpacken von Tupeln auch im Kopf einer `for`-Schleife direkt gemacht werden kann. Beispiel?

➤ Mach mit!

```
>>> for faktoren in ((3,5), (6,9), (5,11)):  
    a, b = faktoren  
    print("{0} * {1} = {2}".format(a, b, a * b))
```

```
3 * 5 = 15  
6 * 9 = 54  
5 * 11 = 55
```

So haben wir's bisher gemacht. Aber es geht auch so:

```
>>> for a, b in ((3,5), (6,9), (5,11)):  
    print("{0} * {1} = {2}".format(a, b, a * b))  
  
3 * 5 = 15  
6 * 9 = 54  
5 * 11 = 55
```



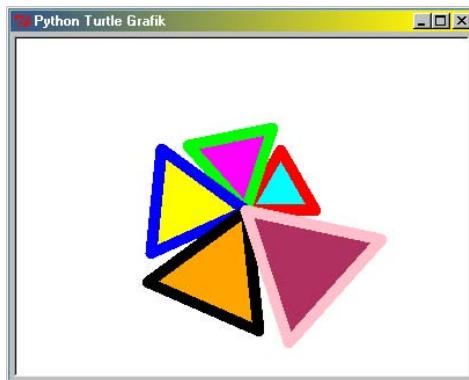
Etwas von dieser Art bauen wir nun in multipack() ein:

```
def multipack(seite, aenderung, farbpaare):
    """Zeichne Muster aus Dreiecken.
    Erstes Dreieck hat Seitenlänge seite.

    """
    anzahl = len(farbenpaare)
    drehwinkel = 360 / anzahl
    for stiftfarbe, fuellfarbe in farbpaare:
        fuelle_dreieck(seite, stiftfarbe, fuellfarbe)
        left(drehwinkel)
        seite = seite + aenderung
```

- Ersetze in dreierpack_arbeit.py die Funktion dreierpack() durch die eben beschriebene Funktion multipack().
- Ersetze den Aufruf von dreierpack() durch einen Aufruf von multipack().
- Füge zur Liste farbenpaare noch ein oder zwei Farbenpaare hinzu. Passe die ersten beiden Argumente des multipack()-Aufrufs so an, dass eine schöne Zeichnung entsteht.
- Wenn alles geklappt hat, speichere eine Kopie des Programms unter dem Name multipack.py ab.

Eine mögliche Ausgabe von multipack.py.



Mini-Quiz umarbeiten

Wir wollen jetzt die eben erarbeitete »Schleifen-Technik« dazu benutzen, das Mini-Quiz-Programm aus Kapitel 4 zu verbessern.

Mini-Quiz umarbeiten



Gleichzeitig ist hier eine gute Gelegenheit, dir eine Technik zu erklären, mit der man Programme umändern kann. Der entscheidende Punkt dabei ist: Man sollte während der Arbeit immer wieder ein Stadium erreichen, wo das Programm ohne Fehlermeldungen ausgeführt werden kann, obwohl alte Programmteile schon verstümmelt oder verändert sind und die neuen noch nicht fertig.

Arbeite die folgende Übung Punkt für Punkt nach und versuche, dir diese Technik anzueignen.

- Schließe alle IPI-Fenster und starte diesmal IDLE (PYTHON GUI)! Der Neustart stellt sicher, dass keine alten, unnötigen oder gar störenden Namen mit irgendwelchen Werten im Arbeitsspeicher sind. Zudem ist für die Entwicklung von Programmen mit Textausgabe IDLE(PYTHON GUI) die bessere Wahl. (Lies dazu Anhang F auf Seite 442.)
- Lade miniquiz05.py, aus Kapitel 4, ändere den Kopfkommentar und speichere es als miniquiz_arbeit.py im Ordner kap07 ab.

Zunächst stellen wir den Teil des Programmcodes »außer Dienst«, der geändert werden soll. Das tun wir, indem wir ihn auskommentieren. Das heißt, wir schreiben an den Anfang der Zeilen Kommentarzeichen. In miniquiz_arbeit.py soll der Teil geändert werden, der sich auf die Quizfragen bezieht. Das sind die neun Codezeilen nach der Anweisung `punkte = 0`.

Markiere diese neun Zeilen mit den `fragen/loesungen` und wähle `FORMAT|COMMENT OUT REGION` oder `Alt]+[3]`.

Dadurch werden diese Zeilen auskommentiert. Sollten sie später doch einmal wieder gebraucht werden, können aus einem markierten Bereich mit dem Menüpunkt `FORMAT|UNCOMMENT REGION` oder

`Alt]+[4]` die Kommentarmarken `##` wieder entfernt werden.

```
punkte = 0
##frage = "Welche Programmiersprache lernst du gerade? "
##loesung = "Python"
##quizfrage()
##
##frage = """Mit welchem reservierten Wort beginnt eine
##Funktionsdefinition? """
##loesung = "def"
##quizfrage()
##
##frage = "Wieviel reservierte Wörter hat Python? "
##loesung = "33"
##quizfrage()

print()
print("Du hast", punkte, "von 3 Punkten erreicht!")
```

Zu beachten ist, dass das Programm nun natürlich nicht mehr dasselbe tut wie vorher. Aber dennoch: Es bleibt ausführbar.

- Sichere `miniquiz_arbeit.py` und führe es aus!

7



Der Effekt ist:

>>>

Hello! Du kannst hier ein paar Quizfragen beantworten, um dein Wissen zu überprüfen.

Wie heißt du denn? *Buffi*

Also viel Glück, *Buffi* - es geht los!

Du hast 0 von 3 Punkten erreicht!

Du stehst noch ziemlich am Anfang, *Buffi*!

Sieh dir doch mal das Python-Video auf der CD an!

Das Programm ist ohne Fehler ausgeführt worden. Es werden keine Quizfragen gestellt und der Kandidat hat daher nur 0 Punkte erreicht. Das war ja zu erwarten.

➤ Nun erstellen wir die »Datenstruktur« für die Quizangaben. Dazu musst du die entsprechenden Strings aus dem Programm wieder in ein Tupel von Tupeln zusammensammeln und zweckmäßigerweise an den Anfang des Programms stellen.

```
# (frage, loesung) - Tupel für das Quiz
# kann erweitert und/oder geändert werden
quizdaten=(( "Welche Programmiersprache lernst du " +
              "gerade? ", "Python"),
            ("Mit welchem reservierten Wort beginnen " +
              "Funktionsdefinitionen? ", "def"),
            ("Wie viele reservierte Wörter hat Python? ", "33")
            )
```

- `quizdaten` an den Anfang zu stellen ist deswegen günstig, weil sie dort am leichtesten zu finden, zu ändern oder zu ergänzen sind. Wenn du noch ein anderes Quiz programmieren willst, sind es gerade diese Daten, die abzuändern sind.
- Nun ist das Programm wieder in einem korrekten Zustand. Speichere es und führe es aus!

Die Ausgabe ist die gleiche, jedoch:

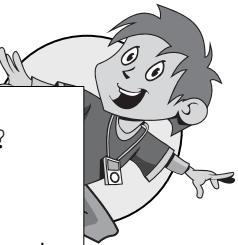
➤ Mach mit!

>>> quizdaten

```
('Welche Programmiersprache lernst du gerade?', 'Python'),
('Mit welchem reservierten Wort beginnen Funktionsdefinitio-
```

Mini-Quiz umarbeiten

```
nen? ', 'def'), ('Wie viele reservierte Wörter hat Python?  
', '33'))  
>>>
```



Offenbar verweist der Name `quizdaten` nun wie gewünscht auf das Tupel von Frage-Antwort-Paaren. Wir versuchen, darauf in einer Schleife zugreifen:

```
>>> for eintrag in quizdaten:  
    print(eintrag)
```

```
('Welche Programmiersprache lernst du gerade? ', 'Python')  
(Mit welchem reservierten Wort beginnen Funktionsdefinitio-  
nen? ', 'def')  
(Wie viele reservierte Wörter hat Python? ', '33')
```

Nun werden die Zweier-Tupel einzeln ausgegeben. Das ist noch nicht das, was wir wollen, doch sind sie das Rohmaterial für die Ausführung einer `quizfrage()`. Wir versehen daher die Funktion `quizfrage()` in `miniquiz_arbeit.py` mit dem Parameter `quizeintrag`:

» Füge im Funktionskopf von `quizfrage()` einen Parameter `quizeintrag` ein.

```
def quizfrage(quizeintrag):  
    global punkte  
    ...
```

» Füge nach `punkte = 0` eine `for`-Schleife ein, die die Quizfragen stellen soll:

```
for eintrag in quizdaten:  
    quizfrage(eintrag)
```

» Speichere das Programm und führe es aus!

Also viel Glück, Buffi, es geht los!

```
Traceback (most recent call last):  
  File "C:\py4kids\kap07\miniquiz_arbeit.py", line 36, in  
<module>  
    quizfrage(eintrag)  
  File "C:\py4kids\kap07\miniquiz_arbeit.py", line 18, in  
quizfrage  
    antwort = input(frage)  
NameError: global name 'frage' is not defined
```

7



Das musste ja einmal kommen, es kann nicht immer fehlerfrei abgehen! Jetzt haben wir übersehen, dass in `quizfrage()` der Name `frage` verwendet wird, aber nicht mehr definiert ist – die entsprechenden Strings stecken ja bereits in den `quizdaten`.

An `quizfrage()` übergeben wir die Texte der Frage und der Lösung jeweils als Zweier-Tupel. Nun packen wir das Tupel einfach aus:

```
def quizfrage(quizeintrag):
    global punkte
    frage, loesung = quizeintrag
    antwort = input(frage)
    ...
```

Damit gibt es nun in `quizfrage()` die beiden benötigten Namen, diesmal als lokale Namen. Und unser Programm läuft.

- Jetzt werden die neun auskommentierten Zeilen sicher nicht mehr gebraucht. Entferne sie daher aus dem Programm.
- Erfinde eine weitere Quizfrage samt Lösung und füge sie in das Tupel `quizdaten` ein!
- Sichere das Programm und teste es!

Bei mir ergab der Testlauf (nur die letzten Zeilen der Ausgabe sind angegeben, damit du siehst, welche Frage ich dazuerfunden habe):

Mit welchem reservierten Wort beginnen Zählschleifen? *for*
Richtig!

Du hast 4 von 3 Punkten erreicht!

Fein, du hast schon einiges gelernt, Buffi!

Und sieh dir doch mal das Python-Video auf der CD an!

Die `for`-Schleife hat brav die vier Fragen abgearbeitet. Aber: 4 von 3 Punkten erreicht?

Vielleicht hätten wir der Programmänderung doch eine genauere Planung voranschicken sollen. Da stimmt noch immer etwas nicht. Wir haben ja jetzt vier Fragen. Und selbst die könnten noch erweitert werden. Das werden wir auch noch hinkriegen. Die Anzahl der Elemente eines Tupels erhalten wir mit `len()`:

```
>>> len(quizdaten)
```

4

Zusammenfassung



Also weisen wir diese Zahl einem Namen zu und verwenden diesen am Ende bei der Ausgabe. Aber auch die `if...elif...else`-Anweisung, die die Beurteilung ermittelt, bedarf noch einer Änderung. Ich schlage vor: »Super!« wird vergeben, wenn die Anzahl der `punkte` größer als 80 % der Anzahl der Fragen ist und die schlechteste Beurteilung, wenn `punkte` kleiner als 25 % dieser Zahl ist.

⇒ Führe die entsprechenden Änderungen im Code unter Beachtung folgender Hinweise durch:

Hinweis 1: Nach der Festlegung der Quizdaten muss eine Zuweisung der Form `fragen_zahl = len(quizdaten)` stehen.

Hinweis 2: 80 % von `x` kann mit dem Ausdruck `x*0.8` berechnet werden.

Hinweis 3: Statt der Bedingung `punkte == 3` schreiben wir nun: `punkte > frage_zahl * 0.8`

Hinweis 4: Beseitige auch das hässliche Leerzeichen zwischen Namen und Rufzeichen in der vorletzten Zeile der Ausgabe.

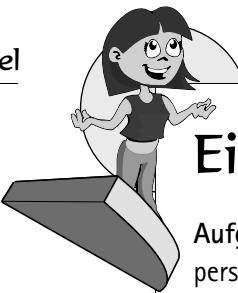
Damit kann unser Quizprogramm leicht mit beliebig vielen Fragen erweitert oder abgeändert werden.

⇒ Sobald das Programm fehlerfrei läuft, speichere eine Kopie davon unter dem Namen `miniquiz06.py` ab.

Zusammenfassung

- ❖ Mit `for element in Wertevorrat:` werden Blöcke von Anweisungen wiederholt. `for` und `in` sind reservierte Wörter.
- ❖ In der `for`-Schleife können sowohl statische Typen als Wertevorrat verwendet werden, z.B. der zusammengesetzte Datentyp `tuple`, als auch dynamische wie beispielsweise `range()` Objekte.
- ❖ Die in Python eingebaute Funktion `len()` ermittelt die Anzahl der Elemente eines Wertevorrats.
- ❖ Die IDLE hat `INDENT/DEDENT-` und `COMMENT-OUT/UNCOMMENT-` Kommandos, um Quellcode zu bearbeiten.
- ❖ Mit der Technik des Tupel-Entpackens kann man die Elemente von Tupeln einzelnen Variablen zuweisen.

7



Einige Aufgaben ...

Aufgabe 1: Schreibe das Ergebnis von Aufgabe 2 aus Kapitel 4, dein persönliches Quiz, so um, dass es von der Schleifentechnik, die wir in diesem Kapitel in miniquiz eingebaut haben, Gebrauch macht.

Aufgabe 2: Ändere miniquiz06.py so zu miniquiz07.py ab, dass das Modul aus den Definitionen zweier Funktionen: quizfrage() – wie gehabt – und quiz() besteht sowie der Zuweisung des Fragenmaterials an den Namen quizdaten. Im anschließenden »Hauptprogramm« soll bloß quiz() mit quizdaten als Argument aufgerufen werden.

Damit soll das Modul folgenden Aufbau haben:

```
# Kopfkommentar

quizdaten = (...)

def quizfrage(quizeintrag):
    ...

def quiz(daten):
    ...

quiz(quizdaten)
```

Die ganze Quizabwicklung soll also in die Funktion quiz() verpackt werden. Beachte besonders, dass punkte eine globale Variable sein muss. Sie muss in allen Funktionen, in denen Zuweisungen an punkte geschehen, global deklariert werden.

... und einige Fragen

1. range() kann auch mit zwei Argumenten aufgerufen werden. Welche Werte erzeugt range(3,8) und welche range(-3, 4)?
2. range() kann auch mit drei Argumenten aufgerufen werden. Welche Werte erzeugt range(4,12, 2)? Welche Rolle spielt das dritte Argument?
3. Welcher Aufruf von range() mit drei Argumenten liefert der Reihe nach die Werte 3, 2, 1, 0, -1, -2, -3?
4. Welche Werte haben a, b und c nach der Anweisung:
`>>> (a,(b,c)) = ((1,2),(3,4))`

8



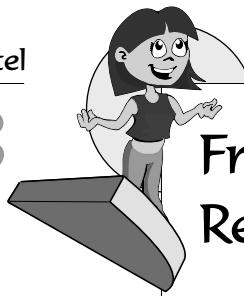
Mehr Schleifen: Friedenslogo, Superrosette

Nachdem du erste Bekanntschaft mit der `for`-Schleife gemacht hast, zeige ich dir an einigen Beispielen, wie du solche Schleifen produktiv einsetzt. Bei dieser Gelegenheit wirst du erfahren, wie flexibel und kreativ man bei der Grafikprogrammierung mit Farben umgehen kann: Viele haben Namen, noch mehr aber lassen sich durch Zahlen festlegen!

Bis hierher hast du wohl auch schon bemerkt, dass die Ausführung von Turtle-Grafikprogrammen oft recht langsam erfolgt. Das ist beabsichtigt, denn solange das Programm noch nicht fertig ist, kannst du damit gut seinen Ablauf beobachten und überprüfen, ob du deine Ideen richtig in den Programmcode umgesetzt hast. Für umfangreichere Programme, besonders wenn sie schon fertig programmiert sind, ist es aber manchmal etwas mühsam. Neben der Funktion `speed()`, die du schon von früher kennst, hat `turtle` auch die Funktion `tracer()`, mit der Grafikprogramme sehr beschleunigt werden können. Die wirst du in diesem Kapitel auch kennen lernen. In diesem Kapitel lernst du ...

- ◎ ... Neues darüber, wie man mit Python plus `turtle` Farben einsetzt,
- ◎ ... wie man die Turtle mit `setheading()` in eine bestimmte Richtung orientieren kann,
- ◎ ... wie man die Turtle mit `tracer()` dazu bringt, ihre Zeichnungen schneller zu erstellen.
- ◎ ... wie man ein Programm in Funktionsdefinitionen und Hauptprogramm gliedert.

8



Friedenszeichen auf der Regenbogenfahne

Bei einer Suche nach peace logo mit Google bin ich einmal auf nebenstehende Grafik gestoßen. Damit du sie in Farben sehen kannst, sieh dir auf der Buch-CD das Bild `peace-rainbow-logo.gif` im Ordner `py4kids\kap08` an. Vielleicht erinnerst du dich auch an Aufgabe 3 aus Kapitel 6, wo eine andere Variante dieses Zeichens vorgekommen ist.



Peace-Logo.

Wir wollen sehen, ob wir unsere neu gewonnenen Kenntnisse über die for-Schleife für die Programmierung dieser Grafik nutzen können. Da du noch nicht viel Erfahrung mit Schleifen hast, wollen wir den Weg der Bottom-up-Entwicklung gehen.

- Starte den IPI-TurtleGrafik neu und öffne ein neues Editor-Fenster. Schreibe einen Kopfkommentar hinein und sichere die Datei mit dem Namen `peace_arbeit.py`.

Zunächst brauchen wir für die sieben Streifen der Fahne sieben Farben. Eine Liste der Farben, die in `turtle` zur Verfügung stehen, findest du in Anhang D. Wir wollen jetzt im Direktmodus die passenden Farben aussuchen. Dazu nutzen wir die Tatsache, dass die Funktion `color()` auch mit nur einem Argument aufgerufen werden kann. Dann setzt sie sowohl die Stiftfarbe wie auch die Füllfarbe auf den angegebenen Farbwert.

- Mach mit!

```
>>> pensize(20)  
>>> color("red")
```

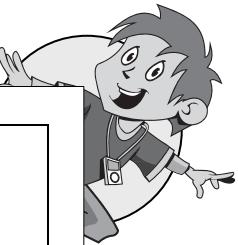
Das zeigt uns, wie das "red" aussieht. Wir wollen aber wissen, wie andere Farben aussehen. Zum Beispiel (Anhang D):

```
>>> color("chocolate")  
>>> color("hotpink")  
>>> color("hotpink3")  
>>> color("royalblue1")  
>>> color("orchid4")
```

Nachdem ich einige Zeit so herumgespielt hatte, bin ich auf folgendes Tupel von sieben Farben für die »Regenbogenfahne« gekommen.

Die Regenbogenfahne

```
friedensfarben = ("red3", "orange", "yellow",
                  "seagreen4", "orchid4",
                  "royalblue1", "dodgerblue4")
```



Wenn du möchtest, kannst du sie übernehmen. Vielleicht möchtest du dir lieber eine eigene Farbkombination zusammenstellen.

➤ Schreibe diese Anweisung an den Anfang des Programms `peace_arbeit.py`. (An den Anfang heißt immer: unter die import-Anweisung[en].) Natürlich braucht unser Programm die Funktionen aus `turtle`, aber auch `jump()` aus `mytools`, wie du gleich sehen wirst.

Die Regenbogenfahne

Als nächstes Teilproblem behandeln wir das Zeichnen *eines* Fahnenstreifens. Der soll eine gewisse Länge, eine gewisse Breite und eine bestimmte Farbe haben. Daher:

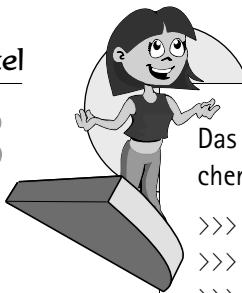
```
def streifen(laenge, breite, farbe):
    pensize(breite)
    pencolor(farbe)
    forward(laenge)
```

Füge diese Funktionsdefinition in dein Programm ein. Das ist eigentlich eine recht vielseitige Funktion. Führe das Programm `peace_arbeit.py` aus, damit die Funktionsdefinition ausgeführt wird. Dann fahre mit dem IPI fort:

```
>>> reset()
>>> streifen(100, 20, "red")
>>> reset()
>>> right(90)
>>> streifen(150, 40, "orange")
```

Jetzt müssen wir immer wieder die Turtle an einen neuen Startpunkt führen. Da können wir `jump()` gut gebrauchen:

```
>>> from mytools import jump
>>> reset(); jump(-100)
>>> right(90); jump(-200)
>>> streifen(400,40,"blue")
```



Das macht schöne Streifen. Aber zurück müssen wir auch wieder und brauchen dabei nicht zu zeichnen:

```
>>> jump(-400)
>>> jump(40, -90)
>>> streifen(400,40,"green")
```

Ja, so dürfte es gehen. Wie breit sollen aber unsere Streifen sein? Das Fenster ist 300 Pixel hoch. Also:

```
>>> 300.0 / 7
42.857142857142854
```

Machen wir die Streifen sicherheitshalber etwas breiter, sagen wir 44. Den Rückwärtssprung schließen wir noch in die Funktion `streifen()` ein. Und mit dem ersten Streifen beginnen wir drei Streifenbreiten unter der Mitte.

Jetzt können wir die `for`-Schleife ins Spiel bringen! Denn es soll *für* (also `for`) jede Farbe ein Streifen gezeichnet – und dann eine Streifenbreite nach links gesprungen werden.

Das ergibt folgende Ergänzungen für unseren Code:

```
streifenbreite = 44

def streifen(laenge, breite, farbe):
    pensize(breite)
    pencolor(farbe)
    forward(laenge)
    jump(-laenge)

    reset()
    jump(-3*streifenbreite)
    right(90)
    jump(-200)

for farbe in friedensfarben:
    streifen(400, streifenbreite, farbe)
    jump(streifenbreite, -90)
```



Die Turtle zeichnet die Fahnenstreifen.

Funktioniert gut. Nur sollte die Turtle danach wieder in die Fenstermitte wandern, denn dort ist der Mittelpunkt des Friedenslogos. Dafür gibt es im Modul `turtle` die Funktion `home()`. Mit dieser Ergänzung wollen wir nun aus diesen Anweisungen auch eine Funktion machen: `fahne()`.



```
def fahne():
    jump(-3*streifenbreite)
    right(90)
    jump(-200)

    for farbe in friedensfarben:
        streifen(400, streifenbreite, farbe)
        jump(streifenbreite, -90)
    penup()
    home()
    pendown()

reset()
fahne()
```

- ⇒ Füge die Ergänzungen in den Code von `peace_arbeit.py` ein und teste das Programm.

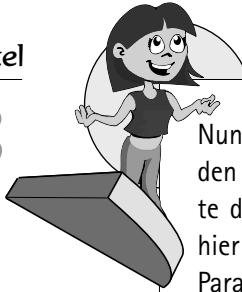
Das Friedenslogo

Nun fehlt uns noch das Friedenslogo. Das besteht aus vier Linien vom Mittelpunkt zum Rand des Kreises und aus eben diesem Rand. Eine Funktion, die einen Kreis mit gegebenem `radius` zeichnet, dessen Mittelpunkt die aktuelle Turtle-Position ist, hatten wir schon in Kapitel 6 für das Yin-Yang-Symbol entworfen (aber nicht codiert). Der Code sieht so aus:

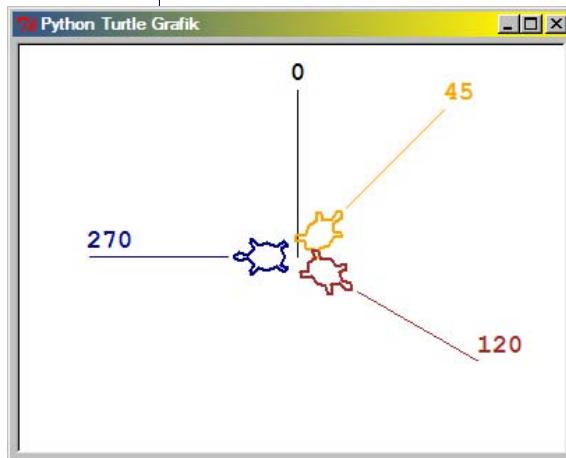
```
def kreis(radius):
    jump(radius, 90)
    circle(radius)
    jump(radius, -90)
```

- ⇒ Füge diese Funktionsdefinition in `peace_arbeit.py` ein, führe das Programm aus und überzeuge dich im interaktiven Modus, dass `kreis()` richtig arbeitet. Zum Beispiel:

```
>>> reset()
>>> pensize(5)
>>> kreis(50)
>>> pensize(10)
>>> kreis(80)
```



Nun müssen noch die vier Radien gezeichnet werden. Das kannst du mit den Turtle-Grafikmitteln, die du bereits kennst, locker erreichen. Ich möchte dich aber mit einer neuen Turtle-Grafik-Funktion bekannt machen, die hier vorteilhaft angewendet werden kann: `setheading()`. Sie hat einen Parameter, und zwar einen Winkel. `setheading(winkel)` stellt die Turtle so ein, dass sie in Richtung `winkel` schaut. Dabei entspricht `winkel = 0` der Richtung nach oben (Norden) und die Winkel werden im Uhrzeigersinn gemessen. Beispiele:



```
>>> reset()
>>> pensize(8)
>>> setheading(45)
>>> setheading(120)
>>> setheading(270)
>>> setheading(-45)
```

So funktioniert `setheading()`.

Im Friedens-Logo haben wir vier Radien zu zeichnen, und zwar mit folgenden Orientierungen: 135° , 180° , 225° und 0° . Noch dazu ist jeder Radius ein `streifen()`.

Führe das Programm `peace_arbeit.py` nochmals aus. Es entsteht das Bild mit der Regenbogenfahne. Fahre nun interaktiv fort:

```
>>> pencolor("white")
>>> pensize(20)
>>> kreis(120)
>>> for winkel in (135, 180, 225, 0):
    setheading(winkel)
    streifen(120, 20, "white")

>>> hideturtle()
```

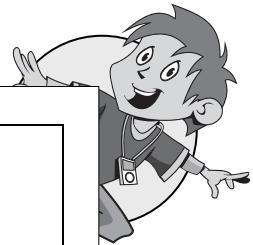
Da hat sich nun eine `for`-Schleife über die Folge der benötigten Winkel sehr bewährt. Natürlich soll das Ergebnis dieser interaktiven Experimente noch in eine Funktion `logo()` im Programm `peace_arbeit.py` gegossen werden:

»tracer()«

```
def logo(radius):
    for winkel in (135, 180, 225, 0):
        setheading(winkel)
        streifen(radius, 20, "white")
    # hier ist pensize 20
    kreis(radius)

def peace():
    fahne()
    logo(120)
    hideturtle()

reset()
peace()
```



- Füge diese Codeteile ein und teste das Programm. Wenn es zufriedenstellend funktioniert, speichere eine Kopie davon unter `peace01.py` ab.

»tracer()«

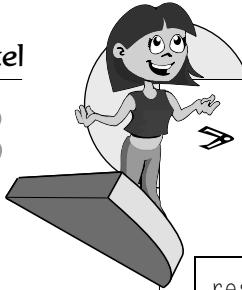
Turtle-Grafik ist unter anderem deshalb so gut zum Programmierenlernen geeignet, weil man der Ausführung der Programme direkt zuschauen kann.

Jetzt, wo wir mit unseren Schleifen immer reichhaltigere Grafiken erstellen werden, ist das aber manchmal mit dem Nachteil verbunden, dass die Programmausführung recht langsam ist. Solange man noch versucht, die Vorgänge zu verstehen, ist das ein Vorteil. Es kann aber mit der Zeit zu nerven beginnen.

Deshalb bietet das Modul `turtle` die Möglichkeit, das langsame Marschieren der Turtle abzuschalten: mit der Funktion `tracer()`. (Das englische Wort `trace` bedeutet (unter anderem) Ablaufverfolgung. Ein `tracer` ist demnach etwa ein »Ablaufverfolger«.)

Diesen »Ablaufverfolger« kann man ausschalten mit dem Funktionsaufruf `tracer(False)`. Die »Ablaufverfolgung« wird wieder eingeschaltet mit `tracer(True)`.

Im Direktmodus ist das Ausschalten des Tracers normalerweise nicht empfehlenswert. Doch Programmabläufe können damit sehr beschleunigt werden. Wir probieren das gleich mit unserem `peace_arbeit.py` aus.



➤ Führe folgende Änderungen aus: Füge vor dem Aufruf von `peace()` die Anweisung `tracer(False)` und danach die Anweisung `tracer(True)` ein!

```
reset()
tracer(False)
peace()
tracer(True)
```

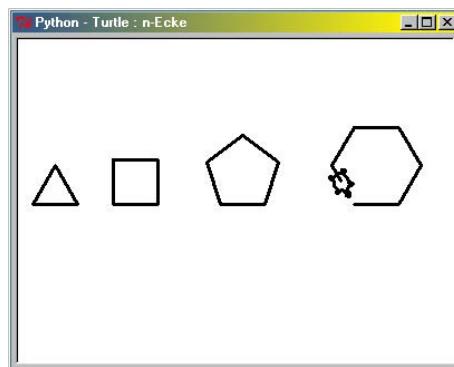
➤ Speichere das Programm und führe es aus!

Die Grafik ist augenblicklich da. Oft wirst du Grafiken entwickeln wollen, die rasch gezeichnet werden sollen. Während der Entwicklung ist die langsame Ausführung durch die Turtle sehr hilfreich bei der Kontrolle, ob das Programm richtig läuft, und bei der Fehlersuche. Sobald das Programm aber fehlerfrei läuft, ist die Anwendung von `tracer()` sehr empfehlenswert. Wesentlich dabei ist, dass am Ende, wenn die Grafik vollständig gezeichnet ist, ein `tracer(True)`-Aufruf gemacht wird. Speichere eine Kopie des fertigen Programms unter `peace02.py` ab.

n-Ecke

Schleifen bieten den Vorteil, dass mit ihnen die Anzahl der Schleifendurchläufe variabel gestaltet werden kann. Wir haben jetzt die Möglichkeit, eine Funktion zu definieren, die folgende vier Figuren und noch viele mehr zeichnen kann:

Die Funktion `n_eck()` zeichnet verschiedene Polygone.



Alle diese Figuren sind regelmäßige n-Ecke (Vielecke, Polygone). In dieser Bezeichnung steht n für eine positive ganze Zahl größer als 2. Die Zahl n gibt die Anzahl der Ecken an. Das heißt, für $n = 3, 4, 5, 6$ usw. sind damit regelmäßige Dreiecke, Vierecke, Fünfecke, Sechsecke usw. gemeint. Regelmäßig heißen Vielecke, deren Seiten und Winkel alle gleich groß sind.



Klar ist, auch aus unseren Erfahrungen mit Dreiecken, Vierecken usw. aus früheren Kapiteln, dass sie durch eine Folge von `forward()`- und `left()`-Anweisungen erzeugt werden können. Die Anzahl der Schleifendurchläufe muss gleich der Anzahl der Ecken sein:

```
def n_eck(eckenzahl, seitenlaenge):
    drehwinkel = ????
    for i in range(eckenzahl):
        forward(seitenlaenge)
        left(drehwinkel)
```

Klar ist weiterhin, dass der Drehwinkel von der Anzahl der Ecken abhängt. Wir müssen also herausfinden, wie wir diesen Drehwinkel aus n berechnen können.

- ⇒ Wenn du eine Idee hast, welcher Ausdruck im obigen Listing für Drehwinkel eingesetzt werden soll, probiere es aus! Experimentiere ruhig ein bisschen herum.
- ⇒ Du kannst es aber auch stattdessen durch systematisches Überlegen herausbekommen:

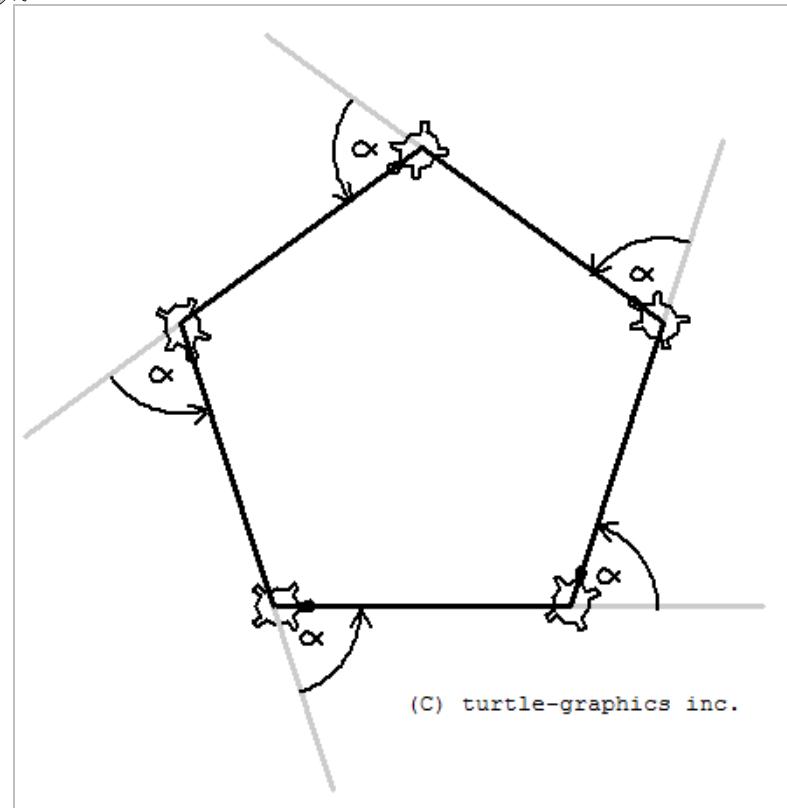
Wir hatten bisher:

Figur	Anzahl der Ecken	Drehwinkel der Turtle
Dreieck	3	120°
Quadrat	4	90°
Fünfeck	5	???
Sechseck	6	60°
n-Eck	n	???

- ⇒ Versuche herauszufinden, welcher Wert für den Drehwinkel in dieser Tabelle für das Fünfeck einzutragen ist!
- ⇒ Überprüfe das Ergebnis deiner Überlegungen, indem du eine Funktion schreibst, die ein Fünfeck zeichnet. Entsteht ein richtiges Fünfeck?

Wenn du den Weg der Turtle verfolgst, wirst du bemerken, dass sie am Ende, nachdem sie sich bei jeder Ecke um den gleichen Winkel gedreht hat, eine Drehung um volle 360° ausgeführt hat. Wenn du das nicht glaubst, dann gehe selbst einen fünfeckigen Weg nach – zum Beispiel auf einem Sportplatz oder auch in einem größeren Raum. Beobachte dabei, um wie viel du dich insgesamt gedreht hast.





Bestimmung des Drehwinkels der Turtle zur Zeichnung eines Fünfecks.

Damit ergibt sich $drehwinkel = 360/5$. (Die Division ergibt 72° .) Für ein beliebiges anderes n -Eck muss in dieser Formel 5 durch die Eckenzahl ersetzt werden. Also:

Fünfeck	5	72°
n -Eck	n	$360/n$

Damit ergibt sich die einzige noch unklare Anweisung der Funktion `n_eck`:

$$drehwinkel = 360 / \text{eckenzahl}$$

Wir wollen nun ein Modul mit Funktionen erstellen, die Figuren aus n -Ecken zeichnen.

- Öffne in der IDLE ein neues Editor-Fenster!
- Schreibe einen Kopfkommentar und speichere unter dem Namen `n_eck_arbeit.py` ab!
- Schreibe die Funktion `n_eck(eckenzahl, seitenlaenge)`!
- Sichere das Programm, führe es aus und prüfe mit dem IPI durch einige Aufrufe der Funktion, ob sie korrekt arbeitet!



- Schreibe eine Funktion `n_eck_demo()`, die das Bild zeichnet, das am Anfang dieses Abschnitts zu sehen ist! (Hinweis: Importiere `jump()` aus `mytools!`). Füge als letzte Anweisung des Programms einen Aufruf von `n_eck_demo()` an.
- Sichere das Programm, teste es und korrigiere es, falls Fehler auftreten sind!

Wir machen daraus ein importierbares Modul

Nichts ist einfacher als das: Wir speichern das Programm unter einem passenden Namen, ich schlage vor `polygon.py` (`Polygon` ist der geometrische Fachausdruck für Vieleck), im Verzeichnis `C:\py4kids\mylib` ab.

Da dieses Verzeichnis bereits im Suchpfad von Python liegt, können wir jetzt `n_eck()` wie gewohnt importieren:

```
>>> from polygon import n_eck
```

- Schließe alle IDLE-Fenster, starte den IPI neu und führe direkt diese import-Anweisung aus.

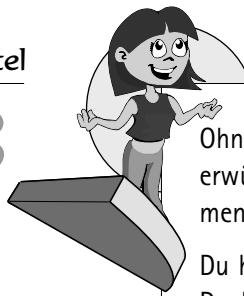
Der `import` funktioniert zwar, hat aber eine unerwünschte Nebenwirkung: die Funktion `n_eck_demo()` wird ausgeführt. Das liegt daran, dass sie in der letzten Zeile von `polygon.py` aufgerufen wird.

The screenshot shows two windows. The top window is titled "76 *IPI Shell / Turtle Grafik*" and contains the Python shell history:

```
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MS
C v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more
information.
===== No Subprocess =====
>>> from polygon import n_eck
```

The bottom window is titled "Python - Turtle : n-Ecke" and displays four geometric shapes: a triangle, a square, a pentagon, and a hexagon, each with a small gear icon next to it.

Die `import`-Anweisung bewirkt die Ausführung von `n_eck_demo()`. Nicht gut!



Ohne Zweifel ist das ein Verhalten, das beim Import von Modulen nicht erwünscht ist. Wer will schon jedes Mal vier Vielecke vorgeführt bekommen, wenn er die Funktion `n_eck()` benötigt?

Du hast hier eine Situation, in die du beim Programmieren öfters kommst: Du hast eine Datei voll Python-Code geschrieben, die eine oder mehrere Funktionsdefinitionen enthält. Am Ende der Datei stehen Anweisungen, die die darüber stehenden Funktionen benutzen und sofort ausgeführt werden sollen. Oft sagt man, diese Anweisungen stellen das »Hauptprogramm« dar.

Diese Datei soll nun auf zwei verschiedene Weisen verwendbar sein:

- ❖ Als selbstständig ausführbares Programm (stand alone program). Beispielsweise, indem man es in ein Editor-Fenster lädt und dann mit **RUN|RUN MODULE** oder **F5** ausführt.
- ❖ Als Modul einer Programmbibliothek, um die vielen schönen Funktionen in anderen Programmen benutzen zu können. Das tut man, indem man mit einer `import`-Anweisung alle oder auch nur einzelne Funktionen des Moduls importiert. In diesem Fall dürfen nur die Funktionsdefinitionen ausgeführt werden, alle Anweisungen des Hauptprogramms aber nicht.

In Python kann man das einfach erreichen: Man schreibt die Anweisung(en) des »Hauptprogramms« – eingerückt! – in eine besondere `if`-Anweisung:

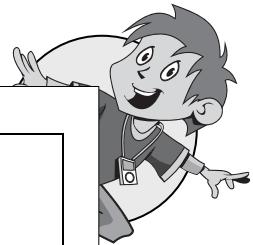
```
if __name__ == "__main__":
    n_eck_demo()
```

Diese Möglichkeit beruht auf folgendem Sachverhalt: Python verwendet einen speziellen eingebauten Namen: `__name__`. Er beginnt und endet mit zwei Unterstrichen. Dieser verweist auf einen String, abhängig davon, wie ein Modul genutzt wird. Wird er importiert, dann wird `__name__` der Name des Moduls ohne die Erweiterung `.py` zugewiesen. Wird ein Modul als Programm ausgeführt, dann verweist `__name__` auf den String `"__main__"`.

- Prüfe das nach, indem du `polygon.py` in einem Editor-Fenster öffnest und vorübergehend die Anweisung

```
print("Name:", __name__)
```

in das Modul `polygon.py` einfügst! Dein Listing könnte dann etwa so (oder ähnlich, je nachdem wie du `n_eck_demo()` programmiert hast) aussehen:



```
def n_eck_demo():
    reset()
    pensize(3)
    right(90)
    jump(-185)
    for ecken in (3, 4, 5, 6):
        n_eck(ecken, 40)
        jump(ecken*24)

print("Name:", __name__)
n_eck_demo()
```

➤ Führe nun das Programm mit **F5** aus. Die print-Anweisung gibt aus:

```
>>>
Name: __main__
>>>
```

➤ Um zu sehen, was beim import geschieht, starte den IPI neu und importiere `n_eck()` aus dem Modul `polygon`. Jetzt zeigt die print-Anweisung, dass `__name__` einen anderen Wert hat, nämlich den Modul-Namen.

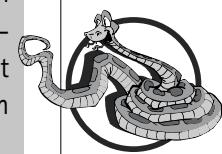
```
>>> from polygon import n_eck
Name: polygon
>>>
```

➤ Entferne die print-Anweisung in `polygon.py` wieder und setze den `n_eck_demo()`-Aufruf in die beschriebene if-Anweisung:

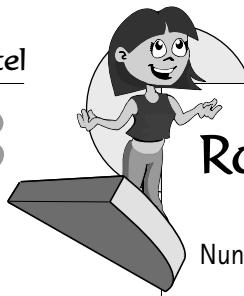
```
if __name__ == "__main__":
    n_eck_demo()
```

Nun wird `n_eck_demo()` nur noch ausgeführt, wenn das Skript ausgeführt wird, aber nicht mehr, wenn daraus etwas importiert wird, denn dann ist ja die Bedingung in der if-Anweisung nicht erfüllt. (Falls dein Hauptprogramm aus mehreren oder auch vielen Anweisungen besteht, können sie alle im Block dieser if-Anweisung untergebracht werden.)

Beachte, dass `__name__` und `__main__` und viele »spezielle Namen« in Python mit zwei Unterstrichen beginnen und mit zwei Unterstrichen enden. Erfinde niemals selbst Namen mit dieser Besonderheit. Es besteht sonst die Gefahr, dass deine Namen mit wichtigen speziellen Namen von Python in Konflikt geraten – mit unvorhersehbaren Folgen!

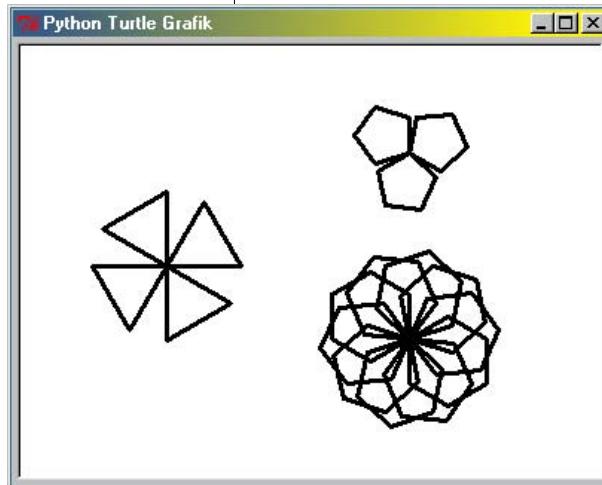


8



Rosetten

Nun geht's darum, Muster aus n-Ecken zu erzeugen. Zum Beispiel solche:



Ich nenne diese Muster Rosetten. Rosetten bestehen aus Polygonen (also n-Ecken) mit einer gemeinsamen Ecke, die miteinander gleiche Winkel einschließen.

Im obigen Bild haben wir drei Rosetten: eine aus vier Dreiecken, eine aus drei Fünfecken und eine aus zehn Sechsecken.

Eine Funktion zum Zeichnen von Rosetten hat einen einfachen Aufbau:

Funktion rosette():

Parameterliste: eckenzahl, blattzahl, seite

drehwinkel $\Rightarrow 360 / \text{blattzahl}$

Wiederhole blattzahl Mal:

n_eck mit seite und eckenzahl zeichnen

Turtle um drehwinkel nach links drehen

➤ Codiere in deinem Programm `n_eck_arbeit.py` die Funktion `rosette()` und teste sie, indem du mit folgendem Code die obige Grafik erzeugst!

```
reset()
pensize(3)
jump(100, -90)
rosette(3, 4, 50)
jump(180, 65)
rosette(5, 3, 24)
jump(-125)
rosette(8, 10, 30)
hideturtle()
```

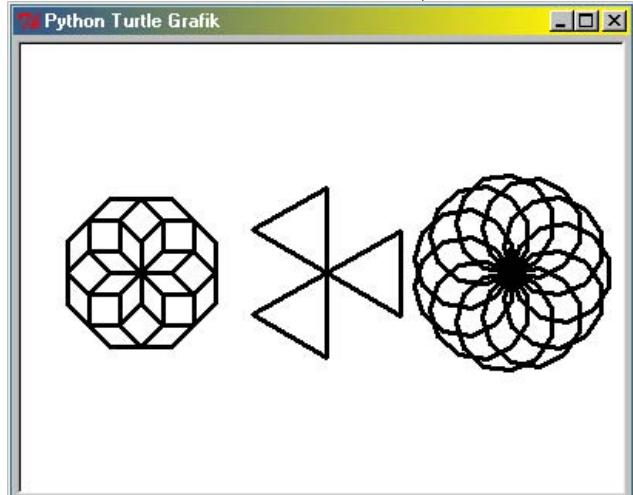
Rosetten



» Besonders schöne Rosetten ergeben sich, wenn die eckenzahl und die blattzahl gleich sind. Codiere in n_eck_arbeit.py eine Funktion super_rosette(eckenzahl, seite), die einfach Rosetten zeichnet, bei denen eckenzahl und blattzahl gleich groß sind.

» Mach mit!

```
>>> reset(); pensize(3)
>>> super_rosette(3, 55)
>>> jump(120, -90)
>>> super_rosette(8, 20)
>>> jump(240, 90)
>>> super_rosette(11, 18)
```



» Wenn du fertig bist, füge die Funktionen rosette() und super_rosette() in die Datei polygon.py in c:\py4kids\mylib ein. Ab nun können sie auch aus dem Modul polygon importiert werden.

Da fällt mir ein Bild ein, das im Wikipedia-Artikel über Turtle-Grafik in der Programmiersprache Logo zu finden ist:

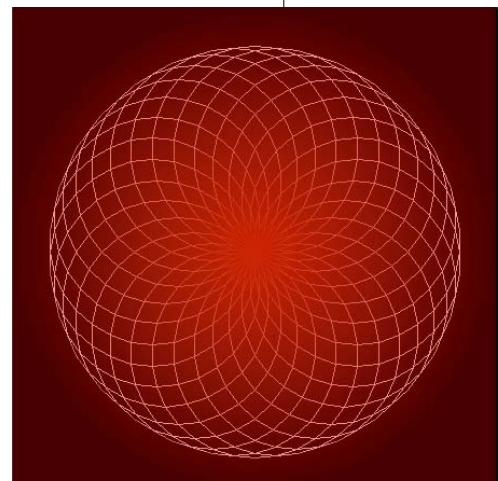
[http://de.wikipedia.org/wiki/Logo_\(Programmiersprache\)](http://de.wikipedia.org/wiki/Logo_(Programmiersprache))

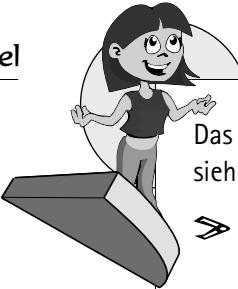
Turtle-Grafik-Beispiel aus wikipedia.de.

Etwas Ähnliches wollen wir nun nachprogrammieren.

» Mach mit!

```
>>> reset()
>>> setup(500,500)
>>> super_rosette(36, 20)
```





Das ergibt ein Bild, das dem im Wikipedia-Artikel schon etwas ähnlich sieht.

- Schreibe die obigen drei Anweisungen in der Datei `neck_arbeit.py` in den Block der `if __name__ == '__main__':`-Anweisung. Speichere, teste.

Der Code von `n_eck_arbeit.py` sollte jetzt etwa so aussehen:

```
from turtle import *
from mytools import jump

def n_eck(eckenzahl, seitenlaenge):
    drehwinkel = 360 / eckenzahl
    for i in range(eckenzahl):
        forward(seitenlaenge)
        left(drehwinkel)

def rosette(ecken, blaetter, seite):
    for i in range(blaetter):
        n_eck(ecken, seite)
        left(360 / blaetter)

def super_rosette(ecken, seite):
    rosette(ecken, ecken, seite)

if __name__ == '__main__':
    reset()
    setup(500, 500)
    super_rosette(36, 20)
```

Die erzeugte Grafik ist etwas farblos und das Zeichnen der Grafik dauert fast endlos. Wir werden das Programm nun mit ein paar kleinen alten und neuen Tricks aufpeppen:

- Stelle vor dem Aufruf von `super_rosette()` die Stiftfarbe auf rot ein.
- Stelle nach der `reset()`-Anweisung die Hintergrundfarbe des Fensters auf schwarz ein. Dazu gibt es im `turtle`-Modul die Funktion `bgcolor()`. Beispiel:

```
>>> bgcolor("black")
```

Jetzt haben wir noch das Problem mit der Geschwindigkeit. Die ist so langsam, weil ja jedes Sechsunddreißigeck aus 36 Segmenten gezeichnet wird.

Farbenspiel



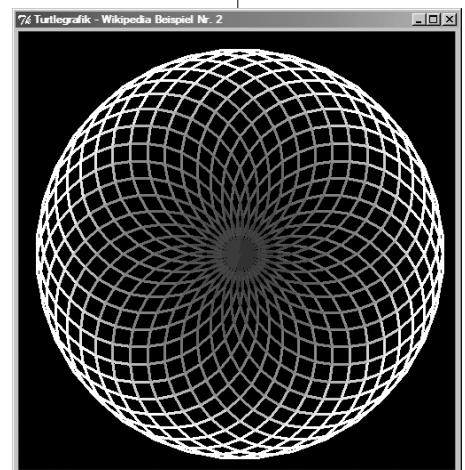
Wir können die Sache erheblich beschleunigen, wenn wir vor dem Zeichnen jedes n-Ecks den Tracer aus- und danach wieder einschalten. Dann kriegen wir nur ein Grafik-Update pro n-Eck und die Zeichnung baut sich viel schneller auf.

```
def rosette(seite, ecken, blaetter):
    for i in range(blaetter):
        tracer(False)
        n_eck(seite, ecken)
        tracer(True)
        left(360/blaetter)
```

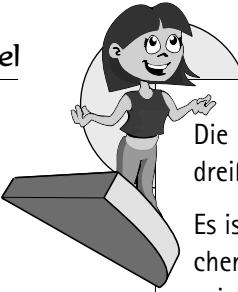
- Baue die tracer()-Anweisungen in die Funktion rosette() ein. Führe das Programm aus und sieh dir die Wirkung dieser beiden Anweisungen an.
- Wir verwenden nun wieder die Funktion title(), mit der der Text in der Titelleiste des Turtle-Grafik-Fensters festgelegt werden kann. Füge die Anweisung title("turtle - Wikipedia Beispiel Nr. 1") als erste Anweisung in das »Hauptprogramm« ein. Führe das Programm nochmals aus, um zu kontrollieren, ob der Fenstertitel richtig erscheint.
- Speichere eine Kopie von neck_arbeit.py als wikipedia_bsp1.py ab.

Farbenspiel

Wir können zwar den verlaufenden Hintergrund der Vorlage mit turtle nicht ganz einfach erzeugen, aber wir wollen versuchen, einen ähnlichen Effekt durch Färbung der Polygon-Streckenzüge zu erreichen. Was ich mir vorstelle, ist auf einer Schwarzweißabbildung nicht leicht darzustellen:



Eine super_rosette mit von blau nach rot verlaufenden Farben.



Die hier heller dargestellten äußeren Teile der Kreise – genauer: Sechsunddreißigecke – sind rot gefärbt, die inneren blau.

Es ist klar, dass wir die Funktion `n_eck()` ändern müssen, um das zu erreichen. Er dürfen nicht mehr alle Seiten des n-Ecks in derselben Farbe gezeichnet werden. Vielmehr muss sich die Farbe der einzelnen Segmente in kleinen Schritten von Blau nach Rot und wieder zurück nach Blau ändern. Wir müssen also in der Funktion `n_eck()` in der for-Schleife vor dem Aufruf von `forward()` einen Aufruf von `pencolor()` einfügen. Aber wie kommen wir zu den passenden Farben? Mit den uns schon bekannten Farbnamen werden wir kaum so feine Farbabstufungen erzielen können. Kein Grund zu verzagen, denn Farben für die Turtles lassen sich auch auf andere Weise festlegen.

Farben durch Zahlen festlegen

Um zu verstehen, wie das geht, musst du wissen, dass die Farbe der Bildpunkte auf einem Computerbildschirm als Mischung von drei Farbanteilen festgelegt wird: Rot, Grün und Blau (oder: red, green, blue). Deshalb spricht man auch von RGB-Farben. Die Größe jedes Anteils wird durch eine Zahl festgelegt. In der Grundeinstellung des `xurtle`-Moduls müssen diese Zahlen im Bereich von 0.0 bis 1.0 liegen. Wie diese Farbmischung funktioniert, können wir uns bequem im IPI ansehen, indem wir zunächst die Turtle vergrößern und dann ausnützen, dass man die Funktion `color()` auch mit drei Zahlen als Argumenten aufrufen kann.

➤ Mach mit!

```
>>> reset()
>>> pensize(30)
>>> color(1,0,0)    # rot
>>> color(0,1,0)    # grün
>>> color(0,0,1)    # blau
>>> color(1,1,0)    # gelb - Komplementärfarbe von blau
```

Wie kriegt man die Komplementärfarben von Rot und Grün?

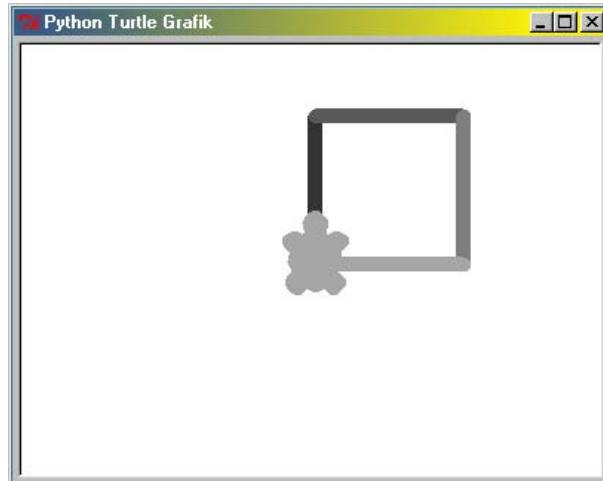
```
>>> color(0,0,0)
>>> color(1,1,1)    # ups! - weiß auf weiß
>>> color(0.95,0.95,0.95) # genau schauen!
```

Jetzt versuchen wir ein paar Rot-Blau-Mischungen!

```
>>> pensize(10)
>>> color(0.8,0,0.2)
>>> fd(100); rt(90)
```

Blau/Rot/Blau in einer Schleife

```
>>> color(0.6, 0, 0.4)
>>> fd(100); rt(90)
>>> color(0.4, 0, 0.6)
>>> fd(100); rt(90)
>>> color(0.2, 0, 0.8)
>>> fd(100); rt(90)
```



Farbmischungen.
Rot ist dunkel, Blau heller
dargestellt.

Blau/Rot/Blau in einer Schleife

Damit können wir uns schon einen Plan für die Färbung unseres 36-seitigen Polygons machen:

- ❖ Wir färben jede Polygonseite mit einer Farbe, bei der der Rot-Anteil und der Blau-Anteil zusammen 1 ergeben. Der Grün-Anteil ist 0. Das machen wir so, dass wir zuerst blau berechnen und daraus $rot = 1 - blau$.
- ❖ Wir beginnen mit einer blauen Seite. Farben: 0, 0, 1.
- ❖ Nach 18 Segmenten (dem halben Polygon) sollten wir bei Rot, also den Farben 1, 0, 0, angelangt sein. Dazu müssen wir von Segment zu Segment den Blau-Anteil um $1.0/18$ verringern. Die Änderung des Blau-Anteils ist eine negative Zahl.
- ❖ Dann geht es umgekehrt weiter, der Blau-Anteil steigt wieder. Die Änderung des Blau-Anteils ist eine positive Zahl, nämlich die entgegengesetzte Zahl von vorher.



❖ Damit müssen wir in n_eck() doch einige Änderungen vornehmen:

```
def n_eck(eckenzahl, seitenlaenge):
    drehwinkel = 360 / eckenzahl
    blau = 1
    aenderung = -2 / eckenzahl
    for i in range(eckenzahl):
        rot = 1 - blau
        pencolor(rot,0,blau)
        forward(seitenlaenge)
        left(drehwinkel)
        if i == eckenzahl // 2:
            aenderung = -aenderung
        blau = blau + aenderung
```

Einiges davon ist einfach zu verstehen: bevor die Schleife beginnt, setzen wir blau auf 1.0 und berechnen die aenderung, die bei jedem Schleifendurchgang erfolgen muss.

Im Schleifenkörper wird rot berechnet und pencolor() aufgerufen. Am Ende des Schleifenkörpers wird blau geändert. Doch was bewirkt die vorletzte Anweisung im Schleifenkörper: die if-Anweisung?

Sie prüft, ob schon etwa die Hälfte der Polygonstrecken gezeichnet ist. Wobei hier der Divisionsoperator // verwendet wird, der eine Ganzahl-Division ausführt. Warum? Die Ganzzahl-Division liefert immer ein ganzzahliges Ergebnis

➤ Mach mit!

```
>>> 36 // 2
18
>>> 35 // 2
17
>>> 35 / 2
17.5
```

Du siehst: bei ungerader Eckenanzahl würde mit der normalen Division für die ganzzahlige Schleifenvariable die Bedingung `i == eckenzahl / 2` nie-mals erfüllt sein.

In unserem Fall ist die Bedingung für den Wert 18 von i wahr. In diesem Schleifendurchgang wird aenderung durch -aenderung ersetzt! Damit ist nun der Wert von aenderung, der vorher negativ war, positiv! Und der Blau-Anteil nimmt wieder zu.

Zusammenfassung

- ⇒ Führe diese Änderung von `n_eck()` in `n_eck_arbeit.py` aus.
- ⇒ Ändere auch die `title()`-Anweisung, sodass sie *Wikipedia Beispiel Nr. 2* anzeigt.
- ⇒ Vielleicht gefällt dir eine größere Strichstärke besser. Probiere das mit einer `pensize()`-Anweisung im Hauptprogramm vor dem Aufruf von `super_rosette()` aus.
- ⇒ Speichere eine Kopie des Programms unter dem Namen `wikipedia_bsp2.py` ab.



Zusammenfassung

In diesem Kapitel hast du folgende Turtle-Grafikfunktionen kennen gelernt:

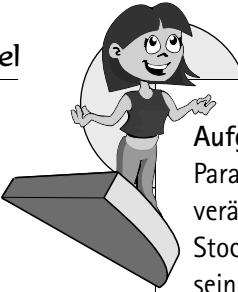
Funktionen aus dem Modul »turtle«

<code>setheading(winkel)</code>	Orientiert die Turtle in Richtung <code>winkel</code> (0° - Nord, 90° - Ost, 180° - Süd, 270° - West)
<code>tracer(True/False)</code>	Dreht die Turtle-Animation an/ab

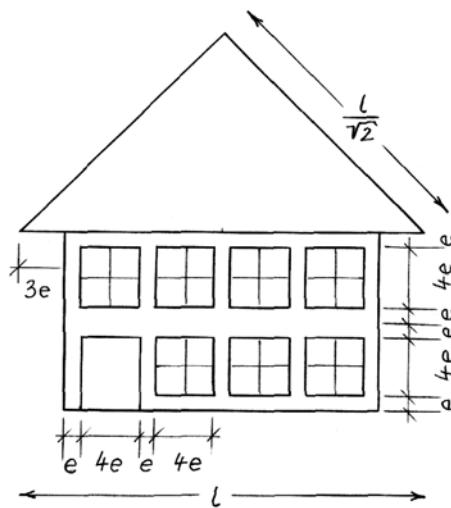
- ❖ Farben können durch eine ganze Latte fantasievoller Farbnamen festgelegt werden (siehe Anhang D).
- ❖ Farben können auch durch drei Zahlenwerte im Bereich 0.0 ... 1.0 für die Grundfarben Rot, Grün und Blau festgelegt werden.
- ❖ Die Anwendung der Funktion `tracer()` kann die Erstellung von Turtle-Grafik-Zeichnungen stark beschleunigen.
- ❖ Module, aus denen Funktionen importiert werden, können ein Hauptprogramm im Körper einer `if __name__ == "__main__":`-Anweisung stehen haben. Dann wird das Hauptprogramm beim Importieren nicht ausgeführt, sondern nur dann, wenn die Datei selbst als Skript ausgeführt wird.

Einige Aufgaben ...

Aufgabe 1: Schreibe eine neue Version des Programms `radioaktiv.py`. Die letzte Fassung von Kapitel 6 enthält einige Anweisungen bzw. Anweisungsfolgen, die wiederholt werden. Setze für diese Wiederholungen `for`-Schleifen ein.



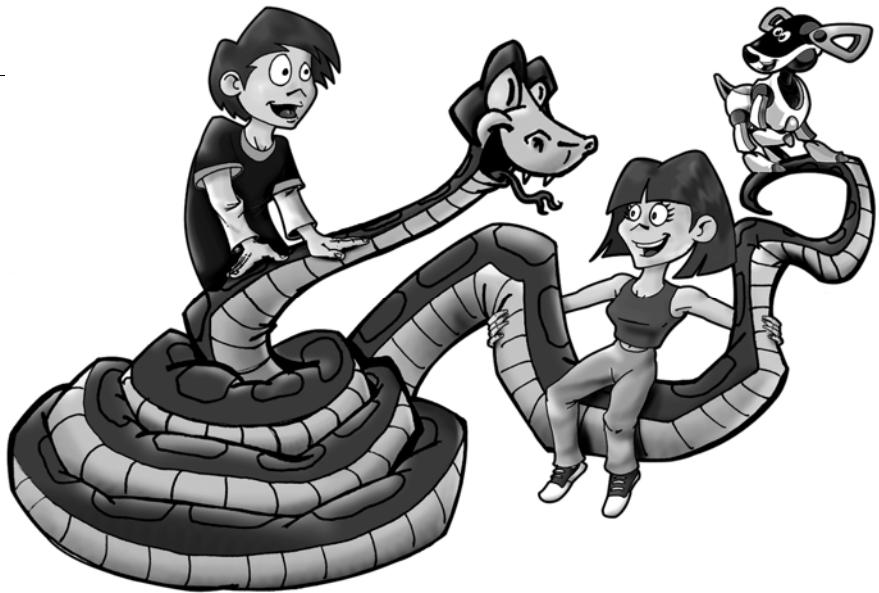
Aufgabe 2: Erstelle eine Funktion `haus()` (mit Standardwerten für die Parameter), die möglichst variable Häuser zeichnen kann, zum Beispiel mit veränderbarer Anzahl der Stockwerke und/oder der Anzahl der Fenster pro Stockwerk. Eine Planungsskizze wie die folgende kann dir dabei hilfreich sein:



... und einige Fragen

1. Welche Farbe ergibt der Funktionsaufruf `color(0.6, 0.0, 0.0)`? Und welche `color(1, 0.6, 0.6)`?
2. Welcher Aufruf von `setheading()` orientiert die Turtle so, dass sie in Richtung des Stundenzigers um genau 7 Uhr zeigt?
3. Welche Möglichkeiten hast du, das Erstellen einer Turtle-Grafik schneller zu machen?

9



Der Zufall und bedingte Schleifen

Wir haben im letzten Kapitel Vielecke, Rosetten und andere Figuren mit Zählschleifen programmiert. Gemeinsam war all diesen Programmen, dass die Anzahl der Wiederholungen in diesen Schleifen vorgegeben war – entweder schon als feste Anzahl, zum Beispiel der Ecken eines Vielecks, oder durch die Anzahl der Elemente einer Sammlung von Dingen, zum Beispiel Farbnamen.

Es gibt aber viele Aufgaben, bei denen der Programmierer nicht weiß, wie viele Durchgänge einer Schleife zu durchlaufen sind. Typischerweise ist das etwa bei Spielprogrammen der Fall, wo mehrere Runden durchlaufen werden müssen und das Ende eines Spiels dann erreicht ist, wenn eine bestimmte *Bedingung* erfüllt ist. In diesem Kapitel werden wir jedoch den Zufall direkt zu Hilfe rufen und unsere Turtle auf von ihm gesteuerten Wege wandeln lassen.

In diesem Kapitel lernst du ...

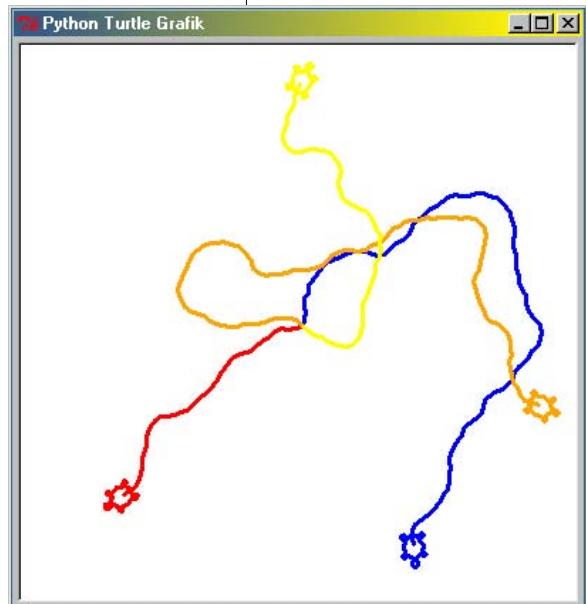
- ◎ ... was Zufallsgeneratoren sind und welche Funktionen dir Python für das Spiel mit dem Zufall bereitstellt.
- ◎ ... wie die Turtle »random walks«, sogenannte Irrfahrten, ausführt.
- ◎ ... die Ausführung von Schleifen durch Bedingungen zu steuern.
- ◎ ... zwei Turtle-Grafik-Funktionen, mit denen die Turtle ihre Position feststellen und Entfernungen messen kann. Und eine, mit der sie Fußabdrücke hinterlassen kann.

9



Eine torkelnde Turtle

Bis jetzt haben wir Dreiecke, Vierecke, Polygone, Rosetten (auch Super-), Kreise, Kreisbögen untersucht, und was man alles daraus machen kann: schön regelmäßige geometrische Figuren. Jetzt ist es Zeit für etwas Neues: etwas richtig Chaotisches, Zufälliges.



Ich stelle mir eine Turtle vor, die nicht immer in die Richtung weiterläuft, die wir ihr vorschreiben, sondern (mehr oder weniger) in eine zufällige Richtung. Vielleicht hat sie zu tief ins Glas geschaut oder sie hat Schwindelanfälle wegen zu niedrigen Blutdrucks. Egal.

Mögliche Wege der torkelnden Turtle.

Um so etwas in ein Programm einzubauen, brauchen wir eine Funktion, die uns so genannte Zufallszahlen erzeugt. Python hat ein Modul, das sich mit Zufallsdingen befasst und mehrere solcher Funktionen enthält.

Dieses Modul heißt `random`. Bevor wir uns ans Programmieren der torkelnden Turtle machen, sehen wir uns eine derartige Funktion genauer an.

Der Zufallsgenerator »randint()«

`randint()` ist eine Funktion aus dem Modul `random`. »Random« ist ein englisches Wort und bedeutet so viel wie zufällig.

Die Funktion `randint()` liefert ganze Zufallszahlen aus einem wählbaren Bereich:

⇒ Mach mit!

```
>>> import random  
>>> random.randint(1,10)
```

Der Zufallsgenerator »randint()«



Wir haben hier einfach nur das Modul importiert. Wir können nun alle Funktionen aus random anwenden, indem wir random.Funktionsnamen schreiben – wie wir es eben mit randint() getan haben.

Hier eine Life-Session mit dem IPI:

```
IPI Shell / Turtle Grafik
File Edit Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32
bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>> import random
>>> random.randint(1, 10)
5
>>> random.randint(1, 20)
1
>>> random.randint(1, 30)
5
>>> |
```

Dieses Ergebnis ist sonderbar, aber eigentlich herzlich nichts sagend. Deines ist vielleicht nicht ganz so sonderbar. Aber über Zufallsergebnisse lässt sich aus Einzelereignissen nichts sagen. In der Dokumentation zum Modul random steht: randint(a,b) erzeugt eine Zufallszahl z im Bereich $a \leq z \leq b$. Um das zu erahnen, müssen wir mit mehr Zahlen experimentieren:

```
>>> for i in range(20):
    print(random.randint(1,6), end=" ")
```

4 6 3 4 6 6 6 3 6 2 6 6 2 3 4 3 6 2 2 4

Hier kommen die Zahlen 1 und 5 nicht vor. Merkwürdiger Zufall! (?) Ein zweiter gleicher Aufruf liefert mit einer ganz anderen Zufallsfolge Klarheit darüber:

```
>>> for i in range(20):
    print(random.randint(1,6), end=" ")
```

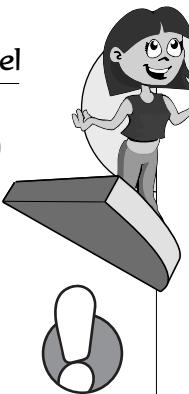
4 4 2 6 6 5 6 4 3 1 1 1 5 4 4 3 3 4 2

```
IPI Shell / Turtle Grafik
File Edit Debug Options Windows Help
>>> random.randint(1, 30)
5
>>> for i in range(20):
    print(random.randint(1, 6), end=" ")

4 6 3 4 6 6 6 3 6 2 6 6 2 3 4 3 6 2 2 4
>>> for i in range(20):
    print(random.randint(1, 6), end=" ")

4 4 2 6 6 5 6 4 3 1 1 1 5 4 4 3 3 4 2
>>> |
```

Ein Zufallsgenerator kann durchaus sehr merkwürdige Ergebnisse liefern!



Diese for-Schleife mit 20 Aufrufen des Zufallsgenerators `randint(1,6)` liefert gerade die Ergebnisse von 20 Mal würfeln. Kann durchaus sein, dass 20 Mal keine 1 dabei ist. Oder auch keine 6.

Die Funktion `randint(a, b)` aus dem Modul `random` liefert eine ganze Zufallszahl aus dem Bereich $a \dots b$. (Die Randwerte a und b gehören zum Bereich dazu.)

Wenn wir Zufallszahlen im Bereich 1 bis 100 erzeugen wollen, müssen wir also `random.randint(1,100)` aufrufen. Ebenso lassen sich Bereiche nutzen, die negative Zahlen enthalten:

```
>>> for i in range(20):
    print(random.randint(-10,10), end=" ")
-10 9 9 3 -2 0 -6 -9 -1 0 8 -9 5 8 9 -7 -8 -5 5 -9
```

Torkeln

Die Idee ist ziemlich einfach: jedes Mal, bevor die Turtle ein Stück vorwärts geht, wählt sie zufällig die Richtungsänderung vor diesem Schritt aus einem gegebenen Bereich aus.

➤ Mach mit!

```
>>> from turtle import *
>>> reset(); pensize(3)
>>> winkel = random.randint(-30,30)
>>> left(winkel); forward(15)
>>> winkel = random.randint(-30,30)
>>> left(winkel); forward(15)
>>> winkel = random.randint(-30,30)
... und noch ein paar Mal. Tatsächlich: die Turtle torkelt!
```

➤ Wenn du Lust hast, kannst du nun noch mit anderen Winkelbereichen und Schrittgrößen experimentieren.

Wir wollen jetzt ein erstes Programm schreiben, das unsere Turtle auf solchen Zufallswegen wandern lässt. Für derartige Spaziergänge gibt es im Englischen einen Spezialausdruck, den wir hier verwenden wollen: »random walk«. Im Deutschen sagt man auch »Irrfahrten«.

Torkeln

- Öffne ein neues Editor-Fenster, schreibe einen Kopfkommentar für das Skript randomwalk_arbeit.py und speichere die Datei im Ordner c:\py4kids\kap09 ab.

Wir beginnen ganz einfach. Wir schreiben eine Funktion zufalls schritt() und eine Schleife, die die Turtle einige solche Zufallsschritte machen lässt. Das Ganze ergänzen wir mit etwas Code für die Herstellung der richtigen Anfangssituation.

```
from turtle import *
import random

def zufallsschritt():
    winkel = random.randint(-30,30)
    left(winkel)
    forward(15)

setup(400,400)
penup()
home()
pendown()
pensize(3)

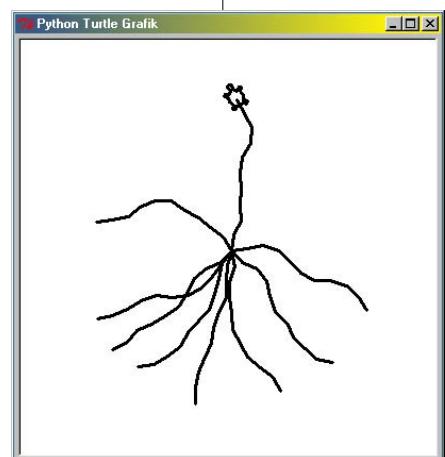
startwinkel = random.randint(0,359)
right(startwinkel)

for n in range(10):
    zufallsschritt()
```

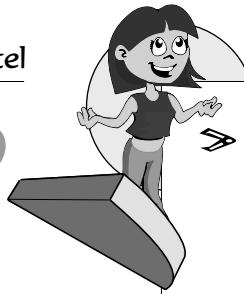
Wenn wir dieses Programm mehrere Male ausführen, wird die Turtle mehrere Zufallswege ins Grafik-Fenster zeichnen. Damit diese nicht alle in die gleiche Richtung starten, wird auch die Startrichtung zufällig ausgewählt. Jeder Schritt der torkelnden Turtle beginnt mit einer Drehung um einen zufälligen »Torkelwinkel« im Bereich von -30° bis 30°. Nach neun Programmläufen sah das bei mir einmal so aus:

- Speichere eine Kopie dieses Programms als randomwalk01.py.

Du wirst dir sicherlich denken, dass dieses Programm noch sehr Verbesserungsfähig ist. Richtig! Arbeiten wir ein, was wir bisher schon alles gelernt haben.



9



➤ Bei zufallsschritt() spielen der maximale Torkelwinkel (hier: 30) und die Schrittgröße (hier: 15) eine Rolle. Baue diese beiden Größen als Parameter der Funktion ein. Dann hast du eine viel flexiblere Funktion zufallsschritt().

Wie du gesehen hast, ist es interessant, mehrere solcher Zufallswege zu zeichnen. Erstelle daher eine Funktion zufallsweg(), die die Turtle einen solchen Zufallsweg – das sind mehrere Zufallsschritte – zurücklegen lässt. Als Parameter drängt sich die Anzahl der Zufallsschritte auf, und Torkelwinkel und Schrittgröße werden sicher auch gebraucht, da ja im Funktionskörper von zufallsweg() die Funktion zufallsschritt() aufgerufen wird.

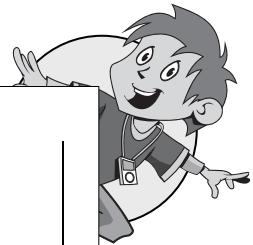
- Schreibe entsprechend dieser Beschreibung die Funktion zufallsweg().
- Die übrigen Anweisungen gehören zum Teil in die Funktion zufallsweg() und zum Teil in den Anweisungsblock einer if __name__ == '__main__': -Anweisung. Schreibe das »Hauptprogramm« so, dass die Turtle zehn Zufallswege mit je zehn Zufallsschritten erzeugt.
- Teste dein Programm, und wenn es fertig gestellt ist, speichere eine Kopie davon als randomwalk02.py ab.

Bei der Ausführung der obigen Schritte hast du einiges an Freiheiten, wie z.B. die Wahl der Variablennamen, in gewissen Grenzen die Reihenfolge der Anweisungen usw. Es folgt hier eine mögliche Lösung, als Hilfestellung und zum Vergleich. Deine wird naturgemäß davon abweichen, doch sie sollte etwa die gleiche Grafik erzeugen. (Du findest meine Lösung, wie immer, auch auf der Buch-CD).

```
from turtle import *
import random

def zufallsschritt(laenge, maxwinkel):
    winkel = random.randint(-maxwinkel,maxwinkel)
    left(winkel)
    forward(laenge)

def zufallsweg(schritt_anzahl, schrittlaenge, torkelwinkel):
    pu(); home(); pd()
    startwinkel = random.randint(0,359)
```



```
right(startwinkel)
for i in range(schritt_anzahl):
    zufallsschritt(schrittlaenge, torkelwinkel)

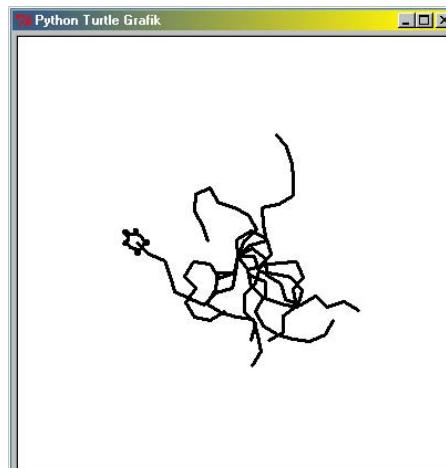
def randomwalk_test(anzahl_walks):
    reset()
    setup(400,400)
    speed(0)
    pensize(3)
    for i in range(anzahl_walks):
        zufallsweg(10, 15, 30)

if __name__ == "__main__":
    randomwalk_test(10)
```

Diese Programm verlockt natürlich dazu, an den Parametern von `zufallsweg()` zu »drehen«, das heißt, die drei Zahlen in der dritten Codezeile von unten zu ändern.

- Stelle den maximalen Winkel, also das dritte Argument, auf 90. (Da ist die Turtle schon ziemlich beduselt.) Führe das Programm aus. Die Wege sind jetzt viel verzweigter. Und die Turtle entfernt sich nicht mehr so weit vom Ausgangspunkt.

Bei mir ergab ein Programmlauf folgendes Bild:



Zehn random walks mit maximalem Torkelwinkel von 90°.

- Stelle den maximalen Winkel, also das dritte Argument, auf 180. (Nun hat die Turtle maximale Winkelfreiheit!) Führe das Programm aus. Die Turtle kann sich in diesem Zustand der Desorientierung noch weniger vom Ausgangspunkt entfernen.

9

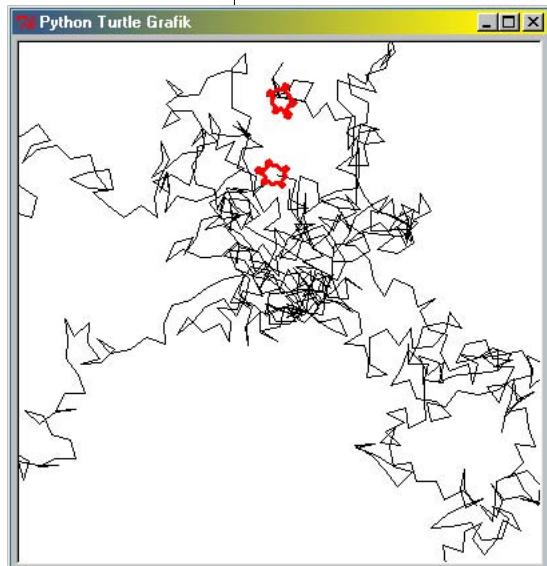


Wir können daher die Turtle eine größere Anzahl von Schritten durchführen lassen, ohne dass sie von der Bildfläche verschwindet.

⇒ Versuche eine `schritt_anzahl` von 30 (anstelle von 10). Stelle vor dem Programmlauf die Strichdicke auf 1.

Die Situation ist jetzt etwas unübersichtlich. Wo enden die einzelnen Wege? Um dies zu sehen, erzeugen wir am Ende jedes Weges einen Abdruck der Turtle mit der Funktion `stamp()`:

```
def randomwalk_test(anzahl_walks):
    reset()
    setup(400,400)
    speed(0)
    pensize(1)
    for i in range(anzahl_walks):
        pencolor("black")
        zufallsweg(30, 15, 180)
        pencolor("red")
        stamp()
```



- ⇒ Führe diese Änderungen aus und lasse das Programm diesmal mit weniger Wegen, sagen wir mit sechs, laufen:
- ⇒ Du siehst, dass die Turtle immer noch nicht weit weg kommt. Versuche es – mutiger – vielleicht mit fünf Mal so viel Schritten: Schrittzahl 150.

Vier von sechs Turtles haben nach 150 Schritten auf ihrer Irrfahrt das Grafik-Fenster verlassen.

Das war etwas zu gewagt. Gleich vier von den sechs Turtles sind uns entkommen. Das bringt uns auf eine neue **Fragestellung**: Wie viele Schritte muss eine torkelnde Turtle im Durchschnitt machen, um sich 200 Einheiten (die halbe Fensterbreite) vom Ausgangspunkt zu entfernen?

»while«-Schleifen

Um das festzustellen, müssen wir unsere Funktion zufallsweg() so umprogrammieren, dass die Turtle stehen bleibt, sobald sie 200 Einheiten vom Ausgangspunkt (0,0) entfernt ist. Dazu benötigt sie aber für jeden Weg eine andere Anzahl von Schritten. Die Schleife muss abgebrochen werden, wenn die Turtle weit genug weg ist, also wenn eine bestimmte Bedingung eingetreten ist. Das geht mit einer einfachen for-Schleife nicht gut. Für solche Zwecke gibt es in Python eine zweite Art von Schleifen. Um diese geht es jetzt, bevor wir mit dem randomwalk-Skript weiter machen:

»while«-Schleifen

Immer wenn wir nicht wissen, wie oft die Schleife durchlaufen werden soll, brauchen wir die Möglichkeit, mit einer Bedingung zu überprüfen, ob ein weiterer Schleifendurchgang erfolgen soll.

In Python gibt es dafür die while-Schleife. Um sie kennen zu lernen, werfen wir wieder den IPI an:

➤ Mach mit!

```
>>> zaehler = 0  
>>> while zaehler < 5:  
        print(zaehler, end=" ")  
        zaehler = zaehler + 1
```

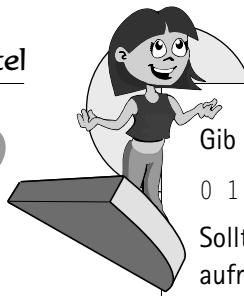
0 1 2 3 4

Die while-Schleife hat eine Form, die du schon kennst: Die erste Zeile ist ein Anweisungskopf, der mit einem Doppelpunkt : endet, dann folgt ein eingerückter Anweisungskörper. Die Ausführung der while-Schleife erfolgt so: Zunächst wird geprüft, ob die Bedingung, die nach dem reservierten Wort while steht, erfüllt ist. Ist das der Fall, dann wird der Schleifenkörper ausgeführt. Danach wird die Bedingung neuerlich geprüft, der Schleifenkörper neuerlich ausgeführt usw.

Erst wenn die Bedingung nicht erfüllt ist, wird die Schleife abgebrochen. Beachte, dass der Schleifenkörper gar nicht ausgeführt wird, wenn schon bei der ersten Prüfung die Bedingung nicht erfüllt ist!



9



Gib nun im IPI eine while-Schleife ein, die nur Folgendes ausgibt:

0 1 2

Sollte dir das auf Anhieb gelungen sein, dann bist du ein ungewöhnlich aufmerksamer Leser. Auf welchen Wert verweist bei dir jetzt zaehler?

```
>>> zaehler
```

3

Beachte: Wenn eine Bedingung wie zaehler < 5 geprüft wird, muss zaehler bereits einen Wert haben, sonst erfolgt ein NameError.

Daher hatten wir bei obiger Übung zunächst die Anweisung zaehler=0 eingegeben. In der folgenden Schleife wurde bei jedem Durchlauf zaehler um 1 erhöht, bis der Wert von zaehler schließlich 5 war. Da 5 < 5 eine falsche Aussage ist, wurde an dieser Stelle die Schleife abgebrochen.

Schreibst du danach im IPI:

```
>>> while zaehler < 3:  
        print(zaehler, end=" ")  
        zaehler = zaehler + 1
```

>>>

so bleibt die Ausgabe aus. Denn zaehler hat noch immer den Wert 5 und ich erwähnte schon, dass der Schleifenkörper nicht ausgeführt wird, wenn die Bedingung schon bei der ersten Prüfung nicht erfüllt ist.

Wenn du eine while-Schleife nochmals ausführen willst, dann musst du vorher wieder passende Anfangsbedingungen herstellen. Hier geht das so:

```
>>> zaehler = 0  
>>> while zaehler < 3:  
        print(zaehler, end=" ")  
        zaehler = zaehler + 1
```

0 1 2

>>>

➤ Überlege, bevor du die folgende Eingabe machst, was hier das Ergebnis sein wird:

»while«-Schleifen

```
>>> zaehler = 0  
>>> while zaehler < 3:  
    zaehler = zaehler + 1  
    print(zaehler, end=" ")
```

? ? ?

Auch hier hat der Name `zaehler` bei Beginn des letzten Schleifendurchlaufs den Wert 2. Doch wird der Wert geändert, nämlich um 1 erhöht, bevor die `print()`-Funktion aufgerufen wird. Und so kommt es zur Ausgabe 1 2 3.

Der Name `zaehler` spielt in dieser `while`-Schleife eine besondere Rolle, weil er in der Schleifenbedingung vorkommt. Man nennt ihn deshalb auch oft die Schleifenvariable. Es ist entscheidend, dass der Wert dieser Schleifenvariablen bei der Ausführung des Schleifenkörpers geändert wird, obwohl das nicht unbedingt bei jedem Schleifendurchgang der Fall sein muss. Andernfalls kann die Bedingung im Schleifenkopf niemals falsch werden und du hast es mit etwas sehr Unangenehmem zu tun: mit einer *Endlosschleife*.

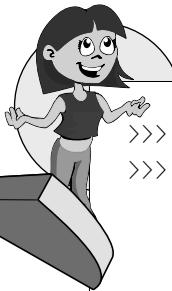
Endlosschleifen bricht man in der IDLE mit `Strg+C` ab!

```
>>> zaehler = 0  
>>> while zaehler < 3:  
    print(zaehler, end=" ")  
  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...  
...  
... 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
Traceback (most recent call last):  
  File "<pyshell#3>", line 2, in <module>  
    ...  
      raise KeyboardInterrupt  
KeyboardInterrupt  
>>>
```

Noch häufiger ergeben sich Endlosschleifen aus dem Fehler, dass die Schleifenvariable »in die falsche Richtung« abgeändert wird:



9



```
>>> zaehler = 0
>>> while zaehler < 3:
    zaehler = zaehler - 1
    print(zaehler, end=" ")
-1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17 -
18 -19 -20 -21 -22 -23 -24 -25 -26 -27 -28 -29 -30 -31 -32 -
33 -34 -35 -36 -37 -38 -39 -40 -41 -42 -43 -44 -45 -46 -47
Traceback (most recent call last):
...
```

```
raise KeyboardInterrupt
KeyboardInterrupt
>>>
```

Hier bleibt der Wert von zaehler für immer kleiner als 3.

Leider gibt es bei der Arbeit mit der IPI-SHELL / TURTLEGRAFIK auch Endlos-schleifen, die sich mit **[Strg]+[C]** nicht abbrechen lassen. In diesem Fall muss IDLE beendet und neu gestartet werden.

Ein Beispiel dafür ergibt sich, wenn du in obiger Schleife die print-Anweisung weglässt.

Das Folgende tippe bitte *nur dann* ein, wenn du üben willst, wie man mit einem »abgestürzten« IPI umgeht:

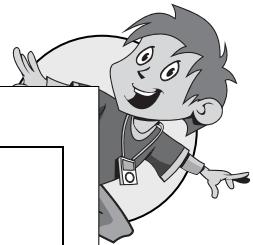
```
>>> zaehler = 0
>>> while zaehler < 3:
    zaehler = zaehler - 1
```

Nun reagiert der IPI nicht mehr auf deine Aktionen. Du kannst ihn nur schließen, indem du auf das Icon mit dem X (Fensterecke rechts oben) klickst. Im darauf folgenden Dialog klicke auf SOFORT BEENDEN und dann auf NICHT SENDEN. Dann starte zähneknirschend IPI-TURTLEGRAFIK neu!

Mit der IDLE (PYTHONGUI) Variante der Python-Shell besteht dieses Problem nicht. Hier kann die Ausführung jeder Anweisung in der Shell und auch die Ausführung jedes Programms stets durch **[Strg]+[C]** abgebrochen werden.

Aufgabe: Schreibe ein kleines Programm `countdown.py`, das folgende Ausgabe erzeugt:

10 9 8 7 6 5 4 3 2 1 START!!!



Muster 12: Bedingte Schleife (while-Schleife)

Zweck: Solange eine Bedingung erfüllt ist, eine Folge von Anweisungen wiederholt ausführen.

Im Programmentwurf:

Initialisiere Schleifenvariablen, die in bedingung vorkommen

Solange bedingung erfüllt ist wiederhole:

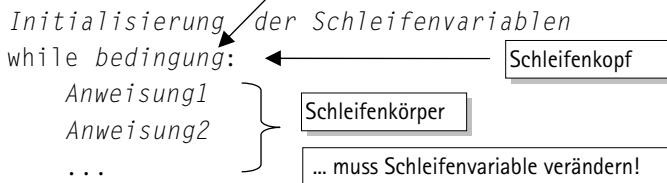
Anweisung 1

Anweisung 2

...

bedingung muss auswertbar sein!

Im Programmcode:



Zurück zum random walk

Wir wollen jetzt herausfinden, wie viele Schritte die Turtle auf ihrem Zufallsweg im Durchschnitt ausführen muss, bis sie eine bestimmte Entfernung vom Ausgangspunkt erreicht hat. Probleme dieser Art sind von bedeutenden Forschern untersucht worden. Begonnen hat es um 1828, als der Biologe Robert Brown unter dem Mikroskop die später nach ihm benannte Bewegung kleiner Pflanzenpollen in einem Wassertropfen untersuchte. Genaueres zur Brown'schen Bewegung findest du zum Beispiel hier:

http://schulen.eduhi.at/riedgym/Physik/10/waerme/temperatur/brownsche_bewegung.htm

1905 hat Albert Einstein in seinem *annus mirabilis* – seinem »Wunderjahr« – klären können, wie es zu dieser Bewegung kommt.

So betrachtet sieht die ganze Sache schon etwas ernsthafter aus als nur eine »torkelnde Turtle«.

➤ Starte IPI-TURTLEGRAFIK und lade randomwalk_arbeit.py in ein Editor-Fenster.

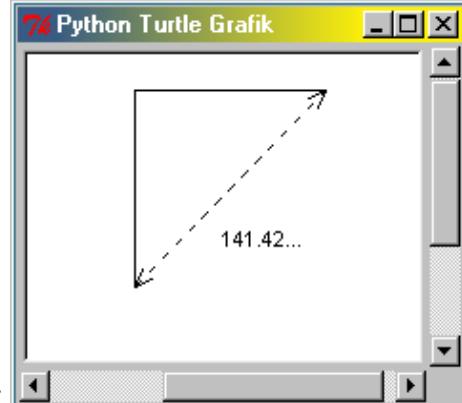


Wir werden jetzt eine Funktion `randomwalk()` hinzufügen, die so wie `zufallsweg()` arbeitet, nur mit dem Unterschied, dass die Anzahl der Schritte nicht angegeben wird, sondern der Zufallsweg abgebrochen wird, wenn eine bestimmte Entfernung vom Ausgangspunkt überschritten wird. Das läuft im Wesentlichen darauf hinaus, die `for`-Schleife durch eine `while`-Schleife zu ersetzen.

Allerdings müssen wir noch klären, wie wir die Entfernung der Turtle zum Startpunkt feststellen können. Dabei helfen uns zwei Funktionen aus dem `turtle`-Modul: die Funktion `position()`, die uns die aktuelle Position der Turtle zurückgibt, und die Funktion `distance()`, die die Entfernung der Turtle zu einem gegebenen Punkt zurückgibt.

» Mach mit!

```
>>> from turtle import *
>>> position()
(0.00,0.00)
>>> start = position()
>>> start
(0.00,0.00)
>>> distance(start)
0.0
>>> forward(100)
>>> distance(start)
100.0
>>> rt(90); fd(100)
>>> distance(start)
141.4213562373095
```



`distance()` misst Entfernungen.

```
>>> rt(90); fd(100)
>>> position()
(100.00,0.00)
>>> distance(start)
100.00000000000001
>>> distance(50,50)
70.710678118654769
```

Zurück zum random walk

Damit können wir leicht eine erste Fassung von `randomwalk()` erstellen:

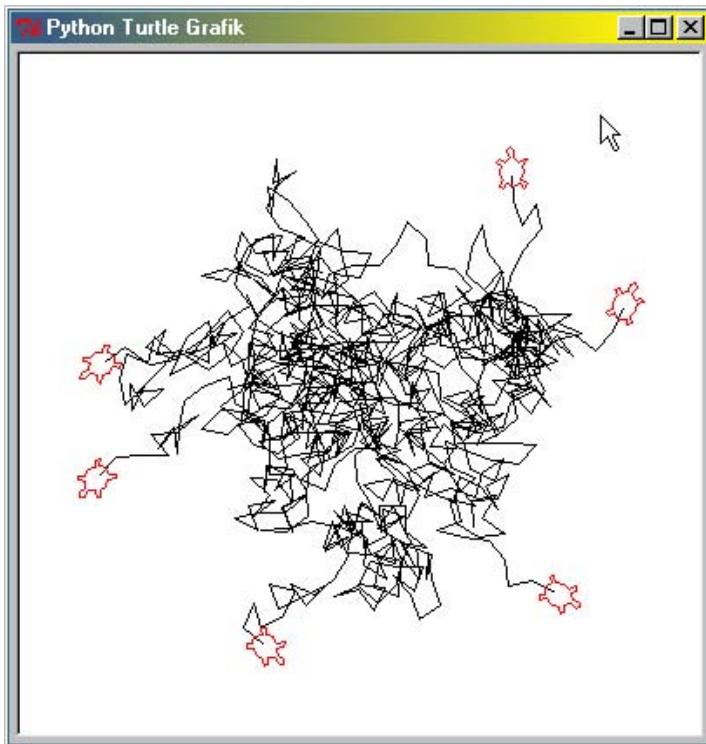
```
def randomwalk(entfernung, schrittlaenge, maxwinkel=180):
    pu(); home(); pd()
    startwinkel = random.randint(0,359)
    right(startwinkel)
    start = position()
    while distance(start) < entfernung:
        zufallsschritt(schrittlaenge, maxwinkel)
```



- Füge den Code von `randomwalk()` in dein Programm ein und entferne den von `zufallsweg()`.
- Ersetze im Hauptprogramm den Aufruf von `zufallsweg()` durch den folgenden Aufruf von `randomwalk()`, bei dem als maximaler Winkel der Standardwert 180 verwendet wird.

```
randomwalk(150, 15)
```

Damit hast du den ersten Teil unseres Problems schon gelöst.



Die Turtle stoppt ihre Irrfahrt in einer vorgegebenen Entfernung vom Startpunkt.

- Speichere eine Kopie dieses Programms als `randomwalk03.py` ab.



Kaum lernst du etwas Neues, hast du es auch schon mit einer Ausnahme zu tun: Ist dir aufgefallen dass in der `while`-Schleife dieses Programms keine Schleifenvariable geändert wird. Es geht eben *doch* auch anders: hier ändert sich der *Zustand* der Grafik, in diesem Fall die Lage der Turtle. Und diese wird in der Bedingung für den nächsten Schleifendurchgang mittels der Funktion `distance()` abgefragt.

Wenn du den Ablauf dieser Programms verfolgst, wirst du sehen, dass die Irrfahrten der Turtle sehr unterschiedliche Länge haben können. Und den durchschnittlichen Wert dieser Längen wollen wir auch noch feststellen. Ermitteln wir zunächst die Weglängen für die einzelnen Irrfahrten:

```
start = position()
schritte = 0
while distance(start) < entfernung:
    zufallsschritt(schrittlaenge, maxwinkel)
    schritte = schritte + 1
print("Anzahl der Schritte:", schritte)
```

➤ Führe diese Änderungen in `randomwalk_arbeit.py` ein und führe das Programm aus. Du erhältst eine Ausgabe wie diese:

```
Anzahl der Schritte: 38
Anzahl der Schritte: 89
Anzahl der Schritte: 63
Anzahl der Schritte: 164
Anzahl der Schritte: 78
Anzahl der Schritte: 153
```

Aber jeder Programmlauf liefert ein anderes Bild und andere Zahlen. Um den Durchschnittswert der erforderlichen Schritte zu erhalten, müssen wir alle diese Schritte zusammenzählen und dann durch die Anzahl der Irrfahrten dividieren. Ein ähnliches Problem hatten wir schon: die Punkte bei `miniquiz.py`. Wir lösen es hier ähnlich:

- Führe eine globale Variable `schrittsumme` ein. Sie muss in den Funktionen `randomwalk()` und `randomwalk_test()` als global deklariert werden.
- Initialisiere in `randomwalk_test()` die Variable `schrittsumme` (vor der `for`-Schleife) mit 0.
- Füge am Ende von `randomwalk()` eine Anweisung ein, die `schrittsumme` jeweils um `schritte` erhöht.

Zurück zum random walk

- ☞ Füge am Ende von `randomwalk_test()` zwei Anweisungen ein, die den Durchschnittswert der Schritte berechnen, indem `schrittsumme` durch die Anzahl der Irrfahrten dividiert wird, und dann diesen Durchschnittswert mit einer `print`-Anweisung ausgeben.

Das Ergebnis eines Programmablaufs sieht jetzt etwa so aus:

```
Anzahl der Schritte: 54  
Anzahl der Schritte: 96  
Anzahl der Schritte: 112  
Anzahl der Schritte: 68  
Anzahl der Schritte: 102  
Anzahl der Schritte: 40  
Durchschnittliche Schrittanzahl: 78.6666666667
```

Bei mir sieht der Code der beiden zuletzt geänderten Funktionen so aus:

```
def randomwalk(entfernung, schrittlaenge, maxwinkel=180):  
    global schrittsumme  
    pu(); home(); pd()  
    startwinkel = random.randint(0,359)  
    right(startwinkel)  
    start = position()  
    schritte = 0  
    while distance(start) < entfernung:  
        zufallsschritt(schrittlaenge, maxwinkel)  
        schritte = schritte + 1  
    schrittsumme = schrittsumme + schritte  
    print("Anzahl der Schritte:", schritte)  
  
def randomwalk_test(anzahl_walks):  
    global schrittsumme  
    reset()  
    setup(400,400)  
    speed(0)  
    pensize(1)  
    schrittsumme = 0  
    for i in range(anzahl_walks):  
        pencolor("black")  
        randomwalk(150, 15)  
        pencolor("red")  
        stamp()  
    durchschnitt = schrittsumme/anzahl_walks  
    print("Durchschnittliche Schrittanzahl:",durchschnitt)
```

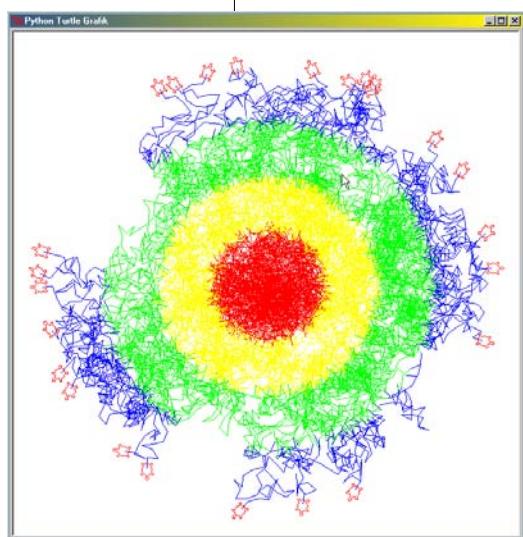


- Wenn dir das Programm zu langsam läuft und/oder du vielleicht längere Testreihen mit mehr als sechs Irrfahrten durchführen möchtest, dann füge in der for-Schleife von randomwalk_test() vor dem Aufruf von randomwalk() die Anweisung tracer(False) und danach die Anweisung tracer(True) ein.
- Für längere Testreihen ersetze im Aufruf von randomwalk_test() in der letzten Zeile die 6 durch eine größere Zahl.
- Teste das Programm und speichere eine Kopie davon als randomwalk04.py.

Farbige Irrfahrten

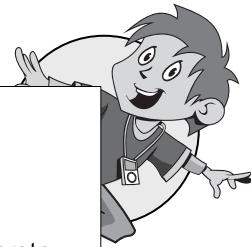
Es ist jetzt eine leichte Übung, die Funktion randomwalk() in randomwalk_arbeit.py so abzuändern, dass die Irrfahrten der Turtle farbige Spuren hinterlassen. Diese könnten beispielsweise mit zunehmender Entfernung ihre Farbe ändern, etwa bis zu einer Entfernung von 75 rot, bis 150 gelb, bis 225 grün und danach blau.

- Ändere den setup()-Aufruf so ab, dass das Grafik-Fenster die Größe 700 × 700 Pixel hat.
- Füge in der while-Schleife von randomwalk() eine if-elif-else-Anweisung ein, die vor dem zufallsschritt() die Stiftfarbe entsprechend der Entfernung vom Startpunkt einstellt.



- Ändere die Abbruchbedingung für die while-Schleife so, dass die Irrfahrt erst bei einer Entfernung von 300 Einheiten endet.
- Teste das Programm und speichere eine Kopie davon als randomwalk05.py ab.

Farbige Irrfahrten, in diesem Bild mit durchschnittlich 507 Schritten.



Zusammenfassung

- ❖ Zur Steuerung von zufälligen Vorgängen braucht man Zufallsgeneratoren. Python hat Zufallsgeneratoren im Modul random.
- ❖ Ein Zufallsgenerator ist die Funktion randint(von, bis). Sie liefert ganzzählige Zufallszahlen im Bereich von ... bis.
- ❖ Mit der Schleife while *bedingung*: wird ein Block von Anweisungen wiederholt, solange die angegebene *bedingung* wahr ist.
- ❖ Bei der while-Schleife ist entscheidend, dass die Bedingung für die Ausführung des Schleifenkörpers einmal falsch wird, damit die Schleife abbricht.
- ❖ Der Abbruch der Schleife kann erreicht werden, indem eine Schleifenvariable verändert wird, damit die *bedingung* schließlich falsch wird.
- ❖ Vor der eigentlichen while-Schleife muss der Schleifenvariablen ein Anfangswert zugewiesen werden.
- ❖ Als Schleifenbedingungen können wie bei Verzweigungen Vergleiche mit ==, !=, <, <=, >, >= verwendet werden.
- ❖ Der Abbruch der Schleife kann auch erreicht werden, indem sich der Zustand des Programms so ändert, dass diese Änderung mit einer Funktion in der *bedingung* abgefragt werden kann.
- ❖ Unterläuft dir versehentlich eine Endlosschleife, brich sie mit **Strg+C** ab. In manchen Situationen funktioniert das nicht. Dann musst du die IDLE beenden und neu starten.

Neue Funktionen aus dem Modul »turtle«

position()	Gibt die Koordinaten der aktuellen Position der Turtle aus. Der Punkt (0,0) ist im Mittelpunkt des Grafikfensters.
distance(x,y) distance(p)	Gibt die Entfernung des Punktes (x,y) oder des Punktes p von der Turtle zurück.
stamp()	Erzeugt einen »Stempelabdruck« der Turtle-Gestalt auf der Zeichenfläche.

9



Einige Aufgaben ...

Aufgabe 1: Manhattan-Walk: Ändere `randomwalk03.py` so ab, dass sich die Turtle nur in eine der Richtungen 0° , 90° , 180° oder 270° bewegen kann. Bei jedem Schritt wählt sie eine dieser Richtungen zufällig aus. Sie bewegt sich dann wie auf einem Zufallsweg in einem rechtwinkeligen Straßennetz. Speichere dein Programm als `manhattanwalk.py`.

Aufgabe 2: Braucht die Turtle beim Manhattan-Walk im Durchschnitt mehr oder weniger Schritte als beim gewöhnlichen random walk, um eine bestimmte Entfernung vom Ausgangspunkt zu erreichen?

Aufgabe 3: Der Funktionsaufruf `randint(1,6)` liefert eine Zufallszahl im Bereich 1..6. Damit kann man gut das Verhalten eines Spielwürfels im Computer nachbilden. (a) Wie oft muss im Durchschnitt gewürfelt werden, bis zwei Mal eine 6 aufgetreten ist? (b) Wie oft muss im Durchschnitt gewürfelt werden, bis zwei Mal hintereinander eine 6 aufgetreten ist?

... und einige Fragen

1. Welche Programmierfehler können zu Endlosschleifen führen?
2. Wie werden Endlosschleifen (im günstigen Fall) abgebrochen?
3. Welche Vergleichsoperatoren können in der Bedingung einer while-Schleife verwendet werden?
4. Warum heißt das Jahr 1905 Albert Einsteins »Wunderjahr«?

10



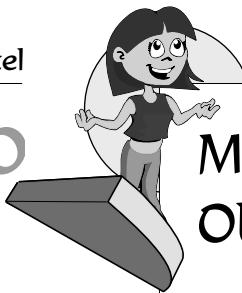
Funktionen mit Wert

Seit dem dritten Kapitel definierst du nun schon Funktionen in deinen Programmen. Die ersten, die du verwendet hast, waren aber nicht selbst definierte, sondern bereits fix in Python eingebaute oder in einem mitgelieferten Modul definierte Funktionen.

Die meisten dieser Funktionen haben Aktionen ausgeführt. Zum Beispiel die Funktion `print()`. Sie schreibt Zeichen auf den Bildschirm. Oder die Funktionen `forward()`, `left()` und `right()` aus dem Modul `turtle`. Sie bewegen die Turtle. Du hast aber auch Funktionen verwendet, die Werte ermittelt und zur weiteren Verarbeitung zurück gegeben haben. Zum Beispiel `len()`, `range()`, `input()` oder auch `sqrt()` aus dem Modul `math`. Auch die Funktionen `position()` und `distance()` aus dem Modul `turtle` haben keine Aktionen bewirkt, sondern Werte zurückgegeben.

In diesem Kapitel erfährst du ...

- ⌚ wie du selbst Funktionen definieren kannst, die Objekte ermitteln und zurückgeben.
- ⌚ zu welchen Zwecken du solche Funktionen in deinen Programmen verwenden kannst.
- ⌚ dass Python mit beliebig großen ganzen Zahlen rechnen kann.
- ⌚ dass Funktionen in Python auch vollwertige Objekte sind.



Manche Funktionen geben Objekte zurück

Am einfachsten erkennt man Funktionen, die ein Objekt zurückgeben, wenn man sie in der PYTHON-SHELL aufruft. Die schreibt nämlich das zurückgegebene Objekt an. Hier einige Beispiele, die schon früher vorgekommen sind:

➤ Starte IDLE(PYTHON GUI) und mach mit!

```
>>> round(1.35247809346)
1
>>> int("24 ")
24
>>> input("Wie heißt du? ")
Wie heißt du? Buffi
'Buffi'
>>> len("abaracadabara")
13
>>> from random import randint
>>> randint(1, 1000)
162
```

Das zurückgegebene Objekt kann ein Zahlenwert, ein String, ein Tupel oder ein beliebiges anderes Objekt sein. Oft nennt man dieses Objekt den von der Funktion zurückgegebenen Wert oder auch *Rückgabewert*.

In Programmen werden Funktionen mit Rückgabewert normalerweise nicht als Anweisungen hingeschrieben, sondern sie sind *Bestandteile* von Anweisungen, die die zurückgegebenen Objekte weiter verwenden.

Dies kann auf unterschiedliche Weise geschehen. Nehmen wir zum Beispiel den Funktionsaufruf `sqrt(2)`. Der Wert, den dieser Aufruf liefert, kann ...

... auf den Bildschirm ausgegeben werden:

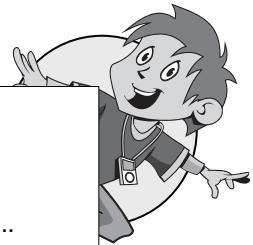
```
>>> from math import sqrt
>>> print(sqrt(2))
1.41421356237
```

... oder einem Namen zugewiesen werden:

```
>>> a = sqrt(2)
>>> a
1.4142135623730951
```

... oder in einem Ausdruck verwendet werden, der als Argument in einen anderen Funktionsaufruf eingesetzt wird:

Wir definieren eine Funktion mit Rückgabewert



```
>>> from turtle import *
>>> forward(20*sqrt(2))
... mit dem bekannten Effekt, dass die Turtle ... (wie viel? Hilf mir, IPI) ...
>>> 20 * sqrt(2)
28.284271247461902
... (ah ja!) ... also ungefähr 28.3 Pixel nach vorne wandert.
>>> tuple(range(3,9))
(3, 4, 5, 6, 7, 8)
```

Im letzten Beispiel gibt die Funktion `range()` ein dynamisches Wertevor-
rat-Objekt zurück, das als Argument für `tuple()` eingesetzt wird.
`tuple()` erzeugt daraus ein Zahlentupel und gibt dieses zurück.

Wir definieren eine Funktion mit Rückgabewert

Nicht erst eben, sondern schon im ersten Kapitel dieses Buches hast du die Quadratwurzel-Funktion verwendet:

```
>>> sqrt(4)
2.0
```

Wir möchten nun die Funktion definieren, die die umgekehrte Rechenope-
ration ausführt: Sie berechnet das Quadrat einer Zahl. Hätte ich das von dir
im vorigen Kapitel verlangt, hättest du vielleicht geantwortet: Kein Prob-
lem!

» Mach mit!

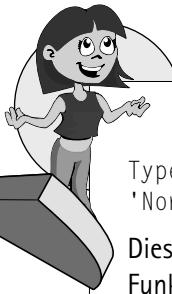
```
>>> def quadrat(x):
    print("Das Quadrat von", x, "ist", x*x)

>>> quadrat(5)
Das Quadrat von 5 ist 25
```

Das ist ja ganz nett, aber zum Rechnen ist es leider völlig unbrauchbar!
Angenommen, wir wollen damit ausrechnen, wie viel das Quadrat von 24
plus dem Quadrat von 7 ist:

```
>>> quadrat(24) + quadrat(7)
Das quadrat von 24 ist 576
Das quadrat von 7 ist 49
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in ?
```

10



```
quadrat(24) + quadrat(7)
```

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

Diese (noch) etwas rätselhafte Fehlermeldung besagt, dass die beiden Funktionsaufrufe `quadrat(24)` und `quadrat(7)` keine Zahlen zurückgegeben haben und der Operator `+` daher nichts addieren konnte.

Die Quadrat-Funktion hat zwar einen *Effekt*; das heißt, sie tut etwas: Sie gibt mit Hilfe der `print()`-Funktion das Quadrat des Arguments auf dem Bildschirm aus. Aber sie gibt keinen Wert zurück.

Um nochmals klarzustellen, was wir wollen: Wir wollen, dass `quadrat` auf die gleiche Weise arbeitet wie `sqrt()`:

```
>>> sqrt(36) + sqrt(169)
19.0
```

Also müssen wir für unsere `quadrat()`-Funktion erreichen, dass die berechneten Werte nach der Berechnung »außerhalb der Funktion« verfügbar sind. Dafür hat Python eine spezielle Anweisung, die `return`-Anweisung:

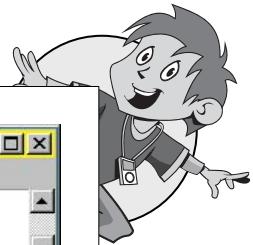
```
>>> def quadrat(x):
    qu = x*x
    return qu

>>> quadrat(24)
576
>>> quadrat(7)
49
>>> quadrat(24)+quadrat(7)
625
```

Wir berechnen im Funktionskörper von `quadrat()` zuerst den gesuchten Wert und speichern ihn in einer lokalen Variablen. Das ist von außen nicht sichtbar. Dann geben wir diesen Wert mit der `return`-Anweisung zurück.

➤ Schreibe nun den Code für die Funktion `quadrat()` in ein Skript `quadrat.py` nach dem folgenden Muster. Überlege gleich anhand der Abbildung, was geschieht, wenn du dieses Programm laufen lässt.

Wir definieren eine Funktion mit Rückgabewert



```
quadrat.py - C:/py4kids/kap10/quadrat.py
File Edit Format Run Options Windows Help
# quadrat.py : Übung für Kapitel 10

def quadrat(x):
    qu = x * x
    return qu

quadrat(24)
quadrat(7)
quadrat(24) + quadrat(7)

Ln: 4 Col: 35
```

Nun haben wir die Funktion `quadrat()` korrekt definiert. Doch der Ablauf des Programms **F5** hat keinerlei Effekt. Der Grund dafür ist, dass hier die Funktion nicht in der richtigen Weise gebraucht wird!

Die Quadrate von 24 und 7 werden zwar berechnet und sogar addiert, aber dann sofort »vergessen«. Das Ergebnis wird weder ausgegeben noch gespeichert. Null Effekt!

Hüte dich vor folgender Verschlimmbesserung – also mach diesmal NICHT mit:

Denn wenn du mit dieser `quadrat()`-Funktion den pythagoräischen Lehrsatz verwenden willst – und in der Computergrafik wirst du auf Dauer kaum ohne ihn auskommen –, erhältst du Folgendes:

Und so etwas kann man wirklich nicht brauchen. Je mehr Rechnungen du durchführst, desto mehr wird der Bildschirm mit Zeug voll geschrieben! Was ist der Grund dafür? Diese Funktion

`quadrat()` hat einen Effekt (sie gibt eine Meldung aus) und sie gibt einen Wert zurück. So etwas sollte man in der Regel vermeiden!

Stattdessen lassen wir in unserem ursprünglichen Skript die Funktion `quadrat()`, wie sie ist, und berichtigen nur die Zeilen, die unterhalb der Definition der Funktion stehen, also die Anwendung der Funktion. Dann sieht unser Skript `quadrat.py` (ohne Kopfkommentar) so aus:

```
quadrat.py - C:/py4kids/kap10/quadrat.py
File Edit Format Run Options Windows Help
# quadrat.py : Übung für Kapitel 10

def quadrat(x):
    qu = x * x
    print("Quadrat von", x, "ist", qu)
    return qu

quadrat(24)
quadrat(7)

Ln: 9 Col: 13
```

```
>>> a = 5
>>> b = 12
>>> quadratsumme = quadrat(a) + quadrat(b)
Quadrat von 5 ist 25
Quadrat von 12 ist 144
>>> quadratsumme
169
>>> |
```

10



```
def quadrat(x):
    qu = x * x
    return qu

print(24, "zum Quadrat ist", quadrat(24))
b = 7
print(b, "zum Quadrat ist", quadrat(b))
cquadrat = quadrat(24) + quadrat(b)
print("{0}² + {1}² = {2}".format(24, b, cquadrat))
```

Es erzeugt mit seinen drei `print`-Anweisungen folgende Ausgabe:

24 zum Quadrat ist 576
7 zum Quadrat ist 49
 $24^2 + 7^2 = 625$

Vielleicht willst du rasch alle Quadrate der Zahlen von 11 bis 20 wissen.
Oder wenigstens, wie man eine Tabelle dieser Quadratzahlen ausgeben kann. Bitte:

➤ Mach mit:

```
>>> for n in range(11,21):
    print(n, "zum Quadrat ist", quadrat(n))

11 zum Quadrat ist 121
12 zum Quadrat ist 144
13 zum Quadrat ist 169
14 zum Quadrat ist 196
15 zum Quadrat ist 225
16 zum Quadrat ist 256
17 zum Quadrat ist 289
18 zum Quadrat ist 324
19 zum Quadrat ist 361
20 zum Quadrat ist 400
```

In unserer Definition der Funktion `quadrat()` wird der Wert, der schließlich zurückgegeben wird, nämlich `x*x`, zunächst einer lokalen Variablen `qu` zugewiesen. Da aber mit diesem Wert keine weiteren Operationen vorgenommen werden, bevor er von der Funktion zurückgegeben wird, ist dies eigentlich nicht notwendig. Die folgende Definition funktioniert genauso:

```
def quadrat(x):
    return x*x
```





Sie ist kompakter und vielleicht findest du sie auch eleganter. Welche Form du bevorzugst, ist bis zu einem gewissen Grad Geschmackssache. Ich rate dir: Nimm die Form, die dir klarer und verständlicher erscheint.

Als Muster für die Definition von Funktionen solltest du dir jedenfalls das Folgende merken. Es berechnet zuerst den Rückgabewert und weist ihn dem lokalen Namen ergebnis zu. Erst als letzte Anweisung enthält der Funktionskörpers die return-Anweisung mit dem Argument ergebnis.

Muster 13: Definition einer Funktion mit Rückgabewert

```
def Funktionsname(parm1, param2, ...): Funktionskopf  
    Anweisung 1 }  
    Anweisung 2 } Funktionskörper  
    return ergebnis → Anweisungen ... berechnen ergebnis.
```

Wenn du einmal mit der Definition von Funktionen mit Rückgabewert vertraut bist, kannst du dieses Muster kreativ und/oder elegant variieren.

Damit du mit den »Funktionen mit Wert« etwas vertrauter wirst, folgen hier einige einfache Beispiele:

Flächeninhalt und Umfang eines Rechtecks

Als erstes Beispiel wollen wir zwei Funktionen definieren, die uns den Umfang und den Flächeninhalt von Rechtecken berechnen und zurückgeben. Dazu überlegen wir: Wovon hängen Inhalt und Umfang einer Rechteckfläche ab? Von der Länge und der Breite des Rechtecks. Diese beiden Werte müssen also den Funktionen als Argumente übergeben werden.

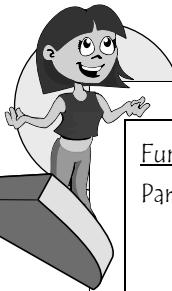
Funktion rechteckumfang():

Parameterliste: laenge, breite

$$\text{umfang} \Leftrightarrow 2 * \text{laenge} + 2 * \text{breite}$$

Rückgabe: umfang

Hierbei drücke ich mit der letzten Zeile aus, dass der Wert von umfang mit einer return-Anweisung zurückgegeben werden soll.

Funktion rechteckflaeche():

Parameterliste: laenge, breite

$$\text{flaeche} \Leftrightarrow \text{laenge} * \text{breite}$$

Rückgabe: flaeche

Beide Funktionen haben dieselbe Gestalt wie die Funktion quadrat() aus dem vorigen Abschnitt. Nur haben sie zwei Eingabeparameter. Beim Codieren können wir daher quadrat() als Vorbild verwenden:

➤ Öffne ein neues Editor-Fenster für das Skript rechteck.py.

➤ Codiere die Funktion rechteckumfang():

```
def rechteckumfang(laenge, breite):
    umfang = 2 * laenge + 2 * breite
    return umfang
```

➤ Codiere nach demselben Schema die Funktion rechteckflaeche().

➤ Nun können wir mit dem IPI kontrollieren, ob die Funktionen richtig funktionieren. Teste sie mit Daten, bei denen du im Kopf mitrechnen kannst:

```
>>> rechteckumfang(1, 1)
4
>>> rechteckflaeche(1, 1)
1
>>> rechteckumfang(3, 7)
20
>>> rechteckflaeche(3, 7)
21
```

Obwohl wir diese Funktionen hier nur zur Übung erstellt haben, wollen wir als eine kleine Anwendung mit ihnen die Frage klären: Welches Rechteck mit Umfang 20 hat die größte Fläche?

Wenn der Umfang 20 sein soll, dann muss Länge plus Breite gleich 10 sein. Oder Breite gleich 10 minus Länge. Sehen wir uns einmal die Rechtecke mit ganzzahligen Längen von 1 bis 9 an:

```
>>> for l in range(1, 10):
    b = 10 - l
    f = rechteckflaeche(l,b)
    print("Länge: {0}, Breite: {1}, Fläche: {2}".format(
        l,b,f))
```

Eine Funktion, die einen String zurückgibt

```
Länge: 1, Breite: 9, Fläche: 9  
Länge: 2, Breite: 8, Fläche: 16  
Länge: 3, Breite: 7, Fläche: 21  
Länge: 4, Breite: 6, Fläche: 24  
Länge: 5, Breite: 5, Fläche: 25  
Länge: 6, Breite: 4, Fläche: 24  
Länge: 7, Breite: 3, Fläche: 21  
Länge: 8, Breite: 2, Fläche: 16  
Länge: 9, Breite: 1, Fläche: 9  
>>>
```



Sieht so aus, als wäre das Rechteck mit der größten Fläche das Quadrat.

Anmerkung: Wenn wir die Suche nach der größten Rechteckfläche verfeinern und in einem kleineren Bereich rund um $a = 5$ weitersuchen wollen, müssen wir beachten, dass `range()` stets nur *ganzzahlige* Werte an die `for`-Schleife liefert. Das könnte etwa so aussehen:

➤ Mach mit!

```
>>> for n in range(45, 55):  
    l = n/10  
    b = 10 - l  
    f = rechteckflaeche(l,b)  
    print("Länge: {0}, Breite: {1}, Fläche: {2}".format(l, b, f))
```

Das Ergebnis dieser Anweisung wird unsere oben formulierte Vermutung bestätigen!

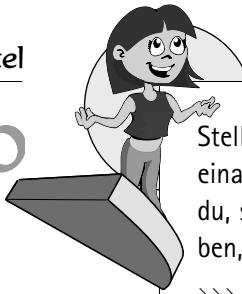
Forschungstipp: Überprüfe, ob auch für andere Rechteckumfänge als 20 gilt, dass das Quadrat die größte Fläche hat.

Forschungstipp: Wie verhält es sich mit den Umfängen von Rechtecken, wenn der Flächeninhalt gegeben ist? Welches davon hat den kleinsten bzw. größten Umfang? Untersuche zunächst Rechtecke mit dem Flächeninhalt 25 und unterschiedlichen Paaren von Länge und Breite.

Eine Funktion, die einen String zurückgibt

Funktionen mit Wert in Python müssen nicht immer nur Zahlen, sondern sie können ganz beliebige Objekte zurückgeben. Weil du noch nicht sehr viele Typen von Objekten kennst – das kommt weiter hinten! –, versuchen wir's einmal mit einer Funktion, die einen String zurückgibt.

10



Stelle dir vor, du hast einen Haufen von Stäbchen und willst diese nebeneinander auflegen. Damit du leichter erkennst, wie viele es sind, beschließt du, sie in Pakete zu je fünf aufzulegen. Falls dann noch welche übrig bleiben, machst du aus ihnen ein kleineres Paket. Ich meine das so:

```
>>> reihe(12)
'IIIIII IIIIII II'
>>> reihe(19)
'IIIIII IIIIII IIIIII IIIII'
>>> reihe(3)
'III'
```

reihe soll also eine Funktion sein, die die Anzahl der Stäbchen als Argument übernimmt und einen String aus 'I's und Leerzeichen zurückgibt.

Um dies zu programmieren, müssen wir zunächst die Anzahl der Pakete und die Anzahl der restlichen Stäbchen ermitteln. Für die Pakete müssen wir die Stäbchenanzahl ganzzahlig durch 5 dividieren. Das kennst du schon:

```
>>> 12 // 5
2
>>> 19 // 5
3
>>> 3 // 5
0
```

Für die restlichen Stäbchen können wir einen dritten Divisionsoperator verwenden, den du vielleicht sogar schon selbst (bei der Arbeit mit Kapitel 1) entdeckt hast: den »Modulo-Operator« %. Der Ausdruck $a \% b$ liefert den Rest bei der Division von a durch b :

```
>>> 12 % 5
2
>>> 19 % 5
4
>>> 3 % 5
3
```

Wenn du dich nun erinnerst, dass du Strings mit * vervielfachen und mit + verketten kannst, bist du schnell am Ziel:

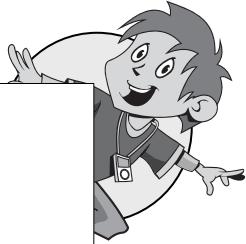
```
>>> s = 19
>>> h = 19 // 5
>>> r = 19 % 5
>>> h
3
>>> "IIIIII " * h
'IIIIII IIIIII IIIIII '
>>> r
```

Der Nachfolger in einer Tischrunde

```
4  
>>> "I" * r  
'IIII'  
>>> "IIIII" * h + "I" * r  
'IIIII IIIII IIIII IIII'
```

Und damit haben wir schon unsere Funktion `reihe()` beisammen:

```
def reihe(staebchen):  
    haufen = staebchen // 5  
    rest = staebchen % 5  
    ergebnis = "IIIII" * haufen + "I" * rest  
    return ergebnis
```



Der Nachfolger in einer Tischrunde

Vielleicht denkst du dir, dass der Modulo-Operator `%` etwas sonderlich und nur selten zu gebrauchen ist. Weit gefehlt, er wird oft verwendet! Sehen wir uns wenigstens noch ein Beispiel an:

Stelle dir vor, vier Personen (oder `n` Personen) sitzen um einen Tisch und spielen ein Spiel, sie kommen der Reihe nach dran. Wir programmieren eine Funktion, die zu jedem Spieler den Nachfolger ermittelt. Dazu nehmen wir an, dass die Spieler (wie schon gewohnt) von 0 bis 3 nummeriert sind. Wir brauchen also eine Funktion, die so funktioniert:

```
>>> n = 4  
>>> nachfolger(0, n)  
1  
>>> nachfolger(1, n)  
2  
>>> nachfolger(2, n)  
3  
>>> nachfolger(3, n)  
0
```

`nachfolger()` übernimmt eine Spielernummer und die Anzahl der Spieler als Argumente und gibt die Nummer des nächsten Spielers zurück. Normalerweise ist diese einfach um 1 größer. Doch wenn durch die Addition von 1 die Anzahl `n` erreicht wird, dann ist der Nachfolger die Nummer 0. Du könntest das sicher leicht mit einer Verzweigung programmieren – versuche es!

10



Doch noch kürzer geht es mit dem %-Operator, weil der Rest immer kleiner ist als der Divisor – ja, der Divisor ist die Zahl, durch die dividiert wird:

```
>>> (2 + 1) % 4
3
>>> (3 + 1) % 4
0
```

Also tut folgende Funktion genau das Gewünschte:

```
def nachfolger(spielder, anzahl):
    return (spieler + 1) % anzahl
```

Du kannst das leicht testen:

```
>>> spielerzahl = 3
>>> for spieler in range(spielerzahl):
    print(spieler, nachfolger(spieler, spielerzahl))

0 1
1 2
2 0
```

Fak!

Apropos Mathe! In diesem Buch werden wir nur noch eine mathematische Funktion behandeln, deren Werte nicht durch eine einzige einfache Rechnung ermittelt werden können. Diese Funktion kommt in (fast) jedem Buch zur Einführung ins Programmieren vor.

Natürlich lernen wir bei dieser Gelegenheit auch wieder einiges über Python.

Es gibt eine recht bekannte Geschichte über den berühmten Mathematiker Carl Friedrich Gauß. Als der neun Jahre alt war, hat sein Klassenlehrer der Klasse einmal die Aufgabe gestellt, die ersten 100 natürlichen Zahlen zu addieren. Die Schüler sollten also $1 + 2 + 3 + \dots + 97 + 98 + 99 + 100$ ausrechnen.

Der Lehrer hatte mehrere Jahrgangsstufen gleichzeitig zu unterrichten und wollte einen Teil der Klasse für einige Zeit beschäftigen. Da damals die Schüler immer das taten, was die Lehrer verlangten, begannen sie sofort mit der Arbeit. Aber bereits nach ganz kurzer Zeit zeigte der kleine C. F. seinem Lehrer, wie damals üblich auf eine Schiebertafel geschrieben, das richtige Ergebnis.



Es stellte sich heraus, dass C. F. das richtige Ergebnis mit einer Formel berechnet hatte, die er sich schnell ausgedacht hatte. Das hatte sein Lehrer nicht erwartet. Er zog daraus den Schluss, dass er selbst ihm in Mathematik nichts weiter beibringen konnte, und ermöglichte ihm bei einem anderen Lehrer eine weitergehende mathematische Ausbildung.

Da das beschriebene Problem mit einer Formel lösbar ist, möchte ich hier kein Programm beschreiben, das die ersten hundert positiven natürlichen Zahlen Stück für Stück addiert.

Vielleicht willst du aber selbst deine Programmierfertigkeiten auf eine Probe stellen? Dann schreibe du doch ein solches Programm. Am besten eine Funktion `summe(n)`, die die Summe der ersten n positiven natürlichen Zahlen ausrechnet und zurückgibt. Deine Funktion `summe()` sollte jedenfalls auch C. F.'s Ergebnis liefern:

```
>>> summe(100)  
5050
```

Wenn dich diese Aufgabe nicht freut oder wenn du sie auf die Schnelle nicht lösen kannst, so ist das auch nicht weiter schlimm. Durchdenke zunächst die Lösung der folgenden Fragestellung, die ganz ähnlich gelagert ist. Vielleicht hast du hinterher Lust, die Sache doch noch (einmal) anzugehen.

Aufgabenstellung: Programmiere eine Funktion, die zu jeder positiven natürlichen Zahl n das Produkt aller positiven natürlichen Zahlen von 1 bis n ausgibt.

Diese Aufgabe ist interessanter, weil es keine Formel gibt, die das Ergebnis sofort liefert.

Zum Beispiel: der Funktionswert für das Argument 12 soll $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot 8 \cdot 9 \cdot 10 \cdot 11 \cdot 12$ sein. Wie viel das ist?

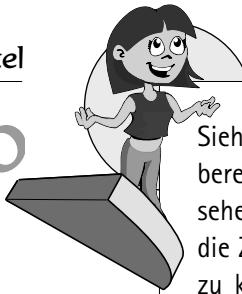
```
>>> 1*2*3*4*5*6*7*8*9*10*11*12  
479001600
```

Diese Funktion hat in der Mathematik einen eigenen Namen, genauer sogar zwei. Sie heißt *Faktorielle* oder auch *Fakultät*. Die Faktorielle der Zahl n wird oft so aufgeschrieben: $n!$

Wir haben also gerade $12!$ (gesprochen: »12-Faktorielle« oder »12-Fakultät«) berechnet: $12! = 479001600$.

Weil diese Funktion oft gebraucht wird, gibt es für sie sogar eine eigene Taste auf deinem Taschenrechner. Üblicherweise ist sie mit $n!$ oder mit $x!$ beschriftet.

10



Sieh nach, was dein Taschenrechner für $0!$, $1!$, $2!$, $3!$, $12!$, $15!$, $20!$, $69!$, $70!$ berechnet. Wenn du einen gewöhnlichen Taschenrechner benutzt, wirst du sehen, dass der da ein paar Probleme hat: Bald ist die Anzeige zu klein, um die Zahlen vollständig darzustellen, und dann ist der ganze Taschenrechner zu klein und gibt nur mehr Fehlermeldungen aus. Aber etwas kannst du doch von ihm lernen: $0! = 1$.

Für negative Zahlen und für Kommazahlen gibt der Taschenrechner auch Fehlermeldungen aus. Für diese Zahlen ist die Faktorielle nicht definiert.

Sehen wir nach, was Python für die Faktorielle-Funktion tun kann!

Die Programmidee muss sein: Nimm die Zahl 1. Wenn $n < 2$ ist, gib sie als Ergebnis zurück. Andernfalls multipliziere sie der Reihe nach mit allen ganzen Zahlen von 2 bis n . Das Ergebnis gib zurück.

Funktion fak():

Parameterliste: n (eine positive ganze Zahl oder 0)

$f \Rightarrow 1$

wenn $n < 2$:

Rückgabe: f

für alle k aus dem Bereich $2, 3, \dots, n$ wiederhole:

$f \Rightarrow f * k$

Rückgabe: f

Beachte: Wenn die Rückgabe in der dritten Zeile des Funktionskörpers ausgeführt wird, wird die Funktion verlassen und die Schleife nicht mehr ausgeführt. Nur für $n \geq 2$ werden die Zeilen darunter ausgeführt.

Hierbei müssen wir festlegen, was mit dem Wertevorrat $2, 3, \dots, n$ gemeint ist. Für große n ist das klar. Aber für $n = 3$ ist damit nur $[2, 3]$ gemeint. Der Teil » \dots « hat also mehr symbolischen Charakter und bedeutet: »Der Bereich aller ganzen Zahlen«, beginnend mit 2 bis zu n . Nach unserer Kenntnis kommt dafür `range(2, n+1)` in Frage. Für $n = 3$ ergibt `range(2, 4)` eben bloß $[2, 3]$.

Für $n = 2$ muss es $[2]$ bedeuten.

Was ergibt `range(2, n+1)` eigentlich, wenn $n = 0$ oder $n = 1$ ist?

⇒ Wir untersuchen das wieder in der PYTHON-SHELL. Mach mit!

```
>>> n = 10
>>> tuple(range(2, n+1))
(2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> n = 3
```



```
>>> tuple(range(2,n+1))
(2, 3)
>>> n = 2
>>> tuple(range(2,n+1))
(2,)
>>> n = 1
>>> tuple(range(2,n+1)) # ist range(2,2)
()
>>> n = 0
>>> tuple(range(2,n+1)) # ist range(2,1)
()
```

`range()` gibt offenbar einen leeren Wertevorrat zurück, wenn das zweite Argument nicht größer ist als das erste. Das heißt aber, dass für diesen Fall in der Schleife keine Anweisung ausgeführt wird und daher `f` unverändert 1 bleibt. Damit können wir uns die Überprüfung, ob `n` kleiner als 2 ist, und die erste `return`-Anweisung sparen und gelangen zu einem verbesserten Entwurf:

Funktion fak():

Parameterliste: `n`, eine positive ganze Zahl oder 0

$$f \Leftrightarrow 1$$

für alle `k` aus dem Bereich `2, .. ,n` wiederhole:

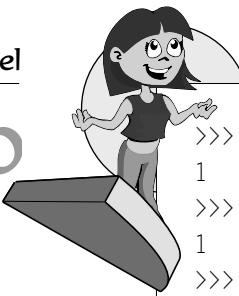
$$f \Leftrightarrow f * k$$

Rückgabe: `f`

Also ergibt sich für die Funktion `fak()` folgender Programmcode:

```
def fak(n):
    f = 1
    for k in range(2,n+1):
        f = f * k
    return f
```

- Schreibe diesen Code in ein Skript `fak01.py`. Speichere das Skript und führe es aus.
- Teste die Funktion `fak()` mit verschiedenen, auch irgendwie besonderen Zahlenwerten. Ich empfehle dir aber, nicht gleich riesige Werte als Argument einzusetzen, sondern die Argumente in kleinen Schritten abzuändern.



10

Langzahlarithmetik

Das Ergebnis von `fak(70)` zeigt uns, dass Python standardmäßig mit einer unbeschränkten Ganzzahlarithmetik arbeitet. Das ist etwas, womit andere gängige Programmiersprachen nicht dienen können.

Beispiel: Bei den meisten Taschenrechnern ist 69 die größte Zahl, für die die Faktorielle berechnet werden kann. Außerdem wird sie nur näherungsweise als Kommazahl in »Exponentialdarstellung« angegeben, und zwar so:

```
>>> float(fak(69))  
1.711224524281413e+098  
>>>
```

Was so viel bedeutet wie $1.711224524281413 \cdot 10^{98}$. In Wahrheit ist aber

```
>>> fak(69)
171122452428141311372468338881272839092270544893520369393648
040923257797541406474240000000000000000
```

Und zwar genau! Du kannst mit Python auch ohne weiteres `fak(10000)` berechnen. Nur musst du dabei eventuell in Kauf nehmen, dass die ermittelte Zahl auf deinem Bildschirm keinen Platz mehr findet. (Sie hat nämlich 35660 Stellen.)

So nebenbei will ich dir zum Abschluss verraten, wie viele verschiedene Tippmöglichkeiten es beim Lotto 6 aus 49 gibt:

```
>>> fak(49)//(fak(6)*fak(49-6))  
13983816
```

Nur für den Fall, dass du geglaubt hast, dass $n!$ im täglichen Leben nicht gebraucht wird. (In Österreich wird übrigens »6 aus 45« gespielt. Wie viele verschiedene Tipps gibt es da?)



Auch Funktionen sind Objekte

Sieh dir mit dem IPI nun einmal an, was es bedeutet, dass Funktionen in Python Objekte sind:

➤ Mach mit!

```
>>> from math import sqrt  
>>> sqrt  
<built-in function sqrt>  
>>> from turtle import *  
>>> penup  
<function penup at 0x00A302C8>
```

Und ebenso erhältst du für selbst definierte Funktionen:

```
>>> def quadrat(x):  
    return x * x  
  
>>> quadrat  
<function quadrat at 0x00A4D880>
```

Wie jedes Objekt können auch Funktionen als Argumente übergeben werden – an andere Funktionen, wenn diese aufgerufen werden. Damit können wir zum Beispiel ein Programm für die Ausgabe von Wertetabellen verschiedener Funktionen schreiben.

Am Anfang dieses Kapitels haben wir eine Wertetabelle für die Funktion `quadrat()` erzeugt. Das ging etwa so:

➤ Mach weiter mit!

```
>>> for n in range(5):  
    print(n, quadrat(n))  
  
0 0  
1 1  
2 4  
3 9  
4 16
```

10



Eine andere Tabelle erhalten wir für die Wurzelfunktion:

```
>>> for n in range(11,15):  
    print(n, sqrt(n))
```

```
11 3.31662479036  
12 3.46410161514  
13 3.60555127546  
14 3.74165738677
```

Hier habe ich den Zahlenbereich der Tabelle geändert und die Funktion, deren Werte ausgedruckt werden sollen. Ich schreibe nun eine weitere Funktion, die diese beiden veränderlichen Bestandteile als Parameter hat und solch eine Tabelle ausdrückt:

```
>>> def tabelle(bereich, funktion):  
    for n in bereich:  
        print(n, funktion(n))
```

Das ist genau unseren zwei Tabellenaufrufen von oben abgekupfert. Und es funktioniert:

```
>>> tabelle(range(5), quadrat)
```

liefert dieselbe Wertetabelle für die `quadrat()`-Funktion, die oben steht. Ebenso:

```
>>> tabelle(range(11,15), sqrt)
```

Hier kommt es besonders darauf an, dass du den Unterschied zwischen dem *Aufruf* einer Funktion und dem Funktionsobjekt genau verstehst. Ich erläutere ihn deshalb nochmals am letzten Beispiel. Dort wird der Funktion `tabelle()` als erstes Argument *der Wert von* `range(11,15)` übergeben,

```
>>> range(11,15)
```

```
range(11, 15)
```

also dieses spezielle Wertevorrat-Objekt. Es wird dem Parameternamen `bereich` zugewiesen.

Beachte, dass die Funktion `sqrt` vor der Ausführung von `tabelle()` *nicht aufgerufen*, sondern als Objekt übergeben wird. Das Funktionsobjekt `sqrt` wird also dem Parameternamen `funktion` zugewiesen. Da es eine Funktion ist, ist dieses Objekt selbst *ausführbar*. Und schließlich wird es auch ausgeführt: nämlich bei jedem Schleifendurchgang einmal, durch den Aufruf `funktion(n)`.

Das Schlaue an dieser Vorgangsweise ist, dass du dich vor oder bei der Programmierung der Funktion `tabelle()` noch nicht entscheiden musst, für welche `funktion` eine Wertetabelle ausgegeben werden soll. Und dass du mit `tabelle()` Wertetabellen für verschiedenste Funktionen und Bereiche erzeugen kannst.



Ganz nebenbei möchte ich hier nochmals auf den wichtigen Unterschied hinweisen zwischen Funktionen, die eine Aktion ausführen, also einen Effekt haben, und solchen, die einen Wert zurückgeben. `tabelle()` ist eine Funktion vom ersten Typ, und ihr Effekt ist die Ausgabe einer Tabelle auf dem Bildschirm. Jede Funktion, die du an `tabelle()` als zweites Argument übergibst, muss aber vom zweiten Typ sein – sonst erhältst du eine Tabelle ohne Funktionswerte.

Laufzeitmessung

Zeitmessung mit dem Computer ist sehr stark von der Hardware und dem Betriebssystem des Computers abhängig.

Python stellt in dem Modul `time` zur Zeitmessung unter anderem die Funktion `clock()` zur Verfügung:

Wenn ich auf meinem Laptop unter Windows den IPI frisch starte, `clock()` aus dem Modul `time` importiere und gleich aufrufe, dann erhalte ich nebenstehendes Ergebnis:

Offenbar gibt die Funktion `clock()` die Zeit in Sekunden an, die seit dem Start der IPI-SHELL vergangen ist.

Wie lange brauchst du, um die erste Zeitangabe von `clock()` zu erhalten?

Zum Glück hängt die Möglichkeit, `clock()` für die Messung der Laufzeit von Schleifen nutzen, davon nicht ab. Du gehst dabei so vor, dass du

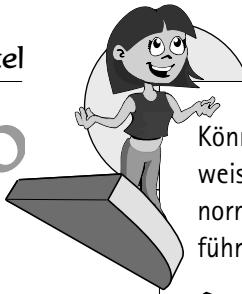
- ◊ eine Zeitmessung vornimmst,
- ◊ dann den Code ausführst, dessen Ausführungszeit dich interessiert,
- ◊ und zuletzt wieder eine Zeitmessung vornimmst.

Die Differenz der beiden Zeitmessungen ist dann gleich der Laufzeit des Codes.

Sehen wir also nach, wie lange dein (bzw. mein) Computer braucht, um eine Million Mal eine `while`-Schleife zu durchlaufen, in der nichts geschieht, außer die Zählvariable um 1 zu erhöhen. Also wie lange er braucht, um bis 1000000 zu zählen.

```
IPI Shell / Turtle Grafik
File Edit Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>> from time import clock
>>> clock()
2.3474466726916408
>>> anfang=clock()
>>> ende=clock()
>>> print(anfang, ende, ende - anfang)
4.8935998849 7.54960197455 2.65600208965
>>> |
```

10



Können wir das interaktiv tun? Dazu müssen wir zuerst drei oder vier Anweisungen eingeben und diese dann auf einmal ausführen lassen. Geht normal nicht, weil im IPI ja jede Anweisung gleich nach der Eingabe ausgeführt wird. Nur mit einem ganz schlauen Trick:

» Mach mit:

```
>>> from time import clock
>>> if True:
    anfang = clock()
    i = 0
    while i < 1000000:
        i = i + 1
    ende = clock()
    print(ende - anfang)
```

0.280498778476

Mein Computer braucht also etwa 0.28 Mikrosekunden für einen Schleifendurchgang der `while`-Schleife. Ist die `for`-Schleife schneller?

Mit der `for`-Schleife kannst du auch ausmessen, wie lange das Abarbeiten von Schleifen braucht, in denen nichts geschieht. Dafür ist natürlich die `while`-Schleife ungeeignet, denn eine `while`-Schleife, in der nichts geschieht, läuft endlos – oder gar nicht! Also nehmen wir dafür eine `for`-Schleife ohne Anweisungen im Schleifenkörper:

*Ohne Schleifenkörper
geht's nicht!*

```
i = i + 1
ende = clock()
print(ende - anfang)

0.280498778476
>>> if True:
    anfang = clock()
```

Der Versuch, dem IPI eine Schleife ohne Schleifenkörper einzugeben und auszuführen, schlägt kläglich fehl! Denn nach Eingabe der ersten Zeile mit dem Doppelpunkt am Ende »weiß« die Python-Syntax-kundige IDLE, dass nun ein eingerückter Block von Anweisungen folgen muss – und wenn er auch nur aus einer Anweisung besteht! Aber keine Anweisung, nein, das geht nicht! Du kannst hier die Eingabe nur abschließen, indem du etwas hinschreibst oder mit `Strg+C` abbrichst.

Laufzeitmessung



In Python gibt es eine spezielle Anweisung, die nichts tut: `pass` – das bedeutet in etwa so viel wie: Geh weiter! Du kannst `pass` überall verwenden, wo die Syntax einen Block verlangt, aber, eventuell vorläufig, nichts gemacht werden soll oder wo du erst später Code einfügen willst.

Für unsere `for`-Schleife sieht das so aus:

```
>>> if True:  
    anfang = clock()  
    for i in range(1000000):  
        pass  
    ende = clock()  
    print(ende - anfang)
```

0.173875450651

Auf meinem Computer läuft also die `for`-Schleife von Python fast 40 Prozent schneller als die `while`-Schleife, oder umgekehrt: Die `while`-Schleife braucht über 60 Prozent mehr Zeit als die `for`-Schleife.

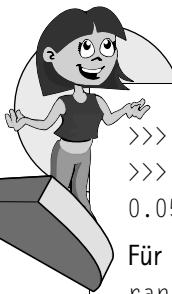
Ich möchte dir hier nun noch zeigen, wie man die Möglichkeit, Funktionen als Argumente zu übergeben, dazu nutzen kann, eine Funktion zur Laufzeitmessung anderer Funktionen zu programmieren. Beschränken wir uns hier auf Funktionen mit einem Argument.

Eine typische Fragestellung ist zum Beispiel: Was geht schneller, die Berechnung einer Quadratwurzel oder die Berechnung einer Zufallszahl?

Beim Aufruf dieser Laufzeit-Funktion möchten wir angeben, welche Funktion gemessen werden soll, welches Argument ihr übergeben werden soll und wie oft sie ausgeführt werden soll. Im Übrigen verfolgen wir dasselbe Schema wie eben:

```
from time import clock  
  
def laufzeit(anzahl, funktion, argument):  
    anfang = clock()  
    for i in range(anzahl):  
        funktion(argument)  
    ende = clock()  
    return ende - anfang
```

- Schreibe ein Skript `laufzeit.py`, das die obige Funktion enthält, und führe es aus.
- Ermittle die Antwort auf unsere obige Frage. Mach mit!



```
>>> from math import sqrt
>>> laufzeit(100000, sqrt, 2)
0.05874517571367477
```

Für die Ermittlung von Zufallszahlen verwenden wir diesmal die Funktion `randrange()`, weil sie mit *einem* Argument verwendet werden kann. `randrange(n)` liefert Zufallszahlen aus dem Bereich `range(n)`.

```
>>> from random import randrange
>>> laufzeit(100000, randrange, 100)
0.19924347926901476
```

Die Berechnung einer Zufallszahl dauert etwa 4 Mal so lange wie die Berechnung einer Quadratwurzel.



Es ist im Allgemeinen nicht sinnvoll, das Laufzeitverhalten von Funktionen zu messen, die `print`-Anweisungen enthalten, besonders nicht, wenn du mit der IDLE arbeitest. `print`-Anweisungen brauchen normalerweise im Vergleich zu Berechnungen innerhalb einer Funktion den Löwenanteil der Zeit.

Was ist Nichts?

Im vorigen Abschnitt haben wir die »leere Anweisung« `pass` kennen gelernt, die »nichts macht«. Mit ihr können wir leicht eine Funktion `test` definieren, die ebenfalls nichts macht, also auch nichts zurückgibt:

```
def test():
    pass
```

Was heißt das? Bevor wir jetzt in eine philosophische Diskussion über die Frage abgleiten, ob Nichts nichts oder doch etwas ist, besinnen wir uns auf die richtige Haltung von Python-Fuzzis – nämlich den IPI zu fragen, wenn uns etwas unklar ist: Kannst du Nichts ausdrucken, oder wirst du doch nichts ausdrucken?

➤ Mach mit:

```
>>> test()
>>> print(test())
None
>>>
```

Wie du nun siehst, hat `test` ein Objekt (`an print()`) zurückgegeben und `print()` hat es ausgedruckt: das Objekt `None`. Übrigens ist `None` in Python

Was ist Nichts?



ein reserviertes Wort. Im Englischen bedeutet das Wörtchen `none` so viel wie »keines« oder auch »nichts«. Du kannst den obigen IPI-Dialog also auch so interpretieren:

»Drucke aus, was die Funktion `test()` zurückgibt!«

»Nichts!«

In der Tat ist dieses `None` ein besonderes Objekt in Python, denn von seinem Typ gibt es nur ein einziges. (Im Gegensatz z.B. zu Objekten vom Typ `int`: Zahlen gibt es viele.) `None` wird von jeder Funktion zurückgegeben, die kein anderes Objekt zurückgibt.

```
>>> from turtle import *
>>> print(forward(50))
None
>>> from mytools import jump
>>> print(jump(50))
None
```

Du hast selbst `jump()` programmiert und nichts dafür getan, dass `None` zurückgegeben wird. Das geschieht automatisch.

Der IPI schreibt alle zurückgegebenen Werte von Ausdrücken – wie du schon weißt, in »roher Form« – immer gleich hin. Deshalb ist er ja so praktisch. Er schreibt aber niemals ein zurückgegebenes `None` hin. Auch das ist eine gute Einrichtung. Es hätte dich bei unseren IPI-Experimenten mit Turtle-Grafik-Anweisungen sicherlich sehr genervt, wenn nach jeder `forward`- oder `left`-Anweisung `None` erschienen wäre.

Im Übrigen ist dieses stets ausgegebene `None` an sich ganz uninteressant. Wenn dir jemand auf alle Fragen immer nur dasselbe antwortet, dann hat diese Antwort null Informationswert.

Dementsprechend kommen auch Anweisungen wie `print(forward(50))` in praktischen Programmen nicht vor. Du solltest nur wissen, dass dieses `None` *stets da* ist.

Jede Python-Funktion gibt ein Objekt zurück.

Wenn im Code der Funktion in einer `return`-Anweisung ein Objekt als Argument angegeben wird, wird dieses zurückgegeben.

In allen anderen Fällen wird `None` zurückgegeben.



Denkst du dir jetzt: »Wozu das Ganze?« Glaubst du vielleicht, dass Nichts zu nichts gut ist? Dann möchte ich dir hier eine nützliche Anwendung von `None` zeigen:

10

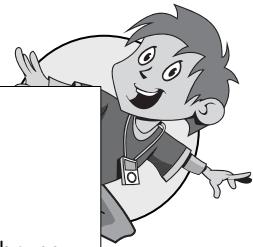


Wenn du weißt, dass du in einem Programm einen bestimmten Namen brauchen wirst, aber noch keine Ahnung hast, auf welches Objekt der später einmal verweisen wird, dann kannst du ihm vorläufig None zuweisen. Damit wird der Name schon in das passende (lokale oder globale) Wörterbuch eingetragen und kann gefahrlos verwendet werden. Seine Verwendung führt nicht mehr zu einem Namen-Fehler.

➤ Mach weiter mit:

```
>>> zeug  
Traceback (most recent call last):  
  File "<pyshell#35>", line 1, in <module>  
    zeug  
NameError: name 'zeug' is not defined  
>>> zeug = None  
>>> zeug                      # None wird vom IPI nicht angezeigt  
>>> print(zeug)  
None  
>>> if zeug == None:  
        print("zeug verweist auf Nichts:", zeug)  
else:  
    print("zeug verweist auf", zeug)  
  
zeug verweist auf Nichts: None  
>>> zeug = 1001  
>>> if zeug == None:  
        print("zeug verweist auf Nichts:", zeug)  
else:  
    print("zeug verweist auf", zeug)  
  
zeug verweist auf 1001
```

Fällt es dir schwer, damit etwas anzufangen? Schon einmal etwas vom Nirvana gehört? Dann denk dir: None ist das Nirvana – des Python-Interpreters. Und im obigen Beispiel verweist eben der Name zeug ins Nirvana. Wenn dir das auch nichts sagt, dann hilft nur abwarten. Es braucht seine Zeit, sich an solche jenseitigen Dinge zu gewöhnen.



Zusammenfassung

- ❖ Die `return`-Anweisung mit Argument bewirkt, dass die Ausführung einer Funktion sofort beendet und das Argument zurückgegeben wird.
- ❖ Damit der Rückgabewert einer Funktion weiter verwendet werden kann, muss er einem Namen zugewiesen oder in einen Ausdruck oder eine Anweisung eingesetzt werden.
- ❖ Python arbeitet mir einer unbeschränkten Langzahlarithmetik für ganze Zahlen.
- ❖ Funktionen sind Objekte und können daher als Argumente an andere Funktionen übergeben werden.
- ❖ Funktionen, die kein anderes Objekt mit einer `return`-Anweisung zurückgeben, geben `None` zurück.

Einige Aufgaben ...

Aufgabe 1: Definiere eine Funktion `hypotenuse()`, die zwei Parameter `a` und `b` für die kürzeren Seiten (die Katheten) eines rechtwinkeligen Dreiecks hat und die Länge der längsten Seite (der Hypotenuse) zurückgibt.

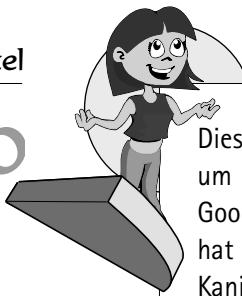
Aufgabe 2: Betrachte die folgende Zahlenfolge:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Jedes Element der Folge ergibt sich, indem man die beiden davor addiert. Das erste Glied der Folge ist 1, das zweite ist auch 1, das dritte ist 2, ..., das 10. Glied ist 55 und so weiter.

Definiere eine Funktion `fib`, die die Zahlen dieser Folge berechnet. Sie liefert folgende Werte:

```
>>> fib(1)
1
>>> fib(2)
1
>>> fib(10)
55
>>> fib(20)
6765
>>> fib(100)
354224848179261915075
```



Diese Folge geht auf den toskanischen Mathematiker Fibonacci zurück, der um 1200 lebte. Er ist so berühmt, dass die Anzahl der Ergebnisse beim Googlen nach »Fibonacci« größer ist als die 32. Fibonacci-Zahl. Fibonacci hat mit den Zahlen dieser Folge die Größe der Nachkommenschaft eines Kaninchenpaares berechnet. Die Zahlen heißen heute Fibonacci-Zahlen.

... und einige Fragen

1. Hier sind zwei Definitionen für Funktionen, die das Volumen eines Würfels berechnen:

```
def wuerfelvolumen1(a):  
    v = a*a*a  
    return v  
def wuerfelvolumen2(a):  
    return a**3
```

Unterscheiden sie sich für den »Anwender«, der sie in einem Programm aufruft?

2. Wie schreibt man in Python eine Anweisung, die nichts tut?
3. Du möchtest 2^{100} berechnen. Du stehst vor der Wahl: Taschenrechner oder Python. Welcher Unterschied besteht zwischen den Ergebnissen, die diese beiden anzeigen werden?
4. Gibt es in Python Funktionen, die keinen Rückgabewert haben?

11



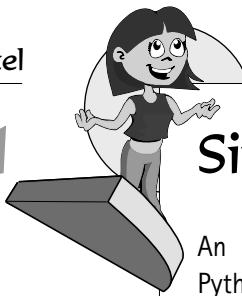
Objekte und Methoden

Seit dem ersten Kapitel ist in diesem Buch immer wieder von Objekten die Rede. Du weißt auch – mindestens von der Einleitung –, dass Python eine objektorientierte Programmiersprache ist.

In diesem Kapitel lernst du mehr über Objekte, nämlich ...

- ◉ am Beispiel der Turtle, wie du Turtle-Objekte erzeugen kannst.
- ◉ dass Turtle-Objekte und Objekte im Allgemeinen Methoden haben.
- ◉ dass du als Programmierer diese Methoden aufrufst, um die Fähigkeiten der Objekte zu nutzen.
- ◉ ... dass es neben Strings und Tupeln noch den Datentyp Liste gibt.
- ◉ ... dass diese drei Datentypen Sequenzen sind.
- ◉ ... dass davon aber nur Listen veränderbar sind.
- ◉ ... über welche Methoden sie verfügen.

11



Sind Turtles Objekte?

An einer früheren Stelle habe ich die Behauptung aufgestellt, dass in Python »fast alles ein Objekt« ist. Wie steht es in dieser Hinsicht mit unserer Turtle?

An sich verhält sie sich wie ein ausgewachsenes Objekt. Sie hat verschiedene Eigenschaften, wie zum Beispiel die Strichdicke und die Farbe, mit der sie zeichnet. Diese Eigenschaften kannst du mit passenden Anweisungen – Funktionsaufrufen wie `pensize()`, `pencolor()` – einstellen. Sie hat auch einige Fähigkeiten. Sie kann vorwärts und rückwärts laufen, dabei Linien zeichnen oder auch nicht, und sie kann sich nach rechts oder nach links drehen. Du bringst sie dazu, das zu tun, indem du Funktionen wie `forward()`, `left()` und so weiter aufrufst.

Und doch ist diese Turtle für uns ein seltsam anonymes Ding: Sie hat keinen Namen. Und wir können sie nicht direkt erzeugen, sondern sie ist mit der Anweisung `from turtle import *` einfach da. Mit der ersten eigentlichen Turtle-Grafik-Anweisung werden sowohl ihr Spielfeld, das Turtle-Grafik-Fenster, wie auch sie selbst sichtbar. (Wenn diese Anweisung nicht gerade `hideturtle()` ist, die mutwillig die Turtle unsichtbar macht.)

Wenn ich bis jetzt von Objekten geschrieben habe, waren das Zahlen, Strings, Listen, Funktionen und so weiter. Das sind Objekte, die zu Datentypen gehören, die in Python fix eingebaut sind.

Objekte vom Typ unserer Turtle sind aber in Python nicht von vornherein eingebaut. Objekte vom Typ Turtle werden nach einem Bauplan erzeugt, den ein Programmierer – in dem Fall war das ich – in ein Python-Modul geschrieben hat, nämlich in das Modul `turtle.py`.

Einen solchen Bauplan, nach dem Objekte erzeugt werden können, nennt man eine *Klasse* oder manchmal genauer eine *Klassendefinition*.

Vergiss nun einmal die namenlose Turtle, die uns bisher so treue Dienste geleistet hat, und schau dir an, wie man eine Klasse dazu benutzen kann, eins, zwei, drei ... viele Objekte zu erzeugen.

Die Klasse, mit der Turtles erzeugt werden können, heißt `Turtle`. Wir wollen also jetzt ein paar `Turtle`-Objekte erzeugen und sie dazu bringen zu zeichnen. Wenn wir mehrere Objekte unterscheiden wollen, müssen wir ihnen jeweils eigene Namen geben, mit denen wir sie ansprechen.

Um `Turtle`-Objekte erzeugen zu können, musst du nur den Namen der Klasse, also `Turtle`, aus dem Modul `turtle` importieren.

Sind Turtles Objekte?

➤ Starte den IPI neu und mach mit!

```
>>> from turtle import Turtle  
>>> alex = Turtle()  
>>>
```

Was haben wir jetzt? Ein Turtle-Objekt namens alex. Du siehst es im Turtle-Grafik-Fenster. Möchtest du, dass alex 50 Einheiten vorwärts läuft?

```
>>> forward(50)  
Traceback (most recent call last):  
  File "<pyshell#2>", line 1, in <module>  
    forward(50)  
NameError: name 'forward' is not defined  
>>>
```

Schluck! Warum ist jetzt auf einmal der Name forward nicht definiert? Ganz einfach: Wir haben ihn nicht importiert. Nur Turtle haben wir importiert! Das hat aber auch seinen tieferen Sinn. Sagen wir mal:

```
>>> bert = Turtle()
```

Dann haben wir ein zweites Turtle-Objekt: Es heißt bert. Leider sieht man es (noch) nicht. Doch frage ich dich, was sollte forward(50) jetzt bedeuten?

Dass alex 50 Einheiten vorwärts läuft? Oder doch bert? Wenn wir eine Botschaft haben, forward(50), aber mehrere mögliche Adressaten, dann müssen wir wohl auch angeben, an wen die Botschaft gerichtet ist. Also? Schicken wir die Botschaft an alex:

```
>>> alex.forward(50)
```

Und schon hat er's getan! Und jetzt können wir auch bert sehen – er und alex haben vorher übereinander gesessen. Wird bert auch so folgsam sein?

```
>>> bert.left(90)
```

Ja, bert reagiert auch. Machen wir ihn rot, damit wir ihn gleich erkennen können:

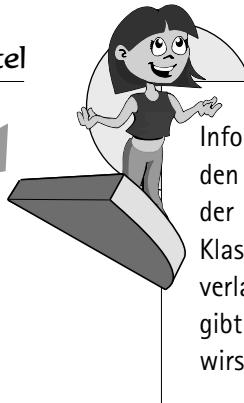
```
>>> bert.color("red")  
>>> bert.forward(50)
```

Machen wir uns rasch noch eine weitere Schildkröte. Das geht genauso wie oben: Man schreibt den Klassennamen hin, Turtle, aber so, als wenn es ein Funktionsaufruf wäre: Turtle().

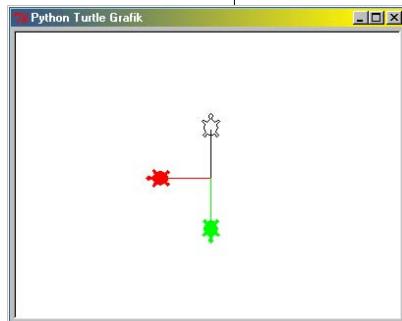
```
>>> carl = Turtle()
```



11



Informatiker sagen: Wir konstruieren ein Turtle-Objekt und geben ihm den Namen carl. Zu diesem Zweck wird eine Spezialfunktion aufgerufen, der *Konstruktor* der Klasse Turtle. Er hat genau denselben Namen wie die Klasse, ist aber ein Funktionsaufruf. Der Konstruktor der Klasse Turtle verlangt keine Argumente. Daher bleibt das Klammerpaar dahinter leer. Es gibt aber auch Klassen, deren Konstruktoren Argumente verlangen. Du wirst sie bald kennen lernen.



carl könnte vielleicht grün sein:

```
>>> carl.color("green")
>>> carl.left(180)
>>> carl.forward(50)
>>>
```

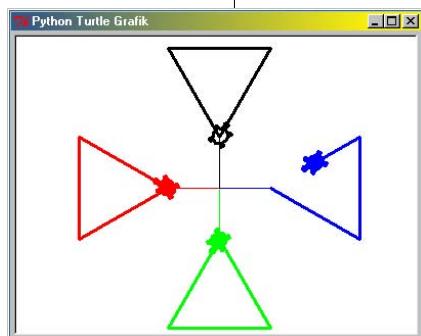
➤ Erzeuge nun eine vierte Kröte namens dinu, die blau ist, und schicke sie um 50 Einheiten nach rechts.

Bevor ich dir wieder einige Erklärungen gebe, lass uns noch die vier zum Tanzen bringen.

Zunächst fassen wir sie zu einem Tupel zusammen:

```
>>> kroeten = (alex, bert, carl, dinu)
```

Und dann die kleine Showeinlage:



```
>>> for krot in kroeten:
    krot.pensize(3)
    krot.right(30)

>>> for i in range(3):
    for krot in kroeten:
        krot.forward(100)
        krot.left(120)
```

alex, bert, carl und dinu zeichnen Dreiecke.

Ist dir *krot* kein Begriff? Das Wichtigste an *krot* ist, dass man es mit einem langen geschlossenen o spricht. Und weil es ein österreichischer Dialektausdruck ist, wo man keine harten Konsonanten kennt, ist es am ehesten so auszusprechen: »groot« – es bedeutet nichts anderes als Kröte – und in unserem Zusammenhang ist es eine Kurzform für Schildkröte ... na ja, du kannst ja einen anderen Namen verwenden. Ein echtes Grundproblem beim

Sind Turtles Objekte?

Programmieren tritt hier zutage: Wo nehme ich schnell einen passenden Namen für ein Objekt her?

Zurück zu den Kröten: Die Showeinlage entsteht durch eine geschachtelte Schleife. Drei Mal: (jede Krot geht vor und dreht sich).

Wie du siehst, ist es ganz leicht, mit Turtle-Objekten zu arbeiten. Sie kennen nämlich alle Funktionen, die wir von früher aus dem Turtle-Modul kennen – doch mit dem Unterschied, dass diese Funktionen jetzt nicht für sich arbeiten, sondern an Objekte – die Turtles – gebunden sind.

Solche an Objekte gebundene Funktionen werden im OO-Sprech (in der Sprechweise der objektorientierten Programmierung) als *Methoden* bezeichnet.

Die Anweisung `alex.forward(50)` ist also der Aufruf der Methode `forward()` für das Objekt `alex`. Was der Effekt eines Methodaufrufs ist, hängt also nicht nur davon ab, welche Methode aufgerufen wurde, sondern auch davon, *für welches Objekt* diese Methode aufgerufen wurde. Ich kennzeichne im Text Methoden ebenso mit einem leeren Klammernpaar wie Funktionen.

Du kennst diese Punkt-Schreibweise schon von Aufrufen der Methode `format()` für Strings: `"Hallo {0}!".format(name)`.

In diesen Fällen stand links vom Punkt kein Objektname, sondern ein buchstäblich hingeschriebenes String-Objekt, zu erkennen an dem " " - Paar.

Du wirst ab jetzt in deinen Python-Programmen Methodaufrufe so häufig brauchen, dass es sich lohnt, sich die Syntax des Methodaufrufs als ein kleines Mini-Muster zu merken:

(Mini-)Muster 14: Methodenaufruf

`objektname.methodename(arg1, arg2, ...)`

oder:

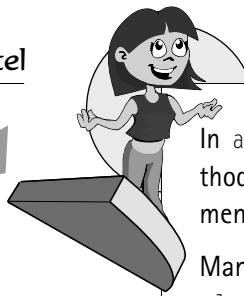
`objekt.methodename(arg1, arg2, ...)`

Bei einem Methodenaufruf sind fixe Elemente nur der Punkt zwischen Objekt und Methodennamen und das Paar runder Klammern, das die Argumente (falls vorhanden) einschließt.

Diese Schreibweise wird auch als *Punkt-Notation* bezeichnet.



11



In `alex.forward(50)` ist also `alex` der Objektname, `forward` der Methodenname und `50` das an die Methode `forward()` übergebene Argument.

Manchmal wird dieser Sachverhalt auch so beschrieben: An das Objekt `alex` wird die Botschaft `forward(50)` gesendet.

Du kannst dir die Methoden am besten als *Fähigkeiten von Objekten* vorstellen, bestimmte Aktionen auszuführen oder bestimmte Ergebnisse zu ermitteln und zurückzugeben. Die Methode `forward()` verleiht unseren Schildkröten die Fähigkeit, vorwärts zu gehen, die Methode `pencolor()` verleiht ihnen die Fähigkeit, ihren Farbstift zu wechseln und so weiter.

Auch `position()`, das du aus Kapitel 9 kennst ist eine Turtle-Methode, und zwar eine, die ein Ergebnis zurückgibt:

```
>>> alex.position()  
(-0.00, 50.00)  
>>> bert.position()  
(-50.00, -0.00)
```

Sie gibt ein Tupel mit zwei Elementen zurück, die die aktuellen Koordinaten der jeweiligen Turtle sind. Beachte bitte, dass die Koordinaten auf zwei Stellen hinter dem Komma gerundet angeschrieben werden. Intern wird aber mit voller Genauigkeit gerechnet! Du kannst dir das ansehen, indem du das Koordinatentupel entpackst:

```
>>> u, v = alex.position()  
>>> u  
-4.263256414560601e-14  
>>> v  
50.0
```

Bei manchen Werten entstehen durch die fortlaufenden Berechnungen des Weges der Turtle (sehr) kleine Rundungsfehler.

Methoden müssen in irgendeiner Form in der Klassendefinition der Klasse festgeschrieben werden, deren Objekte diese Methoden kennen sollen. Wie das geschehen kann, werden wir in Kapitel 14 behandeln, wenn wir selbst Klassen definieren.

Hier rufe ich dir nochmals die Grundlage für das Arbeiten mit Objekten, die nicht in Python eingebaut sind, in Erinnerung: Um Objekte zu erzeugen, muss ein Bauplan für die Objekte vorliegen, der die Fähigkeiten (Methoden) und Eigenschaften der Objekte festlegt. Dieser Bauplan ist die Klassendefinition.

Objekte werden erzeugt, indem eine Spezialfunktion, der Konstruktor der Klasse, aufgerufen wird. Für Objekte wird merkwürdigerweise auch oft die

Sind Turtles Objekte?

Bezeichnung *Instanz* verwendet. Also: alex ist eine Instanz der Klasse Turtle.

(Ich nehme an, dass dieses nicht dein letztes Buch über Programmieren sein wird. So viel muss ich dir wohl hier über Begriffe mitteilen, damit du nicht ganz verwirrt bist, wenn du diese später mal in anderen Büchern vorfindest.)

Besser wäre vielleicht die Sprechweise, alex ist ein Exemplar der Klasse Turtle oder Ähnliches. Instanz leitet sich vermutlich aus einer schlechten Übersetzung des englischen *instance* her. Das heißt auf Deutsch so viel wie Beispiel, Beispieldfall.

Du solltest dir jedenfalls den Unterschied zwischen Objekten und Klassen ganz klarmachen:

Wohnen in deiner Umgebung Hunde? Alle Hunde bilden eine Klasse. Sie haben ja auch viele Fähigkeiten und Eigenschaften gemeinsam. Diese legen fest, was ein Hund ist. Vielleicht hat ein Freund von dir einen Hund namens Bello, eine Freundin einen Waldi zu Hause. Bello und Waldi sind Objekte der Klasse Hund. Ein OOP-Freak würde vielleicht sagen: Bello ist eine Instanz der Klasse Hund. Und im normalen Leben heißt es, Waldi ist ein konkretes Beispiel für einen Hund.

Na ja, so verhält es sich auch mit alex, bert, carl, dinu und der Klasse Turtle.

Vereinbarung über Namen: Damit man im Programmcode sofort zwischen Objekten und Klassen unterscheiden kann, beginnen Namen von Objekten stets mit einem Kleinbuchstaben, Namen von Klassen aber stets mit einem Großbuchstaben.

Es ist sehr wichtig, dass du dich an Vereinbarungen dieser Art hältst. Damit kann der Leser der Anweisung

`dinu = Turtle()`

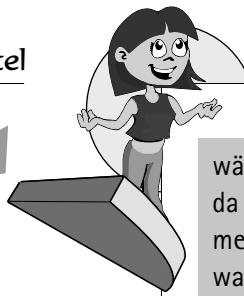
sofort erkennen, dass dinu der Name eines Objekts ist, Turtle der Name einer Klasse, Turtle() der Aufruf eines Konstruktors und daher dinu auf ein (neues) Turtle-Objekt verweist. Eine Menge Information steckt da drin!

Ein Leser dagegen, der in deinem Code lesen würde:

`Fritz = zeichner()`



11



wäre ganz durcheinander und würde sich denken: Was für ein Chaos war da am Werk! Man kann sich nicht einmal darauf verlassen, dass ein Name, der mit einem Großbuchstaben beginnt, eine Klasse bezeichnet. Und was soll nun `zeichner()` sein? Ein normaler Funktionsaufruf oder etwa doch der Aufruf eines Konstruktors einer Klasse, deren Name mit einem Kleinbuchstaben beginnt?

Python hat eine sehr umfangreiche Klassenbibliothek, in der ganz unterschiedliche Arten von Objekten schon vordefiniert sind: nicht nur Turtles, sondern auch Knöpfe, Schieber und Fenster für Benutzeroberflächen oder ganze Webserver (ja, echt!), um die Welt mit Informationen zu versorgen.

Python gestattet dir aber auch, eigene Klassen zu programmieren. Mit denen kannst du dann sozusagen selbst entworfene Objekte herstellen.

In diesem Kapitel werden wir noch mit einigen unterschiedlichen Typen von Objekten arbeiten und am Ende werde ich die grundlegenden Tatsachen über Objekte und Methoden nochmals zusammenfassen.

Zunächst aber machen wir aus unserem eben erzeugten Vier-Turtle-Muster noch ein kleines Programm:

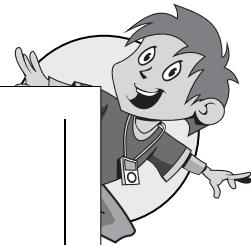
Vier dynamische Spiralen

Wenn wir die Anweisungen des vorigen Abschnittes in ein Skript schreiben, erhalten wir folgendes Skript `vierturtles.py`:

```
from turtle import *

alex = Turtle()
bert = Turtle()
alex.forward(50)
bert.left(90)
bert.color("red")
bert.forward(50)
carl = Turtle()
carl.color("green")
carl.left(180)
carl.forward(50)
dinu = Turtle()
dinu.left(270)
```

Vier dynamische Spiralen



```
dinu.color("blue")
dinu.forward(50)
kroeten = (alex, bert, carl, dinu)

for krot in kroeten:
    krot.pensize(3)
    krot.right(30)

for i in range(3):
    for krot in kroeten:
        krot.forward(100)
        krot.left(120)
```

Daraus soll ein Programm `dynaspiralen.py` werden, bei dem die vier Turtles »gleichzeitig« vier nach innen verlaufende Spiralen zeichnen. Außerdem sollen sie auf dem Weg zum Startpunkt der Spiralen nicht zeichnen.

➤ Speichere das Skript `vierturtles.py` als `dynaspiralen_arbeit.py` ab.

Schlüssel für die paar Änderungen, die dazu nötig sind, ist die Verwendung von `range()` mit drei Parametern. (Denke an die Fragen am Ende von Kapitel 7.)

➤ Mach mit, um dir in Erinnerung zu rufen, wie das dritte Argument von `range()`, die Schrittweite, arbeitet:

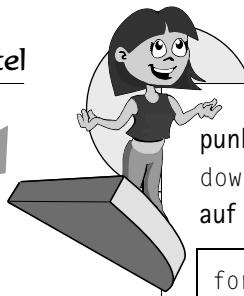
```
>>> tuple(range(3,30,5))
(3, 8, 13, 18, 23, 28)
>>> tuple(range(30, 3, -5))
(30, 25, 20, 15, 10, 5)
>>> tuple(range(100, 65, -5))
(100, 95, 90, 85, 80, 75, 70)
```

Das passt doch! Wir lassen die Turtle für jede dieser Zahlen eine `forward()`-Anweisung ausführen. Die Striche werden dann immer kürzer. Es erfordert folgende Änderung in der letzten Schleife:

```
for laenge in range(100, 0, -5):
    for krot in kroeten:
        krot.forward(laenge)
        krot.left(120)
```

Und was den Code davor betrifft: da färben und drehen wir – wie gehabt – jede Turtle individuell, aber die vier `forward(50)`-Schritte zu den Start-

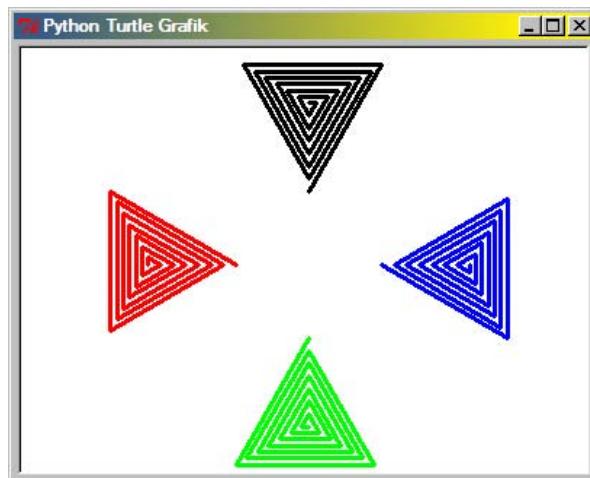
11



punkten der Spiralen verschieben wir (mit passendem penup() / pendown()) in die erste Schleife. Wenn wir auch noch die Geschwindigkeit auf maximal stellen und die Turtle verstecken wollen, ergibt sich für diese:

```
for krot in kroeten:
    krot.hideturtle()
    krot.speed(0)
    krot.penup()
    krot.forward(50)      # ähnlich wie jump()
    krot.pendown()
    krot.pensize(3)
    krot.right(30)
```

Wenn du diese Änderungen vornimmst und das Programm laufen lässt, sollte sich folgendes Bild ergeben:



Gleichzeitig gezeichnet? Ganz gleichzeitig geht es wohl nicht. Wir konnten es aber immerhin so einrichten, dass abwechselnd immer eine andere Turtle eine Dreiecksseite zeichnet.

Noch etwas »gleichzeitiger« geht es, wenn wir die Ablaufverfolgung ausschalten. Bisher haben wir dafür die tracer() Funktion verwendet. Dieser entspricht aber keine Turtle-Methode. Denn die Ablaufverfolgung kann man nicht für einzelne Turtles, sondern nur für das ganze Turtle-Grafik-Fenster aus- und einschalten.

Das wollen wir nun auch »objektorientiert« lösen. Dafür gibt es im turtle-Modul die Klasse Screen. Diese importieren wir auch und erzeugen gleich ein Screen-Objekt. Dieses hat eine Methode tracer().

Sequenzen



Ändere den Anfang des Programms wie folgt ab:

```
from turtle import Turtle, Screen  
  
screen = Screen()  
alex = Turtle()
```

In der Schleife, die die Spiralen zeichnet, können wir nun die Methode tracer() für das Objekt screen aufrufen:

```
for laenge in range(100,0,-5):  
    screen.tracer(False)  
    for krot in kroeten:  
        krot.forward(laenge)  
        krot.left(120)  
    screen.tracer(True)
```

- Füge diese Änderungen in dynaspiralen_arbeit.py ein und teste dein Programm.
- Wenn es funktioniert, speichere eine Kopie als dynaspiralen01.py. Mit dynaspiralen_arbeit.py werden wir in Kapitel 14 weiterarbeiten.

Wolltest du dieses Programm mit nur einer einzigen anonymen Turtle schreiben? (Bitte, tu es nicht!)

Sequenzen

Sequenzen sind geordnete Sammlungen von Bestandteilen. Okay, Sequenz ist ein Fremdwort. In meinem Duden steht: Reihe, Folge, Reihenfolge. Schon in Kapitel 7 hast du eine Art von Sequenzen kennen gelernt: Tupel. Hier möchte ich dir Sequenzen allgemeiner und genauer erklären. Zu diesem Zweck werden wir die IDLE etwas ausführlicher bemühen, bevor wir die neuen Erkenntnisse in praktischen Programmen verwenden.

Es gibt in Python verschiedene Typen eingebauter Objekte, die Sequenzen sind. Wir werden hier drei davon behandeln: Strings, Tupel und Listen.

11



Sequenz-Objekte haben sehr viele interessante Eigenschaften, Operationen und Methoden. Gewöhne dir besonders im Hinblick auf Sequenzen an, Fragen über sie und Möglichkeiten, die sie bieten, mit der Python-Shell zu untersuchen!

Auf diese Weise wirst du bald ein Gefühl dafür bekommen, was man mit ihnen alles machen kann und wie man sie am besten verwendet. In den folgenden Abschnitten haben solche interaktiven Übungen daher auch einen besonders hohen Stellenwert!

Erzeugen von Sequenzen durch Anschreiben der Elemente:

1. *Listen*: Listen sind Objekte, die aus Elementen bestehen. Die Elemente von Listen können beliebige Objekte sein, sogar wieder Listen.

Du kannst Listen ähnlich wie Tupel bilden: indem du ihre Elemente in der gewünschten Reihenfolge, durch Kommas getrennt, anschreibst, aber zwischen eckige Klammern eingeschlossen. Das geht gut, wenn es nicht zu viele sind und wenn sie von Anfang an alle bekannt sind. Zur Erinnerung ein paar Beispiele:

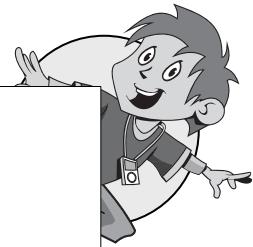
➤ Starte IDLE neu und mach mit oder lade sequenzen.py und führe es aus:

```
>>> listeA = [3, 3.5, 4, 4.5]
>>> listeB = ["red", "blue", "yellow"]
>>> listeC = [1, [2, 3], [4, [5, 6]]]
>>> listeD = []
```

2. *Tupel*: Bei der Erzeugung durch Anschreiben unterscheiden sich Tupel von Listen nur dadurch, dass ihre Elemente von runden Klammern umgeben sind:

```
>>> tupelA = (1,3)
>>> tupelB = (1, (2,3), "huch", [])
>>> tupelC = ("fritz",)
>>> tupelD = ()
```

Hier siehst du eine wichtige Ausnahme: Einelementige Tupel brauchen nach dem einzigen Element ein Komma, damit der Python-Interpreter sie von Klammerausdrücken wie (3+4) unterscheiden kann:



```
>>> (3 + 4)  
7  
>>> (3 + 4,)  
(7,)
```

3. *Strings*: Sie werden angeschrieben, indem Zeichen zwischen ein Paar von Anführungszeichen gesetzt werden – die Elemente von Strings können also nur Textzeichen sein:

```
>>> stringA = "X"  
>>> stringB = 'bad'  
>>> stringC = """Eine Zeile  
und noch eine Zeile  
und noch eine Zeile."""  
>>> stringD = ""
```

Damit haben wir genügend Material beisammen, dass wir damit zunächst die gemeinsamen Eigenschaften von Sequenzen bestimmen können.

Die Länge von Sequenzen

Die (in Python) »eingebaute« Funktion `len()` kann eine Sequenz als Argument übernehmen und gibt ihre Länge zurück. Zum Beispiel:

```
>>> len(listeA)  
4  
>>> len(listeD)  
0  
>>> len(tupelC)  
1  
>>> len(stringD)  
0
```

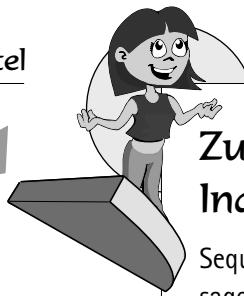
Was heißt das? `listeA` hat vier Elemente. `listeD` hat 0 Elemente. Man nennt sie die *leere Liste*. `tupelC` hat ein Element. `stringD` hat 0 Zeichen. Man nennt ihn den *Leerstring*. Wie viele Elemente hat `listeC`?

```
>>> listeC  
[1, [2, 3], [4, [5, 6]]]  
>>> len(listeC)  
3
```

Überrascht dich das? Überlege auch noch für die anderen Objekte, wie groß ihre Länge ist, und stelle sie dann mit der Funktion `len()` fest.

Der IPI behauptet, dass `listeC` drei Elemente hat. Welche sind das?

11



Zugriff auf die Elemente von Sequenzen mit Indizes

Sequenzen sind *geordnete Folgen* von Elementen. Die Elemente sind sozusagen nummeriert. Man kann ein Element einer Sequenz ermitteln, indem man nach dem Namen, der auf die Sequenz verweist, die Nummer des Elements in eckigen Klammern angibt. Sehen wir nach, welches das erste Element von `listeA` ist.

```
>>> listeA  
[3, 3.5, 4, 4.5]  
>>> listeA[1]  
3.5
```

Das ist merkwürdig. Wie groß ist dann das vierte Element?

```
>>> listeA[4]  
Traceback (most recent call last):  
  File "<pyshell#30>", line 1, in <module>  
    listeA[4]  
IndexError: list index out of range
```

Wir finden einen Index-Fehler. Was heißt das? Um das zu verstehen, musst du wissen, was ein Index ist. Der Index eines Elements einer Sequenz ist die Nummer dieses Elements. (*Index* kommt aus dem Lateinischen und heißt so viel wie Zeiger. Die Mehrzahl des Wortes *Index* lautet daher *Indizes*, gemäß dem Duden darf man aber auch einfach *Indexe* sagen.) Wir haben soeben das vierte Element von `listeA`, also das Element mit dem Index 4 gesucht. Der IPI sagt uns, dass der Index 4 außerhalb des erlaubten Bereichs liegt. Es ist einfach so, dass Zählen in der Informatik fast immer bei 0 beginnt. Erinnere dich an:

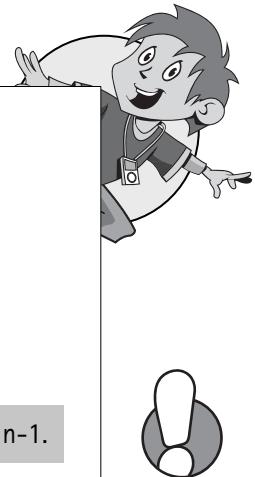
```
>>> tuple(range(4))  
(0, 1, 2, 3)
```

Man kann aus den Werten des Vorrats, den `range(4)` liefert, ebenso leicht eine Liste machen:

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

Bei dieser Liste ist jedes Element gerade so groß wie sein Index. Der höchste Index von `listeA` ist daher 3 – und der kleinste 0. Stell es dir so vor:

Sequenzen



Der Index ist der Abstand des Elements vom Anfang der Sequenz

```
>>> listeA  
[3, 3.5, 4, 4.5]  
>>> listeA[3]  
4.5  
>>> listeA[0]  
3
```

Die Indizes der Elemente einer Sequenz mit Länge n laufen von 0 bis n-1.

Jetzt können wir auch feststellen, wie es sich mit den Elementen von listeC verhält. Ihre Länge ist 3, daher muss sie Elemente mit den Indizes 0, 1, 2 enthalten:

```
>>> listeC[0]  
1  
>>> listeC[1]  
[2, 3]  
>>> listeC[2]  
[4, [5, 6]]  
>>>
```

Du siehst, die Elemente mit den Indizes 1 und 2 sind selbst Listen.

- ⇒ Untersuche, welche Elemente tupelB hat.
- ⇒ Überlege, wie das folgende Ergebnis zustande kommt:

```
>>> len(stringC)  
51
```

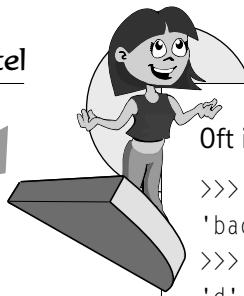
Wenn man die Zeichen in den drei Zeilen abzählt, sieht es doch so aus, als hätte stringC $10 + 19 + 20 = 49$ Zeichen. Du kannst dir aber stringC näher anschauen:

```
>>> stringC  
'Eine Zeile\nund noch eine Zeile\nund noch eine Zeile.'
```

Nanu! Jetzt sieht es wieder wie 53 Zeichen aus. Ich verrate dir des Rätsels Lösung: \n zählt als ein Zeichen, und zwar als ein Sonderzeichen – das Zeilenvorschubzeichen. Es bedeutet, dass bei der Ausgabe durch die print()-Funktion danach eine neue Zeile beginnt. Sonderzeichen beginnen immer mit einem *backslash*, einem rückwärts geneigten Strich. Er besagt, dass das Zeichen dahinter – in unserem Fall das n für *new line* – kein normales Zeichen, also kein normales n ist, sondern etwas Besonderes. Du kannst dieses Zeichen auch selbst in Strings einbauen. Versuche mal:

```
>>> print("ha\nha\n\nna und?")
```

11



Oft ist es sehr nützlich, dass Python auch negative Indizes versteht:

```
>>> stringB  
'bad'  
>>> stringB[-1]  
'd'  
>>> stringB[-2]  
'a'  
>>> listeC  
[1, [2, 3], [4, [5, 6]]]  
>>> listeC[-3]  
1
```

Negative Indizes bedeuten also, dass die Elemente einer Sequenz vom letzten, das den Index -1 hat, weg nach vorn gezählt werden. Denke so: Wenn du bei der hinteren schließenden Klammer der Liste stehst – wie viele Schritte musst du zurückgehen, um zum Element zu kommen?

Erzeugen von Sequenzen mit den Operatoren + und *

Neben dem direkten Anschreiben gibt es noch andere Arten, Sequenzen zu erzeugen. Eine Art, die für alle Sequenzen funktioniert, ist die Verkettung, eine andere die Vervielfachung von Sequenzen gleichen Typs:

```
>>> listeA  
[3, 3.5, 4, 4.5]  
>>> listeB  
['red', 'blue', 'yellow']  
>>> listeA + listeB  
[3, 3.5, 4, 4.5, 'red', 'blue', 'yellow']  
>>> listeA * 3  
[3, 3.5, 4, 4.5, 3, 3.5, 4, 4.5, 3, 3.5, 4, 4.5]  
>>> stringA  
'X'  
>>> stringB  
'bad'  
>>> (stringA + stringB) * 5  
'XbadXbadXbadXbadXbad'  
>>> 3*tupelC + tupelA  
('fritz', 'fritz', 'fritz', 1, 3)
```

Sequenzen

Kommen wir nochmals auf das Beispiel weiter oben zurück:

```
>>> (3 + 4) * 7  
49  
>>> (3 + 4,) * 7  
(7, 7, 7, 7, 7, 7, 7)
```

$(3 + 4)$ ist eben im Gegensatz zu $(3 + 4,)$ kein Tupel, sondern eine Zahl!

Clara Pythias Python-Special: Sequenzen, scheibchenweise:

Oft braucht man von Sequenzen nicht einzelne Elemente, sondern Teilbereiche. Python hat ein einfaches Verfahren, »Scheiben« aus Sequenzen herauszuschneiden. (Im Englischen heißen diese Dinger *slices*. Im Deutschen spricht man manchmal auch von Abschnitten.) Scheiben einer Sequenz sind immer *neue* Objekte vom selben Typ. Schauen wir uns das an einem Beispiel an:

```
>>> neue_liste = listeA + listeB  
>>> neue_liste  
[3, 3.5, 4, 4.5, 'red', 'blue', 'yellow']  
>>> neue_liste[2]  
4  
>>> neue_liste[5]  
'blue'
```

Bis hierher lief alles wie erwartet. Nun machen wir »Scheiben«:

```
>>> neue_liste[2:5]  
[4, 4.5, 'red']
```

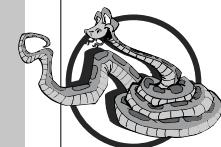
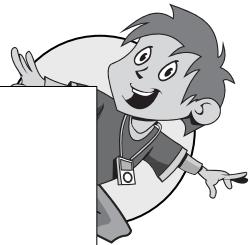
Dies enthält die Elemente 2 bis 4 von `neue_liste`

```
>>> neue_liste[3:4]  
[4.5]
```

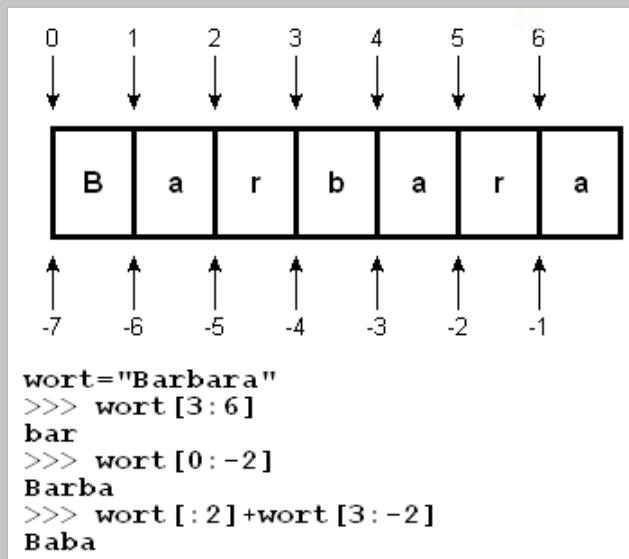
Dies enthält nur das Element mit Index 3 von `neue_liste`. Neugierige Geister wollen jetzt sicher wissen, was herauskommt, wenn die beiden Indizes der Scheibe gleich sind:

```
>>> neue_liste[3:3]  
[]
```

Solche Scheiben werden durch ein *Paar* von Indizes festgelegt: `[a:b]`. Die entsprechende Scheibe enthält als erstes Element das Element der ursprünglichen Liste mit dem Index a, das mit dem Index b aber gerade nicht mehr. Du kannst dir das mit folgendem Bild veranschaulichen.



11



Stelle dir das so vor, dass die Indizes jeweils zwischen die Elemente der Liste zeigen, und zwar jeder vor das Element mit der entsprechenden Nummer, dann enthält die Scheibe gerade die Elemente zwischen den Indizes.

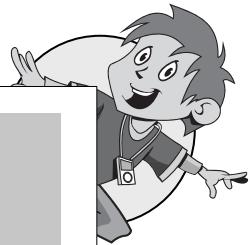
Dasselbe geht für Listen, Tupel und Strings, sogar für direkt hingeschriebene:

```
>>> stringC[11:30]
'und noch eine Zeile'
>>> tupelB[1:2]
((2, 3),)
>>> "Barbara"[3:6]
'bar'
```

Lässt du in einer Scheibe einen der beiden Indizes weg, dann wird die Scheibe vom entsprechenden Ende der Ausgangssequenz gebildet. Das heißt, sie wird vom Anfang weg gebildet, wenn der erste Index fehlt:

```
>>> neue_liste[:3]
[3, 3.5, 4]
neue_liste[:3] bedeutet also dasselbe wie neue_liste[0:3].
Ebenso ist neue_liste[1:] dasselbe wie
neue_liste[1:len(neue_liste)]:
>>> neue_liste[1:]
[3.5, 4, 4.5, 'red', 'blue', 'yellow']
```

Sequenzen



Fehlen beide Indizes, so erhältst du eine Kopie der Liste:

```
>>> neue_liste[:]  
[3, 3.5, 4, 4.5, 'red', 'blue', 'yellow']
```

Die Scheibenbildung ist auch nicht pingelig, wenn du Indizes verwendest, die in der Sequenz gar nicht vorkommen:

```
>>> (1,2,3)[-2:6]  
(2, 3)  
>>> (1,2,3)[6:1001]  
()  
>>> (1,2,3)[1001:]  
()  
>>> "abc"[20:30]  
''
```

Das ermöglicht ganz witzige Konstruktionen. Angenommen, du möchtest von einer Liste unbekannter Länge das vierte Element, außer sie hat weniger als vier Elemente, dann möchtest du einfach das letzte. Wahrscheinlich würdest du es so machen:

```
if len(l) < 4:  
    a = l[-1]  
else:  
    a = l[3]
```

Es geht aber auch so: `a = l[:4][-1]`

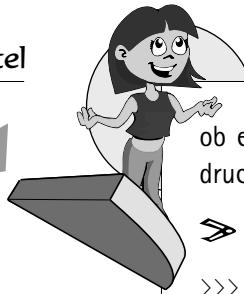
```
>>> l = [5,6,7,8,9]  
>>> l[:4]  
[5, 6, 7, 8]  
>>> l[:4][-1]  
8  
>>> l = [5,6]  
>>> l[:4]  
[5, 6]  
>>> l[:4][-1]  
6
```

Very Python Special!!

Der logische Operator »in«

Weiter oben haben wir gesehen, dass die Operatoren `+` und `*` zum Verketten und Vervielfachen von Listen benutzt werden können. Ein weiterer wichtiger Operator für Sequenzen ist der boolesche Operator `in`. Du verwendest ihn, um mit dem Ausdruck `element in sequenz` festzustellen,

11



ob ein Element in einer Sequenz enthalten ist. Ebenso gültig ist der Ausdruck `element not in sequenz`.

⇒ Mach weiter mit:

```
>>> "red" in listeB  
True  
>>> "e" in "red"  
True  
>>> "a" in "red"  
False
```

Nur für Strings hat `in` noch eine erweiterte Funktion:

```
>>> "re" in "red"  
True
```

Für Strings prüft `in` also, ob der erste String ein Teilstring des zweiten ist.

Weitere Beispiele:

```
>>> 2 in tupelB  
False  
>>> 2 not in tupelB  
True  
>>> (2,3) in tupelB  
True  
>>> 2 in tupelB[1]  
True
```

Methoden von Sequenzen

Am Anfang dieses Kapitels haben wir gesehen: Methoden sind Funktionen, die an Objekte gebunden sind. Sie werden mit der speziellen Punkt-Notation aufgerufen.

Für die Methoden der hier behandelten Sequenzen gilt:

- ❖ Tupel haben nur zwei Methoden: `count()` und `index()`. Die funktionieren so wie bei Strings und Listen.
- ❖ Strings und Listen haben wenige gemeinsame und viele unterschiedliche Methoden. Und das hat tiefere Gründe!

Strings und Listen haben eine ganze Reihe von Methoden. Man kann ihnen also viele verschiedene Botschaften schicken. Hier will ich dir nur ein paar Beispiele geben. Wenn wir später weitere brauchen, werde ich sie an Ort



und Stelle vorstellen. Die folgenden Beispiele sollen aber helfen, einen wichtigen Unterschied von String- und Listenobjekten zu klären.

Methoden von Strings

Betrachten wir zunächst einige Methoden von Strings. Wir arbeiten wieder mit den Strings aus `sequenzen.py`:

```
>>> stringA, stringB, stringC, stringD  
('X', 'bad', 'Eine Zeile\nund noch eine Zeile\nund noch eine  
Zeile.', '')
```

Doch zunächst zwei Methodenaufrufe für direkt eingegebene String-Objekte:

```
>>> "gross".upper()  
'GROSS'  
>>> "KLEIN".lower()  
'klein'  
>>> stringA.lower()  
'x'  
>>> stringB.upper()  
'BAD'
```

Die Methoden `lower()` und `upper()` erzeugen aus den Strings, für die sie aufgerufen werden, neue Strings und geben sie zurück. Es sind also Methoden mit Rückgabewert. Die ursprünglichen Strings bleiben dabei unverändert.

```
>>> stringA, stringB  
('X', 'bad')
```

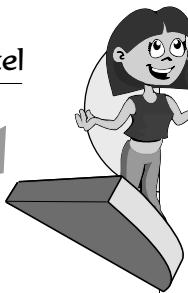
Wenn du mit den neu erzeugten Strings weiterarbeiten willst, musst du ihnen Namen geben:

```
>>> kleinA = stringA.lower()  
>>> grossB = stringB.upper()  
>>> kleinA, grossB  
('x', 'BAD')
```

Natürlich kannst du einem neu erzeugten String auch den alten Namen zuweisen. Dann verweist dieser Name auf den neuen String, der alte ist verschwunden. Wir prüfen das mit der eingebauten Python-Funktion `id()`, die eine Zahl als eindeutige Kennung für das Objekt zurückgibt, das ihr als Argument übergeben wurde.

```
>>> teststring = "test"  
>>> id(teststring)  
10015656
```

11



```
>>> teststring = teststring.upper()
>>> id(teststring)
10789912
```

Du siehst, dass `teststring` nun auf ein *anderes* String-Objekt verweist.

Eine Stringmethode, die für ein String-Objekt aufgerufen wird, gibt einen Wert aus, sie ändert aber nicht das betreffende String-Objekt. Anders gesagt: Schickst du an ein String-Objekt eine Botschaft, so reagiert dieses damit, ein anderes Objekt zurückzugeben. Dabei ändert es sich selbst nicht!

Das zurückgegebene Objekt muss keineswegs bei jeder Methode ein String sein. Hier einige Methodenaufrufe, die andere Objekte zurückgeben:

```
>>> stringC.count("n")
7
>>> stringC.startswith("Eine")
True
>>> stringC.startswith("Keine")
False
>>> stringC.split()
['Eine', 'Zeile', 'und', 'noch', 'eine', 'Zeile', 'und',
 'noch', 'eine', 'Zeile.']
```

Die Methode `count()` gibt eine Zahl zurück: wie oft das Argument in dem String vorkommt. Die Methode `startswith()` gibt einen booleschen Wert zurück: wahr oder falsch, je nachdem, ob das übergebene Argument mit dem Anfang des Strings übereinstimmt. Und die Methode `split()` gibt eine Liste von Strings zurück, nämlich die Liste der Wörter, aus denen der String besteht.

`split()` kann auch als Argument einen String *trenner* übernehmen, zum Beispiel mit "Zeile" als *trenner*:

```
>>> stringC.split("Zeile")
['Eine ', '\nund noch eine ', '\nund noch eine ', '.']
```

`split(trenner)` gibt eine Liste von Teilstrings zurück, die aus dem String, für den `split()` aufgerufen wird, entsteht, indem dieser bei jedem Auftreten von *trenner* geteilt wird. Die Teilstrings *trenner* werden dabei entfernt. Wird `split()` ohne Argument aufgerufen, spielen Zwischenräume, also Leerzeichen, aber auch *newline*-Zeichen, die Rolle von *trenner*.

Eine Aufzählung der Methoden von Strings findest du in der Python-Dokumentation in der »Library Reference / The Python Standard Library« im Abschnitt 5.6.1 String Methods. (Von der IDLE aus: HELP|PYTHON DOCS F1)

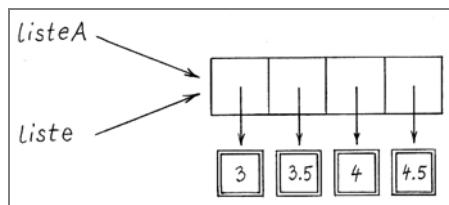


Methoden von Listen:

Unter den Methoden von Listen-Objekten ist vielleicht die wichtigste die Methode `append()`. `append()` wird dazu benutzt, an Listen Elemente anzuhängen. Zuerst sehen wir uns an, wie das funktioniert:

» Mach weiter mit!

```
>>> listeA
[3, 3.5, 4, 4.5]
>>> liste = listeA
>>> liste
[3, 3.5, 4, 4.5]
```

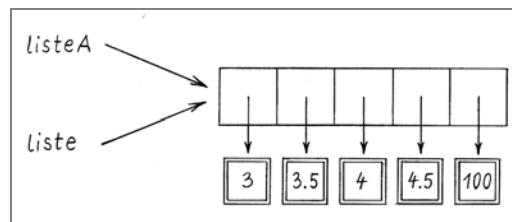


Zwei Namen für eine Liste.

```
>>> liste.append(100.0)
>>>
```

Siehe da, die Methode `append()` hat keinen Rückgabewert! (Sie hat, wie wir wissen, den Rückgabewert `None`, doch weiß der IPI, dass uns dieser Wert nicht interessiert.)

```
>>> liste
[3, 3.5, 4, 4.5, 100.0]
```

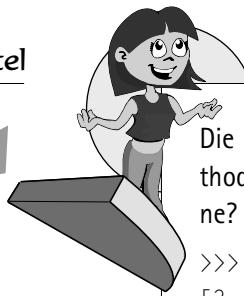


`append()` ändert das Listenobjekt.

Und doch hat sie eine Wirkung: die Liste hat sich geändert. Das Element `100.0` ist hinzugekommen. Weil aber `append()` keinen Rückgabewert hat, ist es sinnlos, zu schreiben

```
>>> neue_liste = liste.append(150.0) # SINNLOS!
>>> neue_liste
>>> print(neue_liste)
None
>>> liste
[3, 3.5, 4, 4.5, 100.0, 150.0]
```

11



Die Methode `append()` ändert das Listen-Objekt selbst! So, wie die Methode `color()` den Zustand einer Turtle ändert. Weißt du, was ich meine? Wenn ja, dann wird dich Folgendes nicht überraschen:

```
>>> listeA
[3, 3.5, 4, 4.5, 100.0, 150.0]
>>>
```

Es sind `listeA` und `liste` zwei Namen für *dieselbe* Liste, weil wir oben geschrieben haben: `liste = listeA`. Auch hier gibt `id()` Auskunft.

```
>>> id(liste)
10655120
>>> id(listeA)
10655120
>>> listeA.append(-1)
>>> liste
[3, 3.5, 4, 4.5, 100.0, 150.0, -1]
>>> id(listeA)
10655120
```

`append()` ist also eine Methode, die einen *Effekt* hat: Sie ändert das Objekt, für das sie aufgerufen wird. Sie ändert aber nicht die Identität des Listen-Objekts. (Das ist so ähnlich wie bei dir: Du hast deine *Identität* in den letzten zehn Jahren auch nicht geändert. Du bist du! Obwohl du auch größer geworden bist oder klüger ... also, obwohl du dich geändert hast.)

Manchmal möchtest du vielleicht eine gegebene Liste ändern, aber doch das alte Exemplar bewahren. Das heißt, du möchtest eigentlich eine Kopie einer Liste erzeugen. Das geht mit Scheibenbildung. Denn Scheiben sind immer neue Objekte.

➤ Mach mit:

```
>>> liste = [1, 2, 3]
>>> alias = liste
>>> kopie = liste[:]
>>> liste == kopie
True
>>> liste is kopie
False
>>> liste is alias
True
>>> liste, alias, kopie
([1, 2, 3], [1, 2, 3], [1, 2, 3])
```

Hier ist `alias` ein anderer Name für `liste`. (Alias ist diesmal ein lateinisches Wort und bedeutet so viel wie »anderer Name«.) `kopie` ist hingegen



der Name für ein anderes, aber gleichwertiges Objekt, also tatsächlich für eine Kopie.

```
>>> ([1, 2, 3], [1, 2, 3], [1, 2, 3])
>>> liste.append(4)
>>> liste, alias, kopie
([1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3])
```

Es gibt natürlich viele Wege, Listen zu ändern, nicht nur durch Aufruf der Methode `append()`. Zwei wichtige davon möchte ich hier noch vorstellen:

Ändern von Listenelementen durch Zuweisungen

Möchtest du zum Beispiel, dass das zweite Element von `liste`, das ist das Element mit dem Index 1, die Zahl 1001 wird, dann kannst du ihm den Wert 1001 zuweisen:

```
>>> liste[1] = 1001
>>> liste
[1, 1001, 3, 4]
```

Natürlich wirkt sich das in unserem Beispiel auch auf `alias`, nicht aber auf `kopie` aus:

```
>>> alias, kopie
([1, 1001, 3, 4], [1, 2, 3])
>>> kopie[2] = -1
>>> liste, alias, kopie
([1, 1001, 3, 4], [1, 1001, 3, 4], [1, 2, -1])
>>>
```

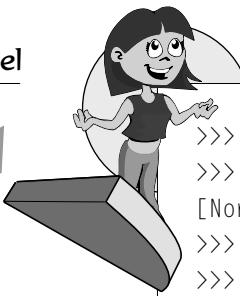
Bei Zuweisungen an Elemente einer Liste ist jedoch stets große Genauigkeit geboten, denn:

```
>>> kopie[3] = 71
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in ?
    kopie[3] = 71
IndexError: list assignment index out of range
```

Man kann nur solchen Listenelementen Werte zuweisen, die bereits einen Wert haben. Sonst erhält man einen Zuweisungsfehler, weil der Index außerhalb des zulässigen Bereichs liegt.

Brauchst du einmal eine Liste mit einer vorgegebenen Anzahl von Elementen, deren Werte du aber noch nicht kennst, sondern erst später zuweisen willst, kannst du folgenden Trick verwenden:

11



```
>>> meine_liste = [None] * 5
>>> meine_liste
[None, None, None, None, None]
>>> meine_liste[4] = 12345
>>> meine_liste
[None, None, None, None, 12345]
```



Strings und Tupel sind Sequenzen, die im Gegensatz zu Listen nicht veränderbar sind:

```
>>> ein_tupel = (5, 3, 0)
>>> ein_tupel[2]
0
>>> ein_tupel[2] = 1
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in ?
    ein_tupel[2] = 1
TypeError: 'tuple' object doesn't support item assignment
Strings und Tupel unterstützen deshalb Wertzuweisungen nicht.
```

➤ Führe eine ganz gleichartige Übung für den String "Laser" aus: Versuche mittels Elementzuweisung, das a gegen ein e zu tauschen.

Poppen von Listenelementen

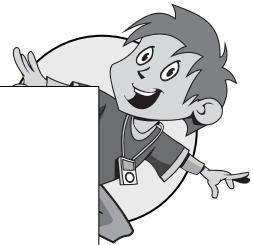
Recht oft tritt die Situation auf, dass man von einer Liste das letzte Element entnehmen will, um damit weiterzuarbeiten. Die Liste soll es aber dann nicht mehr enthalten. Dafür gibt es die Methode `pop()`. (Das Wort *pop* hat, glaube ich, hier mehr mit Popcorn zu tun als mit Popmusik.) Sie ist wieder einmal so ein Beispiel für eine Methode, die einen Effekt hat (sie verkürzt die Liste) *und* einen Rückgabewert – nämlich das ehemals letzte Listenelement.

➤ Mach weiter mit:

```
>>> liste.pop()
4
>>> liste
[1, 1001, 3]
```

So! Das war zwar rasch experimentiert, aber die 4 ist jetzt weg. Zum Glück brauchen wir sie hier nicht wieder. Wenn man eine Methode (oder Funktion) mit Rückgabewert verwendet, sollte man diesen einem Namen zuweisen oder einem Funktions-/Methodenauftruf als Argument übergeben.

Methoden von Sequenzen



```
>>> letztes = liste.pop()
>>> print(liste, letztes)
[1, 1001] 3
>>> letztes = liste.pop()
>>> print(liste, letztes)
[1] 1001
>>> letztes = liste.pop()
>>> print(liste, letztes)
[] 1
>>> letztes = liste.pop()
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in ?
    letztes = liste.pop()
IndexError: pop from empty list
```

Es ist eigentlich nicht verwunderlich, dass man von der leeren Liste kein Element mehr herunterpoppen kann.

Was aber schon geht, ist, ein Element von einer beliebigen Stelle einer Liste herunterzupoppen. Man muss dazu nur seinen Index kennen. Beispiele:

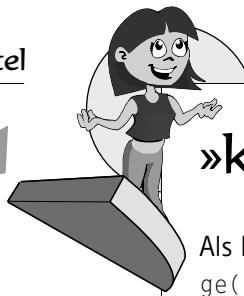
```
>>> liste = range(5, 14, 2)
>>> liste
[5, 7, 9, 11, 13]
>>> element = liste.pop(0)
>>> element
5
>>> liste
[7, 9, 11, 13]
>>> element = liste.pop(2)
>>> element
11
>>> liste
[7, 9, 13]
```

Objekte können angesprochen werden, solange ein Name auf sie verweist. Die Namen können wechseln, es können auch mehrere Namen auf dasselbe Objekt verweisen.

Bestimmte Arten von Objekten können geändert werden, sie sind veränderbar. Listen sind solche Objekte. Für Strings und Tupel trifft das nicht zu, Strings und Tupel sind unveränderbar. Sie können nur in veränderter Form neu gebildet werden.



11



»krange()«

Als Beispiel möchte ich hier eine Funktion definieren, die ähnlich wie range() funktioniert, aber Kommazahlen zurückgibt. Erinnere dich:

```
>>> list(range(7,15,3))
[7, 10, 13]
```

range(7,15,3) berechnet alle Zahlen, beginnend mit 7, mit Abstand 3, die kleiner als 15 sind. list(range(7, 15, 3)) ergibt dann eine Liste mit diesen Zahlen. Wird das dritte Argument nicht angegeben, hat es den Standardwert 1. range() kann auch mit nur einem Argument aufgerufen werden, das als Stoppwert interpretiert wird. Der Startwert hat dann den Standardwert 0.

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

Wir wollen eine Funktion krange() zunächst so definieren, dass sie eine Liste von Dezimalzahlen zurückgibt. krange() soll mit Kommazahlen arbeiten und zwar folgendermaßen:

```
>>> krange(0.75, 4.3, 1.75)
[0.75, 2.5, 4.25]
```

Ich gebe dir eine Definition dieser Funktion an, die du dir selbst durchdenken solltest. Es geht mir hier nicht um die Erklärung des Algorithmus, sondern um eine Demonstration, wie Listen-Objekte erzeugt werden können.

➤ Gib den Code der folgenden Funktion im Editor in eine Datei listenzeug.py ein, speichere ihn und führe ihn aus.

```
def krange(start, stop, schritt=1):
    zahlenliste = []
    element = float(start) # macht aus start eine Kommazahl!
    while element < stop:
        zahlenliste.append(element)
        element = element + schritt
    return zahlenliste
```

Erklärung: In der ersten Zeile wird ein leeres Listenobjekt erzeugt und erhält den lokalen Namen zahlenliste. Dann wird in der while-Schleife bei jedem Schleifendurchgang ein Element an zahlenliste angehängt und somit die Liste geändert. Verstehst du also, warum wir hier eine Liste verwenden? Mit Tupeln wäre das nicht möglich, denn Tupel sind unveränderbar. Zuletzt wird das Listenobjekt zurückgegeben. Beachte, dass in die-

»krange()« – als Generator ...



ser Form die Funktion krange() *nicht* mit negativen Schrittweiten funktioniert! Ein Mangel (und eine Gelegenheit für Verbesserungen).

⇒ Teste krange() z.B. wie folgt, aber auch mit anderen Eingaben:

```
>>> krange(3.25, 7.5, 1.25)
[3.25, 4.5, 5.75, 7.0]
>>> for k in krage(3.25, 7.5, 1.25):
    print(k, end=", ")
```

3.25, 4.5, 5.75, 7.0,

⇒ Kopiere den Code von krange in mytools.py.

Du siehst, krange() kann in der for-Schleife so verwendet werden wie range(). Intern funktioniert es aber doch ganz anders: Es erzeugt zunächst die ganze Werteliste, die dann von der for-Schleife abgearbeitet wird.

Da stellt sich natürlich die Frage, ob wir krange() auch so programmieren können, dass es (wenigstens so ähnlich) wie range() funktioniert. Einen einfachen, wenn auch nicht perfekten, Weg dazu möchte ich dir jetzt zeigen:

»krange()« – als Generator ...

... oder: ein flüchtiger Blick hinter das Geheimnis von range(). Du hast range() als dynamischen Wertevorrat kennengelernt. Nun wollen wir krange() auch zu einem (ähnlich gebauten) dynamischen Wertevorrat machen.

Der Inhalt dieses kurzen Abschnitts wird im weiteren Verlauf des Buchs nicht gebraucht. Solltest du ihn also überspringen wollen oder nicht gleich ganz verstehen, so hast du keine unerwünschten Nebenwirkungen zu befürchten.

Wir werden jetzt ein Objekt erzeugen, das die Werte von krange() erzeugt, »generiert«, nun aber nicht auf einmal, sondern einen nach dem anderen und von einer for-Schleife abrufbar. Um damit nicht unsere alte Definition von krange() zu überschreiben, nennen wir die neue frange() – denke beim f an float.

Die Definition, die wir dazu verwenden, sieht auf den ersten Blick aus wie eine Funktionsdefinition:



```
def frange(start, stop, schritt=1):
    """ein Generator von floating-point-Zahlen
    von start bis stop mit Schrittweite step."""
    element = float(start) # macht Kommazahl
    while element < stop:
        yield element
        element = element + schritt
```

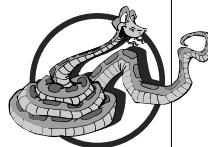
Und sie sieht sogar einfacher aus als die vorige Definition. Sie enthält aber etwas Neues, nämlich das reservierte Wort `yield`. `Yield` heißt so viel wie abgeben, liefern. Und damit ist die Definition eine ganz besondere Funktionsdefinition: Sie hat als Rückgabewert einen *Generator*. Deshalb nennt man eine derartige Funktion eine Generator-Funktion.

Clara Pythias Python-Special: Generator-Funktionen und Generatoren

Generatoren sind Objekte, die eine Folge von Werten liefern, aber nicht auf einmal, wie Sequenzen, sondern einen Wert nach dem anderen immer erst dann, wenn er gebraucht wird.

Generator-Objekte werden von Generator-Funktionen erzeugt und zurückgegeben. Generator-Funktionen sind Funktionen, die das reservierte Wort `yield` enthalten.

➤ Mach mit:



```
>>> def zaehle(bis):
    z = 0
    while z < bis:
        yield z
        z = z + 1
```

```
>>> zaehle
<function zaehle at 0x01360660>
```

Wir benutzen die Generator-Funktion `zaehle()`, um ein Generator-Objekt, kurz einen Generator, zu erzeugen:

```
>>> zaehler = zaehle(6)
>>> zaehler
<generator object zaehle at 0x0135CF80 >
```



Der erzeugte Generator hat nun den Namen `zaehler`. Die Werte, die der Generator liefert, kannst du auf zwei Arten abrufen:

Erstens durch Aufruf der in Python eingebauten Funktion `next()` mit dem Generator als Argument:

```
>>> next(zaehler)  
0
```

Die erste `yield`-Anweisung liefert den Wert 0. Die `yield`-Anweisung funktioniert also ähnlich wie die `return`-Anweisung, aber mit dem wichtigen Unterschied, dass danach die Funktion nicht verlassen wird, sondern ihren Zustand beibehält und nur pausiert. Beim nächsten Aufruf von `next(zaehler)` wird die `z` um eins erhöht und im nächsten Schleifendurchgang wieder durch `yield` abgeliefert usw.

```
>>> next(zaehler)  
1  
>>> next(zaehler)  
2
```

Zweitens kannst du den Generator auch als Wertevorrat für die for-Schleife verwenden:

```
>>> for z in zaehler:  
    print(z, end=" ")  
  
3 4 5 6
```

Hier hat er natürlich nur noch die noch unverbrauchten Werte geliefert. Ein Generator kann seine Werte nur einmal liefern – im Gegensatz zu einem `range`-Objekt.

```
>>> for z in zaehler:  
    print(z, end=" ")  
>>>
```

Was geschieht, wenn wir die Funktion `next()` für einen ausgeschöpften Generator aufrufen?

```
>>> next(zaehler)  
Traceback (most recent call last):  
  File "<pyshell#11>", line 1, in <module>  
    next(zaehler)  
StopIteration  
>>>
```

11



Es wird eine StopIteration-Ausnahme erzeugt. Wenn das innerhalb einer for-Schleife geschieht, bricht diese einfach ab.

Natürlich kann man sich jederzeit mit `zaehle()` einen neuen Generator machen:

```
>>> for z in zaehle(12):
    print(z, end=" ")
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12
```

Fassen wir zusammen: Ein Generator funktioniert so, dass die `yield`-Anweisung einen Wert zurückgibt und dann der Ablauf des Generators pausiert, bis mit `next()` der nächste Wert abgerufen wird, usw.

Die `for`-Schleife weiß das und holt – sozusagen hinter den Kulissen – für jeden Schleifendurchgang automatisch den nächsten Wert des Generators ab, bis eine `StopIteration`-Ausnahme auftritt.

Wenn du dir nun den Code für die Generator-Funktion `frange()` ansiehst, wirst du erkennen, dass sie genau denselben Aufbau hat wie die Generator-Funktion `zaehle()`.

» Gib nun den obenstehenden Code der Generator-Funktion `frange()` in ein Editor-Fenster ein, speichere die Datei unter dem Namen `frange_generator.py` ab und führe das Skript aus.

» Nun sollte zum Beispiel Folgendes funktionieren:

```
>>> frange(3.25, 7.5, 1.25)
<generator object frange at 0x0135CFA8>
>>> for f in frange(3.25, 7.5, 1.25):
    print(f, end=", ")
```

```
3.25, 4.5, 5.75, 7.0,
```

» Wenn du `frange` später verwenden möchtest, kopiere den Code ebenfalls in `mytools.py`.

Es gibt in Python einige Objekt-Typen, die ähnlich wie `range()` und unser damit verwandtes `frange()` funktionieren. Ich werde ein Objekt dieser Art weiterhin (wie bisher) als *Wertevorrat* bezeichnen. Eine genauere Untersuchung dieser Typen ist für dieses Buch zu schwierig.



Wikipedia-Beispiel, revisited

Zum Abschluss dieses Kapitels schlage ich vor, das Wikipedia-Beispiel aus Kapitel 8 neu zu implementieren, und zwar so, dass die 36 Sechsunddreißigecke parallel – also quasi gleichzeitig – gezeichnet werden:

Die Grundidee ist ähnlich wie bei unseren Dreiecksspiralen. Wir lassen jedes Polygon durch eine eigene Turtle zeichnen. Und zwar in der Weise, dass alle Turtles zuerst das erste Segment zeichnen. Wenn das alle getan haben, zeichnen alle das zweite Segment usw.

Parallele Entstehung der Polygone im Wikipedia-Beispiel

Wir wollen auch diesmal nur mit den Klassen Screen und Turtle aus dem Modul turtle arbeiten. Dabei wirst du einige Methoden der Klasse Screen kennen lernen, die du schon als Funktionen des Moduls turtle kennst.

Screen() – ein Singleton

Screen ist eine Klasse mit einer besonderen Eigenschaft: Man kann nämlich nur *ein* Objekt von ihr erzeugen.

» Starte IPI-TURTLEGRAFIK und mach mit!

```
>>> from turtle import Screen
>>> screen1 = Screen()
>>> screen1.bgcolor("red")
```

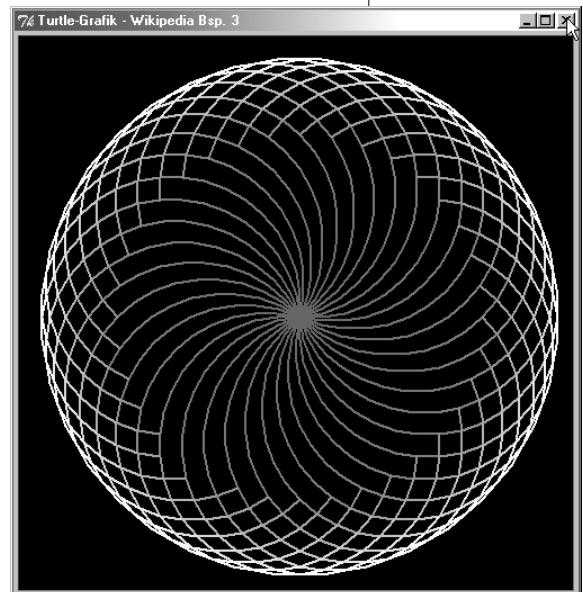
Ein Turtle-Grafik-Fenster geht auf: Wir haben ein Objekt der Klasse Screen() erzeugt. Versuchen wir, noch eines zu erzeugen:

```
>>> screen2 = Screen()
```

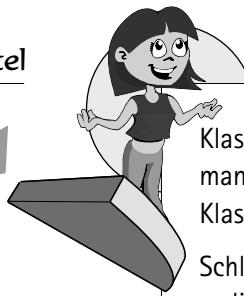
Nichts passiert!

```
>>> id(screen1) == id(screen2)
True
>>> screen2.setup(200,200)
```

Aha! Es gibt wirklich nur ein einziges. Dieses hier hat nun zwei Namen: screen1 und screen2. Die einzige Möglichkeit, ein neues Grafik-Fenster zu erzeugen, ist, das erste zu schließen und neuerlich Screen() aufzurufen.



11



Klassen, von denen immer nur ein einziges Objekt existieren kann, nennt man im OOP-Sprech *Singleton-Klassen*. `Screen()` ist so eine Singleton-Klasse.

Schließe nun das Turtle-Grafik-Fenster. Wir wenden uns wieder dem »Wikipedia-Beispiel« zu.

➤ Öffne ein Editor-Fenster, um einen Kopfkommentar und danach folgenden Code einzugeben:

```
from turtle import Screen, Turtle

def parallel_super_rosette(ecken, seite):
    pass

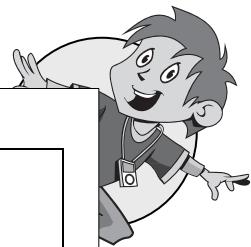
def main():
    global screen
    screen = Screen()
    screen.title("Turtle-Grafik - Wikipedia Bsp. 3")
    screen.clearscreen()
    screen.setup(500, 500)
    screen.bgcolor("black")
    screen.tracer(False)
    parallel_super_rosette(36, 20)

if __name__ == "__main__":
    main()
```

➤ Speichere das Programm unter dem Namen `wikipedia03.py` ab. Führe es aus. Funktioniert es fehlerfrei? Wenn du ein quadratisches schwarzes leeres Fenster siehst und keine Fehlermeldung, wahrscheinlich schon. Steht der Titel richtig in der Titelleiste des Fensters?

Als Nächstes brauchen wir eine Kollektion von 36 Turtles, allesamt mit weitgehend den gleichen Eigenschaften, aber jede nach einem anderen Winkel orientiert. Nachdem wir schon wissen, dass Funktionen beliebige Dinge zurückgeben können, machen wir uns nun eine, die eine Turtle zurückgibt, die in eine bestimmte, durch einen Winkel angegebene Richtung schaut.

➤ Füge den folgenden Code gleich nach der `import`-Anweisung ein:



```
from turtle import Screen, Turtle

def myturtle(winkel):
    turtle = Turtle()
    turtle.speed(0)
    turtle.pensize(2)
    turtle.setheading(winkel)
    return turtle

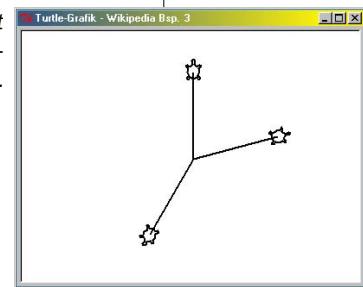
def parallel_super_rosette(ecken, seite):
```

» Kommentiere vorübergehend die `if __name__ == "__main__":-`-Anweisung aus. Falls noch ein Grafik-Fenster offen ist, schließe es. Speichere das Programm und führe es erneut aus.

» Mach mit, im IPI-Shell-Fenster:

```
>>> a = myturtle(0)
>>> b = myturtle(75)
>>> c = myturtle(210)
>>> for krot in a,b,c:
        krot.fd(100)
```

Drei myturtle()-s, mit unterschiedlichen Winkeln erzeugt.



Wir brauchen aber nicht nur drei, sondern 36. Da wäre es ziemlich fad, für jede einen eigenen Namen zu erfinden. Daher tun wir sie lieber in eine Liste! Bei dieser Gelegenheit will ich dir ein ganz elegantes Verfahren zur Erzeugung von Listen zeigen:

List Comprehension

Mit »List Comprehension« – wofür es keinen gebräuchlichen deutschen Ausdruck gibt – erzeugt man Listen aus anderen (meist einfacheren) Listen. Das Verfahren ähnelt dem der Beschreibung von Mengen in der Mathematik.

» Schließe das Grafik-Fenster und mach mit!

```
>>> list(range(8))
[0, 1, 2, 3, 4, 5, 6, 7]
>>> [2 * x for x in range(8)]
[0, 2, 4, 6, 8, 10, 12, 14]
>>> [i * i for i in range(8)]
[0, 1, 4, 9, 16, 25, 36, 49]
>>> [j * "c" for j in range(6)]
['', 'c', 'cc', 'ccc', 'cccc', 'ccccc']
```

11



Für jedes Element einer Ausgangsliste wird ein neues gebildet und in die neue Liste aufgenommen. Im ersten Beispiel ist das immer das Doppelte, im zweiten das Quadrat des Ausgangselements.

Eine »List Comprehension« schreibst du auf, indem du zwischen Listenklammern zuerst einen »Ausdruck« schreibst, der beschreibt, welche neuen Elemente gebildet werden sollen. Danach folgt immer eine Floskel der Form: `for index in Wertevorrat`.

Im letzten Beispiel ist der »Ausdruck« `j * "c"`. Der `index` ist `j` und der `Wertevorrat` ist `range(6)`. Somit nimmt `j` die ganzen Zahlen von 0 bis 5 an. Die Liste enthält daher Vervielfachungen von `"c"`.

Der Ausdruck darf aber ruhig auch ein Funktionsaufruf sein, z.B. einer von `myturtle()`:

```
>>> kroeten = [myturtle(w) for w in (0, 75, 210)]
>>> for krot in kroeten:
    krot.forward(100)
```

Diese Anweisungen erzeugen die gleiche Grafik, wie das letzte Beispiel oben. Du willst wissen, was nun eigentlich die Liste `kroeten` ist?

```
>>> kroeten
[<turtle.Turtle object at 0x03C36350>, <turtle.Turtle object at 0x03C36310>, <turtle.Turtle object at 0x03C2CD50>]
```

Eine Liste mit drei Turtle-Objekten. So richtig praktisch wird das natürlich, wenn man viele Turtles braucht.

⇒ Schließe wieder das Grafik-Fenster!

⇒ Mach mit!

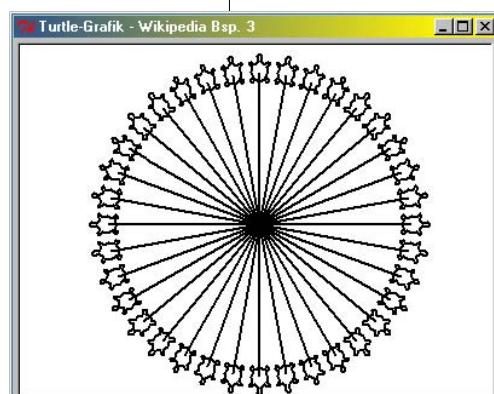
```
>>> winkel = 360/36
>>> winkel
```

10.0

```
>>> [i*winkel for i in range(36)]
[0.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0,
 70.0, 80.0, 90.0, 100.0, 110.0, 120.0,
 130.0, 140.0, 150.0, 160.0, 170.0, 180.0,
 190.0, 200.0, 210.0, 220.0, 230.0, 240.0,
 250.0, 260.0, 270.0, 280.0, 290.0, 300.0,
 310.0, 320.0, 330.0, 340.0, 350.0]
```

```
>>> kroeten = [myturtle(i*winkel) for i in range(36)]
```

```
>>> for krot in kroeten:
    krot.forward(125)
```





Jetzt überlegen wir uns noch etwas für die Farben. Wie früher nehmen wir rot = 1 - blau. Für die Berechnung der Blauwerte wenden wir jetzt, da wir schon List Comprehensions kennen, einen schlauen Trick an. Ich zeige dir das im IPI für, sagen wir, achtseitige Polygone:

» Mach mit!

```
>>> from mytools import krange  
>>> krange(-1, 1, 2/8)  
[-1.0, -0.75, -0.5, -0.25, 0.0, 0.25, 0.5, 0.75]
```

Der Wert 2 kommt daher, dass der Unterschied von 1 und -1 eben gleich zwei ist. Bis auf das Vorzeichen wären das die richtigen Zahlen für die Blauwerte. Sie müssen aber alle positiv sein. Oder anders gesprochen: Man muss von all diesen Zahlen den Absolutwert nehmen. Dafür gibt es in Python auch eine eingebaute Funktion: `abs()`.

» Mach weiter mit:

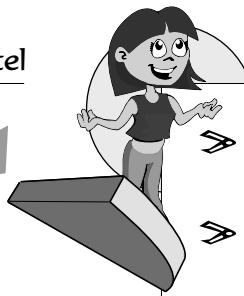
```
>>> abs(0.5)  
0.5  
>>> abs(-0.1)  
0.1  
>>> abs(-2)  
2  
>>> [abs(k) for k in krange(-1, 1, 2/8)]  
[1.0, 0.75, 0.5, 0.25, 0.0, 0.25, 0.5, 0.75]
```

Na, und mit 36 geht das wohl eben so. Damit gelangen wir zum Code von `parallel_super_rosette()`:

```
def parallel_super_rosette(ecken, seite):  
    winkel = 360 / ecken  
    turtles = [myturtle(i*winkel) for i in range(ecken)]  
    blauwerte = [abs(k) for k in krange(-1, 1, 2/ecken)]  
    for blau in blauwerte:  
        rot = 1 - blau  
        screen.tracer(False)  
        for turtle in turtles:  
            turtle.pencolor(rot, 0, blau)  
            turtle.forward(seite)  
            turtle.left(winkel)  
        screen.tracer(True)
```

» Füge diesen Code in `wikipedia03.py` ein.

11



- Importiere von mytools krange() in das Programm (am Anfang des Programmcodes).
- Entkommentiere wieder die if __name__ == "__main__" - Anweisung
- Füge eventuell noch einen Aufruf der Methode hideturtle() für turtle in myturtle() ein.
- Speichere das Programm und führe es aus. Beseitige allfällige Fehler.

Du kannst nun dein Werk bei der Programmausführung bewundern. Oder auch damit experimentieren, indem du im Aufruf von parallelе_ super_rosette() die Argumente änderst.

Zusammenfassung

- ❖ Objekte gehören entweder zu eingebauten Typen (Zahlen, Strings, Listen und so weiter) ...
- ❖ ... oder sind »Instanzen« benutzerdefinierter Klassen. Beispiel: Turtles sind Objekte der Klasse Turtle.
- ❖ Python stellt eine Vielzahl solcher Klassen in den Modulen seiner Bibliothek zur Verfügung.
- ❖ Objekte von benutzerdefinierten Klassen werden durch Aufruf eines Konstruktors erzeugt.
- ❖ Die »Fähigkeiten« von Objekten liegen in ihren Methoden.
- ❖ Methoden sind an Objekte gebundene Funktionen.
- ❖ Methoden werden mit der Punktschreibweise aufgerufen:
objektname.methodename(arg1, arg2,...)
- ❖ Sequenzen sind geordnete Folgen von Elementen.
- ❖ Strings, Tupel und Listen sind Sequenzen.
- ❖ Sequenzen können mit + verkettet und mit * vervielfacht werden.
- ❖ Auf die Elemente von Sequenzen wird über Indizes zugegriffen.
- ❖ Zahlen, Strings und Tupel sind unveränderbare Objekte.
- ❖ Listen sind veränderbare Objekte.
- ❖ append() und pop() sind Methoden, die Listen verändern.
- ❖ Listen können durch »List Comprehensions« erzeugt werden.

Einige Aufgaben ...

- ❖ Generatoren liefern Folgen von Werten, einen nach dem anderen bei Bedarf.
- ❖ Generatoren können durch Generator-Funktionen erzeugt werden. Die Definition von Generator-Funktionen muss das reservierte Wort `yield` enthalten.
- ❖ Das `turtle`-Modul stellt die Klassen `Screen` und `Turtle` zur Verfügung. Man kann daher mit mehreren/vielen Turtle-Objekten arbeiten
...
- ❖ ... aber nur mit einem `Screen`-Objekt. Denn `Screen` ist eine Singleton-Klasse.



Einige Aufgaben ...

Aufgabe 1: Erstelle eine Funktion, die wie folgt funktioniert:

```
>>> besprache("Das ist witzig!")  
'Dabas ibist wibitzibig!'
```

Aufgabe 2:

```
>>> txt = """Erstelle eine Funktion, die aus einem  
String, der einen längeren Text enthält, eine Liste  
jener Wörter erzeugt, die einen bestimmten Buchstaben  
enthalten."""
```

```
>>> woerter_mit(txt,"s")  
['Erstelle', 'aus', 'Liste', 'bestimmten', 'Buchstaben']
```

Aufgabe 3: Verbessere die Funktion `krange()` derart, dass sie auch für negative Werte des dritten Arguments, der Schrittweite, funktioniert.

... und einige Fragen

1. Strings haben die Methode `replace(old,new)`. Was kannst du mit dieser Feststellung anfangen?
2. Gegeben ist `a = [(1,2), (3,4), (5,6)]`. Was ist `a[2][1]`? Was ist `a[:2][1]`?
3. Welchen Effekt hat mit obiger Liste `a[1] = (8,9)`? Und welchen `a[1][1] = (8,9)`?
4. Which sequences are immutable, which are mutable?
5. Wozu dient das reservierte Wort `is`? Steht nicht im Buch! (Aber sicherlich woanders.)

12



Wörterbücher, Dateien und der alte Cäsar

Nachdem du im vorigen Kapitel den Umgang mit Objekten und ihren Methoden kennen gelernt hast, wirst du es hier mit zwei weiteren nützlichen Objekttypen zu tun bekommen, die für die Programmierpraxis unentbehrlich sind.

In diesem Kapitel wirst du ...

- ◎ mit dem Datentyp *Dictionary* erfahren, was Wörterbücher wirklich sind,
- ◎ sehen, wie man diese zum Beispiel für das Verschlüsseln von Daten mit der Methode des alten Cäsar anwenden kann
- ◎ und schließlich auch noch lernen, Text von *Dateien* zu *lesen* und in *Dateien* zu *schreiben*.

12



Dictionaries

Dictionary heißt Wörterbuch. Python hat einen zusammengesetzten Datentyp, der heißt dictionary. Dictionaries sind *mappings*, wieder englisch: Abbildungen. (Kennst du den Begriff aus der Mathematik?) Jedem Element einer Menge von so genannten *Schlüsseln* (keys) wird ein *Wert* (value) zugeordnet. Im Deutschen werden Dictionaries manchmal als »assoziative Felder« bezeichnet, weil sie Verbindungen (Assoziationen) von Schlüsseln mit Werten beinhalten – nur, falls dir der Begriff einmal über den Weg laufen sollte.

Dictionaries bestehen daher aus einer Menge von Paaren von Objekten, so genannten Schlüssel-Wert-Paaren. Dabei ist zu beachten, dass die Schlüssel unveränderliche Objekte sein müssen. Die Werte können dagegen beliebige Objekte sein.

In diesem Kapitel kommen keine Grafik-Anwendungen vor. Es ist daher sinnvoll, mit der Standard-IDLE zu arbeiten.

➤ Starte die IDLE (Python GUI) und mach mit!

```
>>> berufe = {"Guido": "Computer-Guru", "Eva": "Model",  
             "Fritz": "Lehrer", "Katja": "Lektorin"}
```

Achte gut auf die Syntax für das Anschreiben von Dictionaries. Die Schlüssel-Wert-Paare werden durch Beistriche voneinander getrennt. Zwischen Schlüssel und zugehörigem Wert steht ein Doppelpunkt. Das ganze Dictionary wird in *geschwungene Klammern* eingeschlossen. Unseres hat den Namen beruf. Zeig her, was in dir steht:

```
>>> berufe  
{'Eva': 'Model', 'Fritz': 'Lehrer', 'Katja': 'Lektorin',  
   'Guido': 'Computer-Guru'}
```

Beachte, dass die Einträge im Wörterbuch hier in einer anderen Reihenfolge auftauchen. Wörterbücher sind *ungeordnet*. Du hast keinen Einfluss auf irgendeine Art von Reihenfolge der Einträge!

➤ Frage nun nach den Werten im Dictionary berufe, die zu einzelnen Schlüsseln gehören:

```
>>> berufe["Fritz"]  
'Lehrer'  
>>> berufe["Katja"]  
'Lektorin'
```

➤ Füge einen neuen Eintrag hinzu:

Dictionaries



```
>>> berufe["Buffi"] = "Programmierer"  
>>> berufe  
{'Eva': 'Model', 'Fritz': 'Lehrer', 'Buffi': 'Programmierer',  
'Katja': 'Lektorin', 'Guido': 'Computer-Guru'}
```

Der Datentyp Dictionary hat die beiden Methoden `keys()` und `values()`. Sie haben keine Parameter und geben die Folgen der Schlüssel beziehungsweise der Werte zurück:

```
>>> berufe.keys()  
dict_keys(['Eva', 'Fritz', 'Buffi', 'Katja', 'Guido'])  
>>> berufe.values()  
dict_values(['Model', 'Lehrer', 'Programmierer', 'Lektorin',  
'Computer-Guru'])
```

`dict_keys` und `dict_values` sind wieder einmal so dynamische Wertevorrat-Objekte. Du kannst fragen, ob sie ein Objekt enthalten:

```
>>> "Eva" in berufe.keys()  
True  
>>> "Model" in berufe.values()  
True  
>>> "Skater" in berufe.values()  
False
```

(Leider kein Beruf!)

Für Dictionaries hat der Operator `in` eine besondere Bedeutung: Mit ihm prüfst du kürzer als oben, ob ein Objekt ein Schlüssel in einem Dictionary ist.

```
>>> "Eva" in berufe  
True  
>>> for name in berufe:  
    print("Der Beruf von {0} ist {1}."  
          format(name, berufe[name]))
```

```
Der Beruf von Eva ist Model  
Der Beruf von Fritz ist Lehrer  
Der Beruf von Buffi ist Programmierer  
Der Beruf von Katja ist Lektorin  
Der Beruf von Guido ist Computer-Guru  
>>> berufe["Inge"]
```

```
Traceback (most recent call last):  
  File "<pyshell#23>", line 1, in <module>  
    beruf["Inge"]  
KeyError: 'Inge'
```



Diese Fehlermeldung solltest du dir merken: `KeyError – Schlüsselfehler`. Sie tritt auf, wenn du nach einem Wert für einen Schlüssel suchst, den es im Wörterbuch nicht gibt. "Inge" kommt als Schlüssel im Dictionary `berufe` nicht vor!

Zuletzt stelle ich dir noch die Methode `items()` von Dictionaries vor. Sie gibt eine Liste der Schlüssel-Wert-Paare, also der Einträge im Dictionary, als Zweier-Tupel zurück:

```
>>> berufe.items()
dict_items([('Eva', 'Model'), ('Fritz', 'Lehrer'),
('Buffi', 'Programmierer'), ('Katja', 'Lektorin'),
('Guido', 'Computer-Guru'))]
```

Mit ihr kannst du die letzte `print`-Ausgabe auch so erzeugen:

```
>>> for name, beruf in berufe.items():
    print ("Der Beruf von {0} ist {1}.".format(name, beruf))
```

Wir wollen den Umgang mit Wörterbüchern zuerst in einem ganz kleinen Programm üben:

➤ Öffne ein Editor-Fenster der IDLE und schreibe einen Kopfkommentar für `passwort.py`.

Wir nehmen an, es ist ein Wörterbuch mit Benutzername-Passwort-Paaren gegeben. Schreibe folgendes Dictionary in die Datei `passwort.py`:

```
passwort = { "griemer" : "elli007",
             "kkrahl" : "Susim1",
             "kschrei" : "gottogott" }
```

Nun wollen wir eine Funktion `login()` definieren, die folgendermaßen funktioniert:

Funktion `login()`:

Parameterliste: leer

wiederhole immerfort:

`name` ⇔ Eingabe »Benutzername«

`pwd` ⇔ Eingabe »Passwort«

 Wenn das Paar `name, pwd` im Wörterbuch `passwort` vorkommt:

 Ausgabe: Begrüßung

 Ausstieg aus der Schleife

 Ausgabe: »Anmeldung misslungen«

Dictionaries



Es erheben sich drei Fragen:

- ❖ Wir haben hier eine »Endlosschleife« mit bedingtem Ausstieg im Schleifenkörper. Eine Endlosschleife bekommt man, indem man ausnützt, dass True niemals falsch ist, mit folgendem Schleifenkopf:

```
while True:  
    Schleifenkörper
```

- ❖ Wie soll die Prüfung der Eingaben genau geschehen? Es gibt dafür verschiedene Möglichkeiten. Zum Beispiel: prüfe mit der Methode items(), ob das Paar (name, pwd) in den Einträgen des Wörterbuchs vorkommt.
- ❖ Wenn ja, verlasse die Schleife! Wie macht man das? In Python gibt es für das Abbrechen von Schleifen eine spezielle Anweisung, die break-Anweisung. Sie bewirkt, dass die Schleife, in der die Anweisung steht, sofort beendet wird. (Bei geschachtelten Schleifen wird nur die innerste Schleife abgebrochen!) Du erkennst unschwer, dass break ein reserviertes Wort ist.

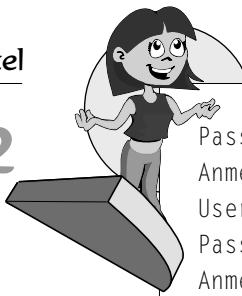
Diese Überlegungen führen zu folgendem Code:

```
def login():  
    while True: # endlose Anmeldeprozedur  
        name = input("Username: ")  
        pwd = input("Password: ")  
        if (name, pwd) in passwort.items():  
            print("Hi, {}! Have fun!".format(name[1:]))  
            break  
        print("Anmeldung misslungen!")
```

Testen wir unsere Funktion.

➤ Drücke **[F5]** und mach mit:

```
>>> ====== RESTART ======  
>>>  
>>> login()  
Username: kschrei  
Password: gottogott  
Hi, schrei! Have fun!  
>>> login()  
Username: kkrahl  
Password: Gabim1  
Anmeldung misslungen!  
Username: kkrahl
```



Password: Elfim1

Anmeldung misslungen!

Username: kkrah1

Password: Noraml1

Anmeldung misslungen!

Username: kkrah1

Password: Karom1

Anmeldung misslungen!

Username:

Da das nicht endlos so weitergehen kann, solltest du nun noch eine Abbruchmöglichkeit einbauen, falls dem Benutzer sein Passwort gerade nicht einfällt. (Ich hoffe, diese Situation kommt dir nicht bekannt vor!) Beispiel: Abbruch nach drei Abfragen (`passwort02.py`). Oder: Abbruch, wenn eine der beiden Eingaben leer ist.

Verschlüsseln

Verschlüsselung von Texten und von Daten ist heute eine ganz wichtige und ziemlich komplizierte Sache. Aber schon vor 2000 Jahren wurden Verfahren ersonnen, um Nachrichten zu verschlüsseln. Eines der ältesten und bekanntesten ist der so genannte Cäsar-Code, den Julius Cäsar verwendet hat, um militärische Nachrichten vor seinen Feinden im Gallischen Krieg geheim zu halten.

Wir werden jetzt erkunden, wie man mit Python Texte nach Cäsars Verfahren verschlüsseln kann. In diesem Abschnitt entwickeln wir kein Programm, das diese Verschlüsselung vornimmt. Wir erforschen jedoch alle Bestandteile, die so ein Programm braucht. Vielleicht hast du danach Lust, selbst ein derartiges Programm zu entwickeln.

Ich finde, dass das eine gute Gelegenheit ist, einmal selbstständig ein etwas umfangreicheres Programm zu entwerfen und zu erstellen.

Bevor wir mit einer interaktiven Sitzung mit der IDLE (PYTHON GUI) beginnen, muss ich dir jedoch erklären, wie das Verschlüsselungsverfahren mit dem Cäsar-Code geht:

Der Cäsar-Code

Beim Verschlüsselungsverfahren des Cäsar-Codes wird jeder Buchstabe eines gegebenen so genannten *Klartextes* durch einen anderen Buchstaben ersetzt. Dadurch entsteht der *Geheimtext*. Entscheidend ist nun die Ersetzungsvorschrift, die angibt, welcher Buchstabe durch welchen ersetzt wer-

Verschlüsseln



den soll. Beim Cäsar-Code ist diese Vorschrift sehr einfach. Sie wird durch die Angabe des Buchstabens, der das a ersetzen soll, festgelegt. Wenn beispielsweise a durch g ersetzt werden soll, ergibt sich die Ersetzungsvorschrift für die anderen Buchstaben aus Folgendem:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z  
g h i j k l m n o p q r s t u v w x y z a b c d e f
```

Jeder Buchstabe im Klartext wird durch den darunter stehenden ersetzt, also zum Beispiel i durch o oder u durch a. Es ist nahe liegend, diese Ersetzungsvorschrift durch ein Dictionary folgender Art zu beschreiben:

```
code = { "a": "g", "b": "h", "c": "i", ... }
```

und so weiter.

Wir werden also zuerst eine Funktion entwickeln, die uns zu einem gegebenen Startbuchstaben, der das a ersetzt, das zugehörige Code-Wörterbuch zurückgibt:

» Mach mit:

```
>>> klar = "abcdefghijklmnopqrstuvwxyz"  
>>> len(klar)  
26
```

Strings haben eine Methode `index()`, mit der du feststellen kannst, als wievieltes Zeichen (mit dem nullten beginnend!) ein bestimmter Buchstabe in dem String vorkommt.

```
>>> klar.index("a")  
0  
>>> klar.index("e")  
4  
>>> klar.index("z")  
25
```

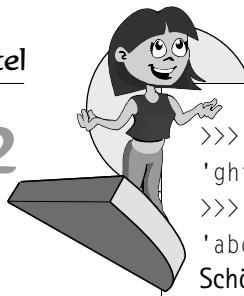
Klar!

```
>>> i = klar.index("g")  
>>> i  
6
```

Wir müssen das Klartextalphabet zerschnipseln und umordnen, um zum Geheimtextalphabet zu gelangen. Versuche es mit »Scheiben«:

```
>>> klar[i:]  
'ghijklmnopqrstuvwxyz'  
>>> klar[:i]  
'abcdef'  
>>> geheim = klar[i:]+klar[:i]
```

12



```
>>> geheim
'ghijklmnopqrstuvwxyzabcdef'
>>> klar
'abcdefghijklmnopqrstuvwxyz'
```

Schön! Jetzt haben wir schon das da stehen, was ich dir oben zur Erklärung des Cäsar-Codes aufgeschrieben habe. Konstruieren wir nun einmal Stück für Stück unser Code-Wörterbuch:

```
>>> code = {}
>>> code["a"] = "g"
>>> code["b"] = "h"
>>> code
{'a': 'g', 'b': 'h'}
```

Das wirst du kaum so für jeden Buchstaben hinschreiben wollen. Nimm also einen Index j:

```
>>> j = 2
>>> klar[j]
'c'
>>> geheim[j]
'i'
>>> code[klar[j]] = geheim[j]
>>> code
{'a': 'g', 'c': 'i', 'b': 'h'}
```

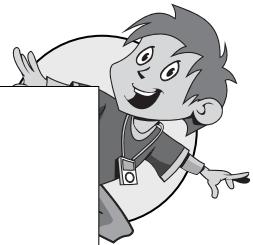
Dies lässt sich doch für alle j von 0 bis 25 automatisieren:

```
>>> code = {}
>>> for j in range(26):
    code[klar[j]] = geheim[j]

>>> code
{'a': 'g', 'c': 'i', 'b': 'h', 'e': 'k', 'd': 'j', 'g': 'm',
 'f': 'l', 'i': 'o', 'h': 'n', 'k': 'q', 'j': 'p', 'm': 's',
 'l': 'r', 'o': 'u', 'n': 't', 'q': 'w', 'p': 'v', 's': 'y',
 'r': 'x', 'u': 'a', 't': 'z', 'w': 'c', 'v': 'b', 'y': 'e',
 'x': 'd', 'z': 'f'}
```

Etwas durcheinander zwar – das kennen wir ja schon von Dictionaries –, aber alles wie gewünscht da. Machen wir daraus eine Funktion! (Wenn dir das lieber ist, kannst du die folgende Funktionsdefinition auch gleich in ein Editor-Fenster der IDLE schreiben.) Sie soll zu jedem Klartextbuchstaben den zugehörigen Cäsar-Code erzeugen:

Verschlüsseln



```
Python Shell
File Edit Shell Debug Options Windows Help
>>> def caesarcode(buchstabe):
    klar = "abcdefghijklmnopqrstuvwxyz"
    i = klar.index(buchstabe)
    geheim = klar[i:] + klar[:i]
    code={}
    for j in range(len(klar)):
        code[klar[j]] = geheim[j]
    return code

>>> caesarcode("j")
{'a': 'j', 'c': 'l', 'b': 'k', 'e': 'n', 'd': 'm', 'g': 'p', 'i':
'f', 'o': 'r', 'h': 'q', 'k': 't', 'j': 's', 'm': 'v', 'l':
'u', 'o': 'x', 'n': 'w', 'q': 'z', 'p': 'y', 's': 'b', 'r':
'a', 'u': 'd', 't': 'c', 'w': 'f', 'v': 'e', 'y': 'h', 'x': 'g',
'z': 'i'}
>>> cc = caesarcode("g")
>>> cc
{'a': 'g', 'c': 'i', 'b': 'h', 'e': 'k', 'd': 'j', 'g': 'm', 'i':
'l', 'o': 'o', 'h': 'n', 'k': 'q', 'j': 'p', 'm': 's', 'l':
'r', 'o': 'u', 'n': 't', 'q': 'w', 'p': 'v', 's': 'y', 'r':
'x', 'u': 'a', 't': 'z', 'w': 'c', 'v': 'b', 'y': 'e', 'x': 'd',
'z': 'f'}
>>> |
Ln: 59 Col: 4
```

Die Funktionsdefinition von `caesarcode()` und ihre Anwendung.

Als Nächstes stellt sich die Aufgabe, das so erstellte Code-Dictionary zu nutzen, um einen Text zu verschlüsseln. Weisen wir zunächst einmal einen kurzen Text zu Testzwecken einem Namen zu. Ich empfehle dir hier, einen englischen Text zu nehmen, damit auch sicher keine Probleme mit Umlauten auftreten. (Unser Alphabet klar hat ja keine Umlaute, auch kein ß.) Wenn du dann später ein Programm zum Codieren schreibst, kannst du ja diese Buchstaben auch ins Klartextalphabet aufnehmen. Dann kommt dir zugute, dass ich in der Funktionsdefinition von `caesarcode` an Stelle von `26 len(klar)` verwendet habe. (Ist dir das aufgefallen? Hast du dich gefragt, warum ich das mache?)

Wenn du, so wie ich, nicht besonders gerne tippst, schreibt dir die folgende import-Anweisung einen englischen Text in die IDLE:

```
import this
```

Aus diesem kannst du dir ein Stück (mit Kopieren und Einfügen) entnehmen, zum Beispiel:

```
>>> testtext = """Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated."""

```

Den geheimtext setzen wir uns jetzt Buchstabe für Buchstabe zusammen. Das heißt, wir setzen für jeden Buchstaben von `testtext` den entsprechenden Codebuchstaben an einen anfänglich leeren geheimtext:

```
>>> cc = caesarcode("g")
>>> geheimtext = ""
```

12



```
>>> for b in testtext:  
    geheimtext = geheimtext + cc[b]
```

Traceback (most recent call last):

File "<pyshell#63>", line 2, in ?

 geheimtext = geheimtext + cc[b]

KeyError: B

Hmmm! B ist, als Großbuchstabe, tatsächlich kein Schlüssel in unserem Code-Wörterbuch cc. Wir sollten den testtext zunächst in Kleinbuchstaben verwandeln:

```
>>> testtext = testtext.lower()  
>>> geheimtext = "" # wir beginnen von neuem!  
>>> for b in testtext:  
    geheimtext = geheimtext + cc[b]
```

Traceback (most recent call last):

File "<pyshell#66>", line 2, in ?

 geheimtext = geheimtext + cc[b]

KeyError: ' '

Oh, ich habe übersehen, dass der Klartext auch Leerzeichen und andere »Nichtbuchstaben« enthält. Die kommen auch nicht als Schlüssel vor. Daher entschließe ich mich, vor der Ersetzung abzufragen, ob das Zeichen überhaupt im Code-Wörterbuch vorkommt. Wenn ja, setze ich den zugehörigen Codebuchstaben ein, andernfalls lasse ich das Zeichen unverändert.

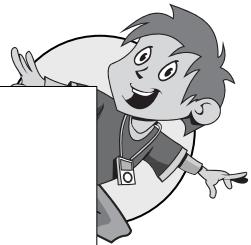
```
>>> geheimtext = ""  
>>> for b in testtext:  
    if b in cc:  
        neu = cc[b]  
    else:  
        neu = b  
    geheimtext = geheimtext + neu
```

```
>>> print(geheimtext)  
hkgazolar oy hkzzkx zngt amre.  
kdvroioz oy hkzzkx zngt osvroioz.  
yosvrk oy hkzzkx zngt iusvrkd.  
iusvrkd oy hkzzkx zngt iusvroigzkj.
```

Das ist O.K.

Nun sind wir also ganz leicht imstande, eine Funktion zu schreiben, die einen Text mit Hilfe eines Code-Wörterbuches in einen Geheimtext umwandelt:

Verschlüsseln



```
>>> def codiere(text, codedict):
    text = text.lower()
    geheim = ""
    for b in text:
        if b in codedict:
            neu = codedict[b]
        else:
            neu = b
        geheim = geheim + neu
    return geheim

>>> geheimtext = codiere(testtext,caesarcode("g"))
>>> print(geheimtext)
hkgazolar oy hkzzkx zngt amre.
kdvroioz oy hkzzkx zngt osvroioz.
yosvrk oy hkzzkx zngt iusvrkd.
iusvrkd oy hkzzkx zngt iusvroigzkj.
```

Natürlich wäre es nun fein, diesen geheimtext auch wieder entschlüsseln zu können. Dazu brauchen wir aber das »umgekehrte« Dictionary. Das kannst du jedoch leicht ermitteln, indem du im ursprünglichen Dictionary Schlüssel und Werte vertauschst: Wenn "a" mit "g" verschlüsselt wird, muss beim Entschlüsseln "g" wieder durch "a" ersetzt werden.

➤ Mach mit!

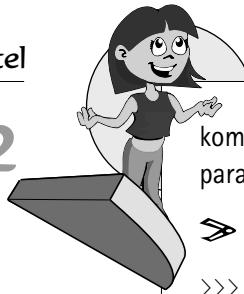
```
>>> cc["a"]
'g'
>>> ("a", "g") in cc.items()
True
>>> dc = {}
>>> dc["g"] = "a"
>>> dc
{'g': 'a'}
>>> dc[cc["b"]] = "b"
>>> dc
{'h': 'b', 'g': 'a'}
```

➤ Schreibe nun eine Schleife, die das Dictionary dc für die Decodierung aus cc.items() erzeugt. Verwende dieses dann, um mit der Funktion codiere() (!) den geheimtext zu entschlüsseln.

Ein Lösung dieser Aufgabe findest du in caesar01.py im Verzeichnis py4kids\kap12 .

Aber jetzt pass auf, weil das ist interessant. Oft denkst du dir ein Lösung für ein kleines Problem durch, verstehst alles bestens, aber irgendwann

12



kommst du dann drauf, dass Python dafür eine noch elegantere Lösung parat hat. So ist das auch mit dem Umkehren von Dictionaries:

⇒ Mach mit:

```
>>> cc = {"a": "g", "b": "h", "c": "i", "d": "j"}  
>>> dc = {wert : schluessel for (schluessel, wert) in  
cc.items()}  
>>> dc  
{'i': 'c', 'h': 'b', 'j': 'd', 'g': 'a'}
```

Dictionary Comprehension nennt sich das. Ist ganz so gebaut wie List Comprehensions. In Python ist es nämlich so wie im richtigen Leben: Du lernst nie aus.

Dateien

Computer-Praktiker, die mit Python arbeiten, müssen verschiedenartigste Textdateien bearbeiten. Bearbeitet werden können diese nur, wenn sie zuerst eingelesen werden, zum Beispiel von der Festplatte. Nach der Bearbeitung müssen sie auch wohin. Wenn sie nicht gerade übers Internet verschickt werden, werden sie wahrscheinlich wieder auf einer Festplatte gespeichert.

Das kannst du jetzt auch brauchen – wenn auch nur spielerisch. Vielleicht willst du einen Text, der in einer Datei gespeichert ist, verschlüsseln und in verschlüsselter Form speichern? Daher möchte ich dir jetzt zeigen, wie du mit Python Textdateien liest und schreibst.

Als Einleitung dazu eine Bemerkung über Dateinamen unter Windows, wo der so genannte »backslash« Verzeichnisnamen voneinander trennt. (Unter Linux und Mac OS X gibt es diese »backslashes« nicht. Die gewöhnlichen »slashes« in den Dateinamen dort machen kein Problem.)

Ich habe eine Datei `probetext.txt` für dich im Verzeichnis `C:\py4kids\kap12` vorbereitet. Der volle Dateiname dieser Datei lautet daher `C:\py4kids\kap12\probetext.txt`.

Dateinamen werden in Python naheliegenderweise mit Strings dargestellt. Für Strings, die Dateinamen bezeichnen, kann man in Python aber auch unter Windows gewöhnliche »slashes« verwenden. Python wird diese hinter den Kulissen bei Bedarf automatisch in »backslashes« umwandeln. Das ist sehr praktisch, . Erstens, weil man so nicht darauf Bedacht nehmen muss, ob man unter Windows, Linux oder auf einem Mac programmiert. Und



zweitens, weil man so Schwierigkeiten vermeidet, die daher röhren, dass der »backslash« \ für Python-Strings das Escape-Zeichen ist (Denke an \n).

Um mit einer Datei zu arbeiten, musst du zunächst ein Dateiobjekt erzeugen. Das geschieht mit der eingebauten Funktion open(). Dabei musst du unterscheiden, ob du die Datei lesen oder beschreiben willst.

Lesen von Dateien

Zunächst muss klar sein, wie der Dateiname lautet. Wir möchten die Datei C:/py4kids/kap12/probetext.txt lesen, etwas verändern und als C:/py4kids/kap12/probetext.caes wieder schreiben.

Wir weisen zunächst den »Stamm« des Dateinamens dem Namen dateiname zu. dateiname1 soll auf denselben String mit der Endung ".txt" verweisen.

```
>>> dateiname = "C:/py4kids/kap12/probetext"  
>>> dateiname1 = dateiname + ".txt"
```

Nun erzeugen wir ein Dateiobjekt zum Lesen der Datei:

```
>>> datei = open(dateiname1, "r")
```

Voraussetzung dafür, dass dies gelingt, ist, dass die Datei tatsächlich existiert. Das zweite Argument im Aufruf von open(), "r", steht für »read« und legt fest, dass die Datei mit dem Modus »Lesen« geöffnet wird. Das Objekt datei hat (unter anderem) die Methode read(), die den gesamten Dateiinhalt in einen String einliest:

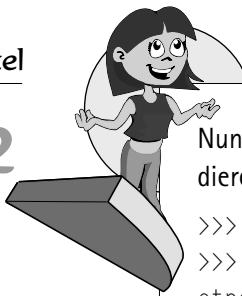
```
>>> text = datei.read()  
>>> datei.close()
```

Ein Dateiobjekt, das nicht mehr gebraucht wird, soll mit der Methode close() geschlossen werden. Es steht dann nicht mehr zum Lesen oder Schreiben zur Verfügung. Aber den Wert von text können wir ausgeben:

```
>>> print(text)  
Dies  
ist  
eine ganz  
kleine Textdatei!
```

```
Lies sie,  
schreib sie,  
  
etc. etc.  
Ciao!  
g1
```

12



Nun steht dieser Text zur Bearbeitung zur Verfügung. Wir wollen ihn codieren:

```
>>> ctext = codiere(text.lower(), caesarcod("l"))
>>> print(ctext)
otpd
tde
ptyp rlyk
vwptyp epieolept!

wtpd dtp,
dnscptm dtp,

pen. pen.
ntlz!
rw
```

Muster 15: Einlesen des Inhalts einer Textdatei

```
datei = open( dateiname, "r")
text = datei.read()
datei.close()
```

Dateiobjekt *datei* wird erzeugt
und zum Lesen geöffnet

Inhalt der *datei* wird als String
dem Namen *text* zugewiesen

Dateiobjekt wird geschlossen

Anschließend kann *text* verarbeitet werden

Aber wenn doch etwas schief läuft?

Weil zum Beispiel die Datei nicht vorhanden ist, die geöffnet werden soll? Für solche Fälle, wo etwas schief gehen könnte, stellt Python eine besondere Anweisung bereit, die `try ... except` Anweisung. Quasi nach dem Muster »Probiere es ... andernfalls ...«:

➤ Mach mit:

```
>>> dateiname = "C:/py4kids/kap12/probe.txt"
>>> datei = open(dateiname, "r")
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    datei = open(dateiname, "r")
IOError: [Errno 2] No such file or directory: 'C:/py4kids/kap12/probe.txt'
```



Natürlich! Du weißt ja, dass die Datei probe.txt nicht existiert. Wir wollen aber nun die Fehler-Meldung mit `try ... except ...` abfangen:

```
>>> try:  
    datei = open(dateiname, "r")  
    text = datei.read()  
    datei.close()  
except:  
    print("{0}: Lesen misslungen!".format(dateiname))  
    text = ""
```

C:/py4kids/kap12/probe.txt: Lesen misslungen!

```
>>> text  
''
```

Wie du siehst, hat diese zusammengesetzte Anweisung einen `try`-Block und einen `except`-Block. Zuerst wird versucht, die Anweisungen im `try`-Block auszuführen. Wenn sie fehlerfrei ablaufen, wird der `except`-Block nicht ausgeführt. Alles läuft so, wie in der ersten, einfacheren Fassung ohne `try.... except....`

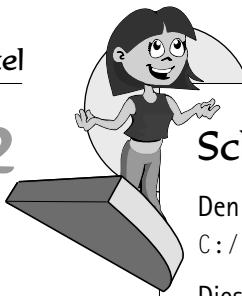
Wenn aber im `try`-Block ein Fehler auftritt, dann wird nun keine Fehlermeldung ausgegeben, sondern der `except`-Block ausgeführt. In unserem Fall wird mittels `print()` eine Meldung über das Problem ausgegeben und danach `text` auf den Leerstring gesetzt. Dann haben wir zwar nichts zu codieren, aber wenigstens ist das Programm nicht abgestürzt!

Dies ist nur ein sehr einfaches Beispiel für das Abfangen von Fehlern. Für höhere Ansprüche gibt es in Python eine ganze Reihe von Varianten der `try`-Anweisung, mit denen man sehr gezielte Fehlerbehandlung programmieren kann. Eine genauere Beschreibung findest du in der Python-Dokumentation: »The Python Tutorial«, Abschnitt 8.3 *Handling Exceptions*.

Muster 16: Fehlerbehandlung mit `try ... except ...`

```
try:  
    Anweisung_T1  
    Anweisung_T2 } } try-Block  
    ...  
except:  
    Anweisung_E1  
    Anweisung_E2 } } except-Block, wird nur im  
    ... Fall eines Fehlers ausgeführt
```

12



Schreiben von Dateien

Den wollen wir nun wieder speichern, und zwar in der Datei C:/py4kids/kap12/probetext.caes.

Diesmal müssen wir der Funktion `open()` als zweites Argument ein "w" für »write« übergeben. Damit zeigen wir an, dass wir die Datei beschreiben wollen.

```
>>> dateiname2 = dateiname + ".caes"
>>> datei = open(dateiname2, "w")
```

Das Objekt `datei` hat eine Methode `write()`, der man einfach den String als Argument übergibt, der in die Datei zu schreiben ist.

```
>>> datei.write(ctext)
80
>>> datei.close()
```

Wenn nichts weiter hineingeschrieben werden soll, kann und soll das Dateiobjekt mit der Methode `close()` geschlossen werden.

Fragst du dich, warum `datei.write()` den Wert 80 zurück gegeben hat? Weil bislang 80 Zeichen in die Datei geschrieben wurden:

```
>>> len(ctext)
80
```

Wir halten auch das wieder in einer Musterbeschreibung fest. (Hinweis: es können beliebig viele Aufrufe der Methode `write()` aufeinander folgen.)

Muster 17: Schreiben von Strings in eine Textdatei

```
datei = open(dateiname, "w")
```

Dateiobjekt `datei` wird erzeugt und zum Schreiben geöffnet

```
datei.write(text1)
```

Der String `text1` wird in die `datei` geschrieben

```
datei.write(text2)
```

...

```
datei.close()
```

... weitere Schreibanweisungen sind möglich

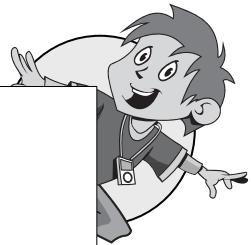
Dateiobjekt wird geschlossen

Da du mit der IDLE beliebige Textdateien öffnen kannst, kannst du jetzt `probestext.caes` in ein IDLE-Editor-Fenster einladen und betrachten. (Im DATEI-ÖFFNEN-Dialogfenster Dateityp auf `*.*` einstellen! Wegen der Dateiendung `.caes`.)

Wenn du all das einfach in eine Funktion zusammenfasst, erhältst du eine kleine nützliche Werkzeug-Funktion:

Zusammenfassung

```
>>> def codiere_datei(dateiname, char):
    f = open(dateiname+".txt")
    text = f.read()
    f.close()
    geheim = codiere(text, caesarcode(char))
    g = open(dateiname+".caes", "w")
    g.write(geheim)
    g.close()
```



Mit dem bisher gelernten sollte es dir nicht schwer fallen auch eine Funktion `decodiere_datei(dateiname, char)` zu schreiben.

Du findest auf der CD im Verzeichnis `py4kids/kap12`, die mit dem Buchstaben q cäsar-codierte Datei `nocheintext.caes`. Decodiere sie mit `decodiere_datei()`.

A screenshot of a Windows Notepad window titled "nocheintext.caes - C:\py4kids\kap12\nocheintext.caes". The window contains the following encoded text:
`|tyui yij uydu wqdp qdtuhu abuydu junjtqjuy.
whqjkqbqjyed, tqii tk tysx tuh ckuxu kdjuhpewud
xqij, iyu pk tuaetyuhud.
tk ryij uyd usxj dukwyuhwuh jof!`

Mit "q" codiert. Wie lautet der Klartext?

Eine genauere Beschreibung der Funktion `open()` findest du in der Python-Dokumentation »Library Reference/The Python Standard Library«, Abschnitt 2: Eingebaute Funktionen.

Methoden von Dateiobjekten werden in Abschnitt 5.9, File Objects, beschrieben.

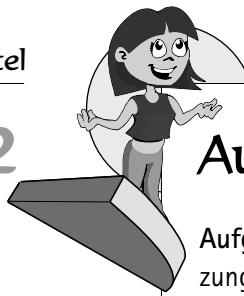


Zusammenfassung

Dieses Kapitel hat eine Vielzahl von Neuigkeiten gebracht:

- ❖ Dictionaries sind ein veränderbarer eingebauter Datentyp in Python. Sie speichern Schlüssel-Wert-Paare. Die Schlüssel müssen unveränderliche Objekte sein.
- ❖ Um zu einem Schlüssel den Wert zu erhalten, kann man den Schlüssel als Index benutzen (wie Zahlen bei Listen).
- ❖ Datei-Objekte werden mit der Funktion `open()` erzeugt. Sie haben Methoden zum Lesen und Schreiben von Strings.
- ❖ Schleifen können mit der Anweisung `break` abgebrochen werden.

12



Aufgaben ...

Aufgabe 1: Erstelle ein Skript – mit dem »Material« der interaktiven Sitzung zu diesem Thema –, das Funktionen enthält, mit denen du Dateien mit einem Cäsar-Code wahlweise verschlüsseln oder entschlüsseln kann.

Aufgabe 2: Jeder Caesar-Code hat die interessante Eigenschaft, dass seine Umkehrung, die zum Decodieren gebraucht wird, wieder ein Caesar-Code ist, aber für einen anderen Buchstaben. Wenn du herausfinden kannst, wie dieser Buchstabe für die Decodierung aus dem für die Codierung ermittelt werden kann, kommst du mit nur einer Codierungs-Funktion aus. Nütze dies, um deine Lösung von Aufgabe 1 zu vereinfachen.

Aufgabe 3: Python hat eine eingebaute Funktion `zip()`, die aus zwei (oder mehr) Sequenzen (oder gewissen anderen Wertevorräten) einen *Vorrat* von Tupeln aus den jeweils ersten, jeweils zweiten Elementen usw. macht. Aus diesem kann dann mittels `list()` eine *Liste* von Tupeln gemacht werden – und im Falle von Zweiertupeln sogar mittels `dict()` ein *Dictionary*:

➤ Mach mit:

```
>>> zip("abcd", "pqrs")
<zip object at 0x01362148>
>>> list(zip("abcd", "pqrs"))
[('a', 'p'), ('b', 'q'), ('c', 'r'), ('d', 's')]
>>> dict(zip("abcd", "pqrs"))
{'a': 'p', 'c': 'r', 'b': 'q', 'd': 's'}
```

Nütze dies um eine kürzere Version von `caesarcode(buchstabe)` zu erstellen.

... und einige Fragen

1. Wenn `d` ein Dictionary ist: Was ist der Unterschied zwischen den booleschen Ausdrücken `a` in `d` und `a` in `d.keys()`?
2. Kann das Tupel (1, 2, 3) als Schlüssel in einem Dictionary verwendet werden?
3. Kann die Liste [1, 2, 3] als Schlüssel in einem Dictionary verwendet werden?
4. Was ist der Unterschied zwischen `f = open("hi.txt")` und `g = open("hi.txt", "w")`?

13

Ereignigesteuerte Programme

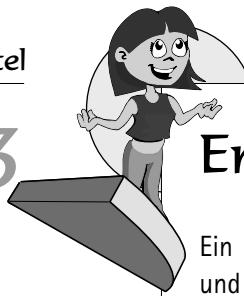


In diesem Kapitel kommen wir zu einem ganz neuen Thema: ereignigesteuerte Programme. Der Ablauf solcher Programme hängt von den Ereignissen ab, die der Benutzer durch seine Aktionen – Tasten drücken, Maus klicken usw. – auslöst.

In diesem Kapitel wirst du lernen ...

- ◎ dass du mit dem Modul `turtle` ereignigesteuerte Programme erstellen kannst.
- ◎ auf welche Ereignisse das Grafik-Fenster und auch die Turtles selber reagieren können.
- ◎ wie du die Reaktionen der Turtle oder auch des Programms als Ganzes auf Ereignisse festlegen kannst.
- ◎ Wie du aus Turtles Buttons zur Programmsteuerung machen kannst.
- ◎ wie du Animationen mit Timer-Ereignissen steuern kannst.
- ◎ wie du das Modul `datetime` dazu nutzen kannst Datum und Uhrzeit in deinen Programme zu verwenden.

13



Ereignisse

Ein Programm, das auf Mausklicks (eventuell auch auf Mausbewegungen) und auf Tastatureingaben mit Aktionen reagiert, nennt man ein *ereignisgesteuertes Programm* (engl.: *event driven program*). Ein Mausklick, ein Tastendruck auf der Tastatur ist ein Ereignis – auf Englisch: ein *event*. Ereignisse zu registrieren, ist zunächst Aufgabe des Betriebssystems deines Computers. GUI-Toolkits sind so programmiert, dass sie vom Betriebssystem Ereignis-Objekte zugestellt bekommen, mit Informationen über die stattgefundenen Ereignisse. Das gilt auch für das Grafik-Toolkit Tkinter, auf dem das Modul `turtle` aufbaut.

Aufgrund dieser vom Betriebssystem zugestellten Informationen kann daher das Turtle-Grafik-Fenster auf Ereignisse reagieren, die der Benutzer ausgelöst hat. Nur müssen wir dem Grafik-Fenster sagen, *mit welchen Aktionen* es auf Ereignisse, z. B. Mausklicks, reagieren soll.

Das klingt ja hübsch abstrakt, ruft also dringend nach einem anschaulichen Beispiel.

»screen.onclick(goto)«

Nachdem du eben gelernt hast, mit Objekten zu programmieren, wollen wir diesen Programmierstil in der Folge weiter verwenden. Wir werden wie bisher zunächst das Screen-Objekt erzeugen und mit einem oder mehreren Turtle-Objekten arbeiten.

Wir werden daher von nun an aus dem Modul `turtle` meistens nur die beiden Klassen `Screen` und `Turtle` importieren.

⇒ Mach mit!

```
>>> from turtle import Screen, Turtle  
>>> screen = Screen()  
>>> turtle = Turtle()  
>>>
```

Du hast nun das wohlbekannte Turtle-Grafik-Fenster mit einer Turtle vor dir. Ereignisse, die dieses Fenster registrieren kann, sind zum Beispiel Mausklicks.

⇒ Klicke einige Male mit der linken Maustaste in das Grafik-Fenster.

»screen.onclick(goto)«



Was geschieht? Nichts! Das liegt daran, dass *wir* noch nicht festgelegt haben, was im Falle eines Mausklicks geschehen soll. Das werden wir gleich ändern!

```
>>> screen.onclick(turtle.goto)
```

```
>>>
```

Zunächst scheint diese Anweisung nichts bewirkt zu haben. Doch:

⇒ Klicke einige Male mit der linken Maustaste in das Grafik-Fenster.

Was beobachtest du? Die Turtle bewegt sich nach jedem Klick zu dem Punkt im Grafik-Fenster, wo du hingeklickt hast. Warte nach jedem Klick, bis die Turtle dort ist, bevor du auf eine andere Stelle klickst!

Das ist wohl einigermaßen überraschend und erfordert eine genaue Erklärung. Bevor ich dir die gebe, machen wir aber noch eine kleine interaktive Übung:

```
>>> def fun(x,y):
    print("Click:", x, y)
    turtle.goto(x, y)
    print("Bin schon da:", turtle.pos())

>>> screen.onclick(fun)
```

Das sieht oberflächlich betrachtet so aus, als wollten wir bloß Spaß mit Mausklicks haben. Jedoch:

⇒ Klicke einige Male mit der linken Maustaste in das Grafik-Fenster.

Das bewirkt, dass nach jedem Mausklick die Koordinaten des Punktes, auf den du geklickt hast, im IPI-SHELL-Fenster ausgegeben werden, die Turtle dann dorthin läuft, was auch sogleich gemeldet wird. Wie kommt das zu stande?

Erklärung: `screen.onclick(fun)` ist ein Aufruf der Methode `onclick()` für das Turtle-Grafik-Fenster `screen`. Als Argument haben wir das Funktionsobjekt `fun` eingesetzt.

Die Methode `onclick()` hat zwei Parameter.

❖ Für den ersten Parameter muss als Argument eine Funktion oder eine Methode eines bestimmten Objekts eingesetzt werden – und zwar ein Funktions- oder Methodenobjekt, nicht ein Aufruf! (Erinnere dich: In Python sind Funktionen auch Objekte.) Diese Funktion oder Methode muss selbst auch zwei Parameter haben. Wir hatten im ersten `onclick()`-Aufruf die `turtle`-Methode `turtle.goto` eingesetzt, im zweiten dann `fun`. `fun` hatten wir so definiert, dass es zwei Parameter

13



x und y hat. `Turtle.goto()` ist eine Methode der Klasse `Turtle`, die ebenfalls zwei Parameter hat.

- ❖ Für den zweiten Parameter im `screen.onclick()`-Aufruf ist eine der Zahlen 1, 2 oder 3 einzusetzen. Die Zahlen bezeichnen die linke, die mittlere beziehungsweise die rechte Maustaste. Dabei ist 1 – also die linke Maustaste – als Standardwert vorgesehen. Deswegen konnten wir dieses Argument in unserem Beispiel weglassen.
- ❖ Ein Aufruf von `onclick()` mit `fun` als erstem Argument bewirkt, dass die Ausführung der Funktion `fun()` an Mausklicks mit der angegebenen Maustaste gebunden wird. Genauer: Jedes Mal, wenn der Benutzer in das Turtle-Grafik-Fenster klickt, wird die Funktion `fun()` ausgeführt, wobei die Koordinaten des angeklickten Punktes als Argumente an `fun()` übergeben werden.



Funktionen, die so wie `fun()` gebraucht werden, nennt man *Callback-Funktionen*, oder – weniger gebräuchlich – Rückruf-Funktionen. Sie werden einer anderen Funktionen als Argument übergeben, die dafür sorgt, dass die Rückruf-Funktion unter bestimmten Bedingungen ausgeführt wird – zum Beispiel wenn ein Mausklick auf das Fenster auftritt.

Im ersten Beispiel hatten wir die Callback-Funktion `turtle.goto` der Funktion `onclick()` als Argument übergeben.

➤ Mach weiter mit!

```
>>> turtle.reset()  
>>> turtle.goto(150,100)
```

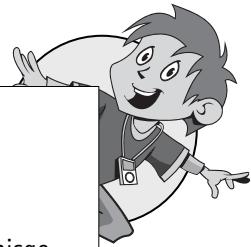
Die `turtle`-Methode `goto()` bewirkt, dass die Turtle zum Punkt mit den Koordinaten (x, y) geht. Und eben das konnten wir mit (linken) Mausklicks erreichen, nachdem wir den Befehl `screen.onclick(turtle.goto)` ausgeführt hatten.

Eine auf diese Weise hergestellte Bindung kannst du mit folgendem Methodenaufruf rückgängig machen:

```
>>> screen.onclick(None)
```

➤ Klicke einige Male mit der linken Maustaste in das Grafik-Fenster.

Was geschieht? Nichts! Okay.



Scribble

Als erste kleine Anwendung wollen wir nun ein ganz einfaches ereignisgesteuertes Malprogramm schreiben, mit dem du kleine Kritzeleien erzeugen kannst: scribble.py.

screen.onclick(turtle.goto) bietet dafür einen guten Ausgangspunkt, denn damit kann man schon geometrische Figuren zeichnen.

- Öffne ein Editor-Fenster und gib als Ausgangsmaterial nach deinem Kopfkommentar folgenden Code ein:

```
from turtle import Screen, Turtle  
  
screen = Screen()  
stift = Turtle()  
  
screen.onclick(stift.goto)
```

- Speichere das Script im Editor unter dem Namen scribble_arbeit.py ab.
➤ Führe das Script aus und erzeuge mit ein paar Mausklicks einen Linienzug.

Wie du siehst, habe ich den Namen der Turtle ihrer Aufgabe angepasst. Ein Zeichenstift muss aber nicht unbedingt die Form einer Schildkröte haben. Daher werden wir das gleich ändern und auch noch einige andere Eigenschaften einstellen:

- Ergänze scribble_arbeit.py im Editorfenster um folgende Anweisungen:

```
from turtle import Screen, Turtle  
  
screen = Screen()  
screen.clear()  
stift = Turtle()  
stift.speed(0)  
stift.shape("circle")  
stift.shapesize(0.4, 0.4, 3)  
stift.pensize(3)  
  
screen.onclick(stift.goto)
```

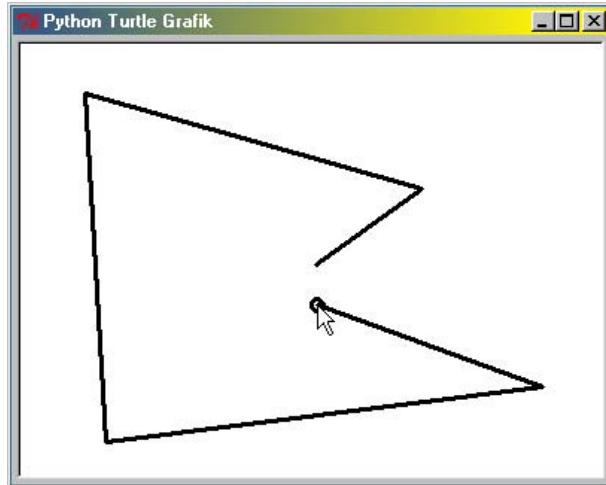
13



Kurze Erklärung:

- ◊ (1) `screen.clear()` versetzt das `screen`-Objekt in einen leeren Zustand. Es löscht alle Zeichnungen und entfernt alle Turtles.
- ◊ (2) "circle" ist eine eingebaute Turtle-Form: ein Kreis. Das passt hier gut, da die Methode `goto()` die Orientierung der Turtle ändert und diese daher keine Rolle spielt.
- ◊ (3) Die `turtle`-Methode `shapesize()` legt fest, wie die Größe der Turtle eingestellt wird. Sie wird mit bis zu drei Zahlenwerten als Argumenten aufgerufen: Die ersten beiden geben an, um welchen Faktor die gewählte Turtle-Form quer zu und längs ihrer Orientierung gestreckt werden soll, die dritte gibt die Strichdicke für die Turtle-Form an. `shapesize()` stellt außerdem die Turtle auf `resizemode("user")` ein, damit ihre Größe nicht wie üblich automatisch aus der Strichdicke ermittelt wird. (Benutze `help(stift.shapesize)` bzw. `help(stift.resizemode)`, um mehr über diese beiden Funktionen zu erfahren.)

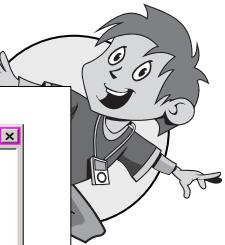
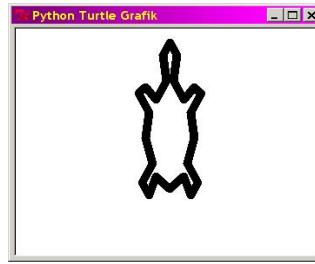
⇒ Speichere eine Kopie des Scripts als `scribble01.py` ab.



- ⇒ Wie üblich bringt es dich vielleicht weiter, `stift.shapesize()` spielerisch im Direktmodus zu untersuchen. Mach mit!

Zeichenstift steuern

```
>>> stift.reset()
>>> stift.shapesize(2)
>>> stift.shapesize(3, 2)
>>> stift.shapesize(2, 4, 5)
>>> stift.shape("turtle")  # arm!
=> Schließe das Grafik-Fenster.
```



Zeichenstift steuern

Der auffälligste Mangel unseres Scripts ist wohl der, dass wir den Zeichenstift nicht anheben können, um die Turtle auch ohne zu zeichnen über die Zeichenfläche zu steuern. Das wollen wir nun beheben.

Eine einfache Lösung: Klicken auf die mittlere Maustaste bewegt die Turtle ohne Zeichnen – hatten wir dafür nicht schon `jump()`? Wir müssen jetzt eine Funktion schreiben, die den `stift` dazu bringt, zu springen. Und diese Funktion an den Klick auf die mittlere Maustaste binden.

=> Füge in `scribble_arbeit.py` gleich unterhalb der `import`-Anweisung folgende Definition für `jump()` ein und am Ende des Programms den `screen.onclick()`-Aufruf für `jump`:

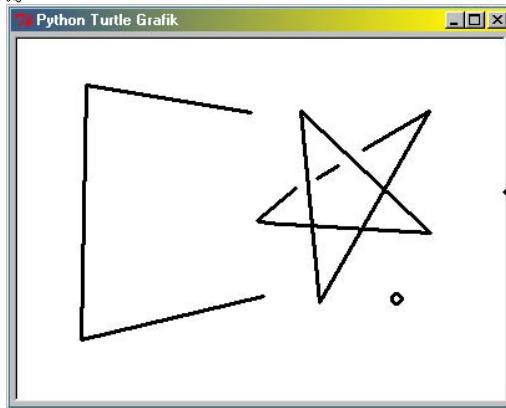
```
from turtle import Screen, Turtle

def jump(x, y):
    stift.penup()
    stift.goto(x, y)
    stift.pendown()

screen = Screen()
...
screen.onclick(stift.goto)
screen.onclick(jump, 2)
```

=> Speichere das Programm und fertige eine kleine Testzeichnung an.

13



Mit Mausklicks können gerade Linien erzeugt werden.

Ganz nett. Aber sicher wäre es nützlich, wenn wir mit unserem Stift nicht nur gerade Strecken, sondern auch krumme Linien zeichnen könnten.

➤ Speichere eine Kopie des Scripts als scribble02.py ab.

Kritzeln

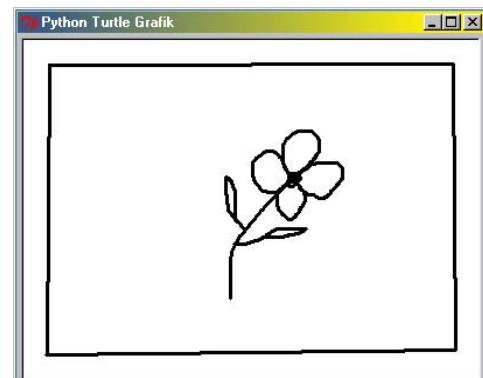
Hier kommt uns sehr gelegen, dass nicht nur das Grafik-Fenster, sondern auch Turtles selbst auf Ereignisse reagieren können: auf Anklicken und auf Ziehen mit gedrückter Maustaste. Letzteres können wir hier gut gebrauchen. Füge Folgendes als letzte Zeile in scribble_arbeit.py ein:

```
...
screen.onclick(jump, 2)
stift.ondrag(stift.goto)
```

Die Turtle-Methode ondrag() bindet nun stift.goto an das Ziehen der Maus. ondrag() verlangt gleichartige Argumente wie screen.goto.

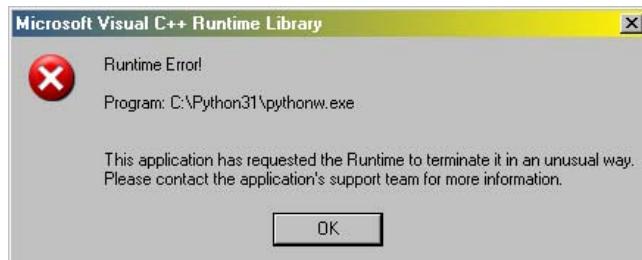
➤ Probiere das aus: Speichere das Script, führe es aus und fertige eine Kritzelei an. (Natürlich können auch gerade Linien und Sprünge dabei sein.)

Vielleicht hast du für deine Grafik recht viel herumkritzelt. Dann kann es passiert sein, dass nicht nur das scribble-Programm, sondern zugleich auch Python



Gefüllte Flächen

abgestürzt ist. Auf meiner Windows-Maschine geschieht das mit einem Laufzeitfehler und der unangenehmen Fehlermeldung:



Dies kannst du durch Beachtung folgender Regel verhindern:

Um `Turtle.ondrag()` sicher verwenden zu können, muss die Rekursionsgrenze für Python vom voreingestellten Standardwert 1000 auf einen hohen Wert, z.B. 20000 gesetzt werden. Dies geschieht am besten, indem du am Anfang des Programms folgende zwei Zeilen einfügst:

```
import sys  
sys.setrecursionlimit(20000)
```



- Füge die beiden Code-Zeilen gleich unter der `from turtle import`-Anweisung ins Script ein.
- Speichere eine Kopie von `scribble_arbeit.py` als `scribble03.py` ab.

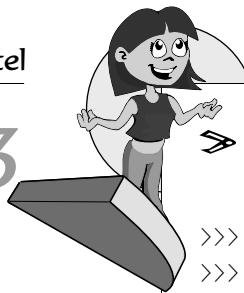
Gefüllte Flächen

Um gezeichnete Flächen füllen zu können, müssen wir dafür sorgen, dass die Turtle-Methoden `begin_fill()` (vor dem Zeichnen des Umrisses) und `end_fill()` (danach) ausgeführt werden können. Vorschlag: Wir binden dies an Mausklicks der rechten Maustaste. Probieren wir den Mechanismus interaktiv aus:

- Führe `scribble_arbeit.py` aus und mach mit:

```
>>> stift.fillcolor("black")  
>>> stift.filling()  
False  
>>> stift.begin_fill()  
>>> stift.filling()  
True
```

13



→ Kritzel nun eine irgendwie geschlossene Linie ins Grafik-Fenster. Dann weiter:

```
>>> stift.end_fill()  
>>> stift.filling()  
False
```

Du siehst, die Turtle hat eine Methode, um festzustellen, in welchem Füllzustand sie ist, nämlich `filling()`. Damit können wir die rechte Maustaste so programmieren, dass sie Füllen einschaltet, wenn es gerade aus ist, und ausschaltet, wenn es eingeschaltet ist. Damit aber auch wir *sehen*, wie der Zustand der Turtle ist, nehmen wir die Füllfarbe einfach weg, wenn die Turtle nicht füllt. Eine Funktion, die das tut, könnte so aussehen:

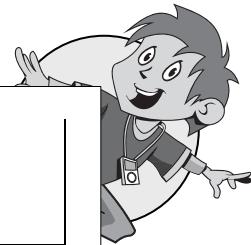
```
def fuellenumschalten():  
    if stift.filling():  
        stift.end_fill()  
        stift.fillcolor("")  
    else:  
        stift.begin_fill()  
        stift.fillcolor("black")
```

Du erinnerst dich aber sicherlich daran, dass Funktionen/Methoden, die an das Klick-Ereignis gebunden werden sollen, zwei Eingabeparameter haben müssen. Deshalb müssen wir ihnen zwei Parameter verpassen, obwohl sie in diesem besonderen Fall gar nicht gebraucht werden. Sie sind also gewissermaßen Schein-Parameter, auf Englisch: *Dummy*-Parameter. Um dies auszudrücken, nennen wir sie `xdummy` und `ydummy`.

Füge den Code für die entsprechend ergänzte Funktion `fuellenumschalten()` nach dem Code für `jump()` in `scribble_arbeit.py` ein und den entsprechenden Aufruf der `onclick()`-Methode als vorletzte Zeile im Programm:

```
def fuellenumschalten(xdummy, ydummy):  
    if stift.filling():  
        stift.end_fill()  
        stift.fillcolor("")  
    else:  
        stift.begin_fill()  
        stift.fillcolor("black")  
  
...
```

»undo()« und Tastatur-Ereignisse



```
screen.onclick(jump, 2)
screen.onclick(fuellenumschalten, 3)
stift.ondrag(stift.goto)
```

- Teste scribble_arbeit.py und speichere eine Kopie dieser Fassung als scribble04.py ab.

»undo()« und Tastatur-Ereignisse

Wenn du mit Scribble nun schon etwas experimentiert hast, dann wirst du sicherlich ab und zu das Bedürfnis verspürt haben, eine gezeichnete Linie wieder rückgängig zu machen. Oder auch den Wunsch gehabt haben, die vorhandene Zeichnung zu löschen und eine neue zu beginnen.

Für beides gibt es einfache Turtle-Methoden: `Turtle.undo()` und `Turtle.clear()`. Aber an welche Ereignisse wollen wir die Ausführung dieser Methode nun binden?

Um Computerprogramme zu steuern, hast du natürlich nicht nur Mausereignisse zur Verfügung, sondern auch Tastaturereignisse: Drücken auf eine bestimmte Taste. Das Modul `turtle` bietet auch die Möglichkeit, Funktionen an Tastaturereignisse zu binden, und zwar mit Hilfe der `screen`-Methode `onkeypress()`.

`onkeypress()` hat zwei Parameter. Für den ersten ist als Argument eine Funktion *ohne Parameter* einzusetzen. Die Methoden `stift.undo()` und `stift.clear()` sind dafür perfekt geeignet, weil sie keine Parameter haben. Für den zweiten Parameter von `onkeypress()` ist ein Tastatursymbol einzusetzen. Für gewöhnliche Buchstabentasten ist das Tastatursymbol ein String mit diesem Buchstaben.

- Starte IPI-TURTLEGRAFIK neu und mach mit!

```
>>> from turtle import Screen, Turtle
>>> screen = Screen()
>>> t = Turtle()
>>> t.pensize(3)
>>> def links():
...     t.left(60)

>>> def rechts():
...     t.right(60)

>>>
```

13



```
>>> def vor():
    t.forward(50)

>>> screen.onkeypress(links, "l")
>>> screen.onkeypress(rechts, "r")
>>> screen.onkeypress(vor, "v")
```

Wenn du nun versuchst, die Turtle mit den Tasten **l**, **r** und **v** zu steuern, wirst du bemerken, dass das noch nicht funktioniert, weil dabei nur eine Folge dieser Buchstaben in die IPI-SHELL geschrieben wird.

Auf deinem Bildschirm sind jetzt zwei Fenster offen: die IPI-SHELL und ein Turtle-Grafik-Fenster. Und dazu vielleicht auch noch ein Webbrowser oder ein Chat-Programm, ein Spiel usw. Klar, dass in solch einem Fall irgendwie festgelegt werden muss, welchem dieser Fenster nun die Tastaturereignisse gelten. Eines der Fenster muss – wie man sagt – den *Fokus* haben. Im Augenblick ist das (mindestens bei mir) die IPI-Shell. Tastendrücke schreiben daher dorthin.

Die Steuerung funktioniert erst dann für das Grafik-Fenster, wenn der Fokus (des Betriebssystems, das die Ereignisse zuerst entgegennimmt) darauf gelegt wird. Das erreichst du mit der `screen-Methode listen()`. (Hörte!)

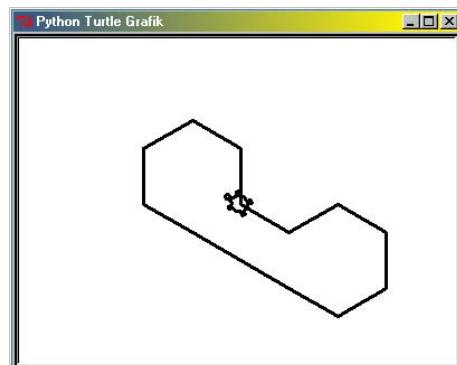
➤ Mach weiter mit!

```
>>> screen.listen()
```

Das Turtle-Grafik-Zeichenfenster erscheint nun hervorgehoben (dunkel umrandet) und Tastaturereignisse werden für dieses Fenster entgegengenommen.

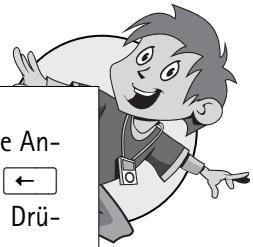
➤ Drücke folgende Tasten: **v**, **l**, **v**, **l**, **v**, **l**, **v**, **l**, **v**, **l**, **v**, **v**, **v**, **v**, **l**, **v**, **l**, **v**, **l**, **v**, **l**, **v**, **r**, **v**.

Die Schildkröte gehorcht
der Tastensteuerung.



Das funktioniert prächtig.

Strichdicke einstellen



Wir sind nun mutig und ergänzen `scribble_arbeit.py` um folgende Anweisungen, damit Zeichen-Aktionen durch Drücken der Rücktaste ← rückgängig gemacht werden können und die ganze Zeichnung durch Drücken der [Leertaste] gelöscht werden kann.

```
stift.ondrag(stift.goto)

screen.onkeypress(stift.undo, "BackSpace")
screen.onkeypress(stift.clear, "space")
screen.listen()
```

- Nach Einfügen dieser drei Zeilen teste das Programm erneut, indem du nicht so gelungene Teile einer Test-Zeichnung mit `undo()` rückgängig machst, weiterzeichnest, vielleicht wieder etwas rückgängig machst usw.
- Wenn eine Zeichnung ganz missraten ist, lösche sie einfach mit der Leertaste und beginne von Neuem.
- Wenn alles ordnungsgemäß funktioniert, dann speichere eine Kopie von `scribble-arbeit.py` als `scribble05.py` ab.

Eine schöne Zeichnung willst du vielleicht auch als Bild abspeichern. Wenn du ein Screenshot-Programm hast, kannst du das damit rasch erledigen.

Wie du eine Python-Funktion, die das für dich macht, in das Scribble-Programm einbauen kannst, erfährst du auf der Buch-Website <http://python4kids.net>. Weil das für jedes Betriebssystem etwas anders geht, ist hier nicht genug Platz für die nötigen Erklärungen.

Strichdicke einstellen

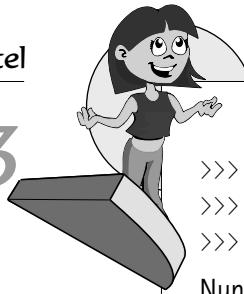
Wir könnten unsere Zeichnungen mit Scribble noch abwechslungsreicher gestalten, wenn wir verschiedene Strichdicken zur Verfügung hätten. Um es einfach zu machen, schlage ich vor, dass Strichdicken von 1 bis 9 zur Verfügung gestellt und durch Tippen auf die Tasten mit den Zahlen 1 bis 9 eingestellt werden können.

Was denkst du, wie viele Programmzeilen nötig sind, um dieses »Feature« hinzuzufügen?

- Führe `scribble-arbeit.py` aus und mach mit (oder auch nicht)!

```
>>> def pensize1():
    stift.pensize(1)
```

13



```
>>> screen.onkeypress(pensize1, "1")
>>> screen.listen()
>>>
```

Nun kannst du durch Drücken der **[1]**-Taste die Strichdicke auf 1 stellen.

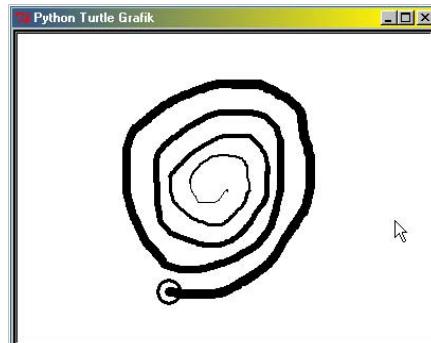
Und jetzt das Ganze noch 8 Mal? Nicht mit Python! Ich stelle dir hier eine ganz Python-typische Lösung vor und werde sie anschließend erklären.

➤ Füge folgende fünf Code-Zeilen in scribble_arbeit.py am Ende des Scripts vor der screen.listen() Anweisung ein:

```
screen.onkeypress(stift.clear, "space")
for c in "123456789":
    def setpensize(ps=int(c)):
        stift.pensize(ps)
        stift.shapesize(0.2 + ps/10)
    screen.onkeypress(setpensize, c)
screen.listen()
```

➤ Erledigt! Und noch dazu mit einem Extra: Die Strichdicke wird durch die Größe des Kreises angezeigt, der den Stift markiert. Nun kannst du das Strichdicken-Feature von Scribble testen und eine Grafik wie z.B. die untenstehende erzeugen.

Der Stift von Scribble zeichnet mit unterschiedlichen Strichstärken.



Nun bin ich dir noch die Erklärung schuldig. Oder hast du vielleicht selbst schon darüber nachgedacht und die Sache verstanden? Darauf könntest du stolz sein!

Die Lösung wird durch eine `for`-Schleife erreicht. Sehen wir uns zunächst den Schleifenkopf an: Die Schleife wird für jedes Zeichen aus der Zeichenkette "123456789" einmal durchlaufen, also für `c` gleich "1", `c` gleich "2" usw., insgesamt neun Mal.

Strichdicke einstellen

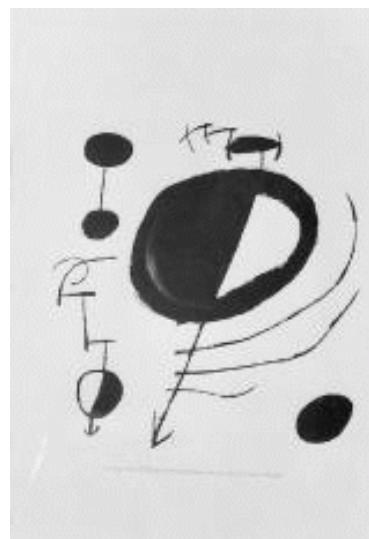


Und der Schleifenkörper? Er besteht aus zwei Anweisungen! Die erste ist eine Funktionsdefinition. Sie erzeugt ein Funktionsobjekt namens `setpensize`. Die zweite ist ein Aufruf der `screen.onkeypress()`-Methode. Sie bindet das eben erzeugte Funktionsobjekt an den Tastendruck auf die `c`-Taste, die dem aktuellen Schleifendurchgang entspricht. Somit werden insgesamt neun Funktionen erzeugt und an das Drücken der entsprechenden Tasten gebunden.

Sehen wir uns nun noch die Funktionsdefinition der Funktion `setpensize()` an. Die Funktion hat einen Parameter `ps`. Aber dieser wird sofort – das heißt bei der Definition der Funktion – mit einem Standardwert belegt: `int(c)`. Das ist die ganze Zahl, die dem Zeichen 'c' entspricht. Bei jedem Schleifendurchgang bekommt somit `ps` einen anderen Wert. Jedes Mal, wenn die Taste `c` gedrückt wird, wird die Funktion `setpensize()` ausgeführt: Sie setzt die Strichdicke von `stift` auf den Wert von `ps` und stellt danach noch die Kreis-Form umso größer ein, je größer die Strichdicke ist.

Der Clou an der Sache ist, dass `setpensize()` nun wie eine Funktion ohne Parameter verwendet werden kann: Man kann sie aufrufen, ohne ein Argument einzusetzen. Und genau eine solche verlangt `screen.onkeypress()`. Aber sie kennt trotzdem »ihre« Strichdicke.

➤ Üben wir nun ein wenig, mit Scribble zu arbeiten. Erzeuge beispielsweise eine Grafik wie nebenstehende. Oder lass deine eigene Fantasie spielen.



Diese Grafik stammt von dem bekannten katalanischen Maler Joan Miró. Sie ist auch im Original schwarz-weiß.

➤ Wenn alles fehlerfrei funktioniert, speichere eine Kopie von `scribble_arbeit.py` als `scribble06.py` ab.

13



Farben

Jetzt wollen wir noch eine Möglichkeit einbauen, die Farbe des Zeichenstifts zu ändern, damit wir auch bunte Kritzeleien erzeugen können.

Ich schlage vor, nach dem Vorbild größerer Anwendungsprogramme auf der linken Seite des Fensters eine Art »Werkzeugeiste« unterzubringen: einige Farb-Buttons zur Wahl der Farbe. Natürlich bieten sich dafür anklickbare Turtles mit quadratischer Form an. Wenn du sie mit der linken Maustaste anklickst, soll das die Zeichenfarbe des Stifts ändern, und wenn du sie mit der rechten Maustaste anklickst, die Füllfarbe.

Für diese Knöpfe brauchen wir Platz und unser Scribble-Programm geht jetzt auch schon in Richtung richtige Anwendung. Daher wollen wir eine größere Zeichenfläche haben:

```
screen.clear()
screen.setup(840, 600)
stift = Turtle()
```

Erste Idee für die Knöpfe:

```
def farbButton(farbe, y):
    t = Turtle(visible=False)
    t.shape("square")
    t.speed(0)
    t.pu(); t.goto(-380,y)
    t.fillcolor(farbe)
    def setpencolor(xdummy, ydummy):
        stift.pencolor(farbe)
    def setfillcolor(xdummy, ydummy):
        stift.fillcolor(farbe)
    t.onclick(setpencolor)
    t.onclick(setfillcolor, 3)
    t.showturtle()
```

Die Funktion `farbButton()` platziert einen »Button« der eben beschriebenen Art links, bei $x = -380$, in der Höhe y , die als Argument übergeben wird. Sie verwendet eine ähnliche Technik wie `setpensize()`, das ich weiter vorn erklärt habe.

➤ Füge den Code dieser Funktion unterhalb der Funktion `fuellenumschalten()` in `scribble_arbeit.py` ein. Führe das Programm aus und probiere:

Farben

```
>>> farbButton("red", 120)
>>> farbButton("blue", 80)
>>>
```

Super, die Knöpfe werden richtig platziert.

- Klicke z.B. auf den blauen Knopf, um die Stiftfarbe auf Blau einzustellen.

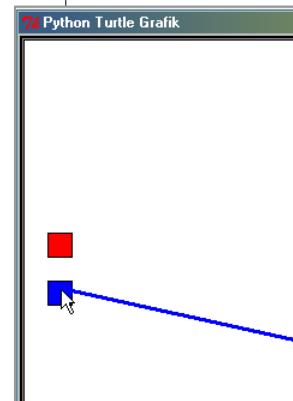
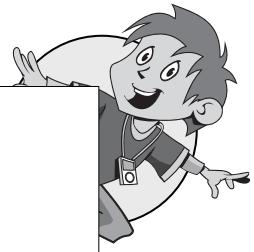
Ups! Die Stiftfarbe ist nun wohl Blau, aber leider hat die Turtle einen Strich bis zum blauen Knopf gezeichnet. Offenbar löst der Mausklick nicht nur ein Klick-Ereignis auf die Turtle, sondern auch auf die Zeichenfläche aus. Aber so war das nicht geplant. Zum Glück lässt sich das reparieren: Wir müssen den Werkzeug-Leisten-Bereich links von x = -360 schützen. Wenn du dorthin klickst, soll der Stift nicht wie bisher reagieren. Das geht so:

```
def jump(x, y):
    if x < -360: return
    stift.penup()
    stift.goto(x, y)
    stift.pendown()

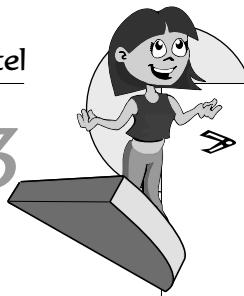
def moveto(x, y):
    if x < -360: return
    stift.goto(x,y)

def fuellenumschalten(xdummy, ydummy):
    if xdummy < -360: return
    if stift.filling():
        stift.end_fill()
    ##      stift.fillcolor("")
    else:
        stift.begin_fill()
    ##      stift.fillcolor("black")

...
screen.onclick(moveto)
screen.onclick(jump, 2)
screen.onclick(fuellenumschalten, 3)
stift.ondrag(moveto)
...
```



13



➤ Führe nun diese Änderungen im Script aus. Füge diese fünf Zeilen wie gezeigt ein. Zwei Zeilen in `fuellenumschalten()` müssen auskommentiert werden, denn die Füllfarbe wird ab nun von den Farbknöpfen eingestellt. Und `stift.goto` muss durch `moveto` ersetzt werden, um den linken Rand zu schützen. Prüfe – wie eben – nach, dass es nun richtig funktioniert.

Natürlich wollen wir doch etwas mehr Farben verwenden. Die entsprechenden Knöpfe kriegen wir so ins Grafik-Fenster:

```
stift.ondrag(moveto)

for (f, y) in (("red", 160), ("green", 120), ("blue", 80),
               ("yellow", 40), ("cyan", 0), ("magenta", -40),
               ("black", -80), ("white", -120)):
    farbButton(f, y)

screen.onkeypress(stift.undo, "BackSpace")
```

➤ Füge auch dies noch in den unteren Teil des Scripts, den »Hauptprogramm«-Teil ein und teste das so erweiterte Programm. Fertige eine bunte Scribblelei an. So was vielleicht:



Schade, dass du die Farben nicht sehen kannst.



» Wenn du keine Fehler mehr findest, speichere eine Kopie von scribble_arbeit.py als scribble07.py ab.

Um das Python-»ROCKS«-Bild herzustellen, musste ich mich sehr konzentrieren und habe doch immer wieder Fehler mit der undo-Taste rückgängig machen müssen. Grund: Man kann jetzt nicht mehr erkennen, ob Füllen ein- oder ausgeschaltet ist.

Wie können wir das ändern? Wir installieren einen Füllfarbanzeiger, der die jeweilige Füllfarbe anzeigt – was bisher der Stift machte. Den können wir dann wieder verwenden, um den Füllzustand anzuzeigen. Wir erzeugen einen Füllfarbanzeiger durch folgenden Code:

```
stift.pensize(3)

fuellanziger = Turtle(shape="circle")
fuellanziger.pu()
fuellanziger.speed(0)
fuellanziger.goto(-380, -240)
fuellanziger.color("black", "")

screen.onclick(moveto)
```

Jetzt müssen wir nur noch unsere Knöpfe, oder genauer: die setfillcolor()-Funktion, so ändern, dass sie die Farbe des Füllfarbanzeigers ändern und die des Stiftes, wenn er gerade füllt.

```
def farbButton(farbe, y):
    ...
    def setfillcolor(xdummy, ydummy):
        fuellanziger.fillcolor(farbe)
        if stift.filling():
            stift.fillcolor(farbe)
    ...

```

Und den Code von füllenumschalten() müssen wir so ändern, dass er wieder wie früher funktioniert, aber eben nicht mit "black", sondern mit der Füllfarbe, die wir dem Füllfarbanzeiger entnehmen können. Die Methode fillcolor() gibt nämlich die Füllfarbe zurück, wenn wir sie ohne Argument aufrufen.

13



```
def fuellenumschalten(xdummy, ydummy):
    if xdummy < -360: return
    if stift.filling():
        stift.end_fill()
        stift.fillcolor("")
    else:
        stift.begin_fill()
        stift.fillcolor(fuellanzeiger.fillcolor())
```

- Führe diese Ergänzungen und Änderungen des Programmcodes durch und teste das Programm.

Du hast jetzt – mit gut 70 Zeilen Programmcode – schon ein ganz schönes Anwendungsprogramm beisammen. Ich finde, es ist Zeit, dem Programm einen Titel zu geben.

- Füge in den Programmcode gleich nach der Erzeugung des screen-Objekts folgende Programmzeile ein:

```
screen.title("SCRIBBLE - Das Malprogramm")
```

Natürlich kannst du dir auch einen eigenen Titel ausdenken, der dir besser gefällt.

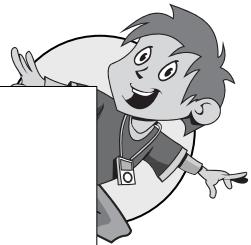
- Speichere eine Kopie von scribble_arbeit.py als scribble08.py ab.

Viel mehr Farben

In ausgewachsenen Anwendungsprogrammen hat man nicht nur Werkzeugleisten zur Verfügung. Vielmehr können manche Eigenschaften mit aufwendigen grafischen Dialogen eingestellt werden. Einen solchen wollen wir jetzt in Betrieb nehmen. Dazu müssen wir aber ein wenig über den Tellerrand des turtle-Moduls hinaus schauen.

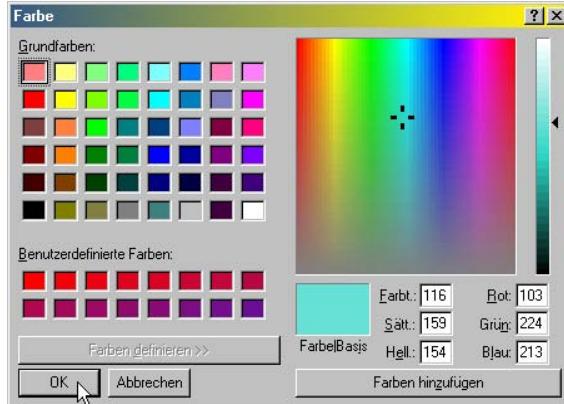
Der grafische Werkzeugkasten Tkinter, der bei Python dabei ist und auf dem auch das turtle-Modul aufbaut, enthält einige solche Dialoge. Darunter einen zur Auswahl einer Farbe aus einer Menge von 16 Millionen Farben. Er wird durch die Funktion askcolor() aus dem Untermodul colorchooser von tkinter aufgerufen.

Viel mehr Farben



» Mach mit!

```
>>> from tkinter.colorchooser import askcolor  
>>> askcolor()
```



Sieht unter Linux und auf dem Mac ganz anders aus!

Nun geht das Farbwahl-Dialogfenster auf. Du erkennst wohl gleich, dass sich hier tkinter bei deinem Betriebssystem bedient. Sorge (mit der Maus) dafür, dass im Feld FARBE|BASIS die gewünschte Farbe erscheint, und klicke auf OK. Nun erhältst du als Rückgabewert der Funktion askcolor() Folgendes:

```
>>> askcolor()  
((103.40234375, 224.875, 213.83203125), '#67e0d5')
```

Das sieht ziemlich sonderbar aus: ein Zweiertupel. Das erste Element ist selbst ein Dreiertupel, das zweite ein String, der eine so genannte Hexadezimalzahl darstellt. Nicht leicht zu erklären.

» Mach weiter mit!

```
>>> (0x67, 0xe0, 0xd5)  
(103, 224, 213)
```

Hexadezimalzahlen werden mit 16 Ziffern aufgeschrieben: neben 0 ... 9 finden auch noch die »Ziffern« a b c d e f Verwendung. Sie stehen für die Zahlen 10, 11, 12, 13, 14, 15. Für Python bedeutet 0xd5 die zweistellige Hexadezimalzahl mit den Ziffern d und 5. Sie ergibt, in eine gewöhnliche Dezimalzahl umgewandelt 213. Wenn du nun von '#67e0d5' jeweils zwei Hexadezimalziffern in Dezimalzahlen umwandelst, erhältst du die ganzzahligen Anteile des Dreiertupels, das von askcolor() ausgegeben wurde. Neugierig?

<http://de.wikipedia.org/wiki/Hexadezimalsystem!>

13



Für uns genügt es, das zweite Element der Ausgabe von `askstring()` in `pencolor()` oder `fillcolor()` einzusetzen. Diese Methoden kennen nämlich die von `colorchooser()` verwendete Hexadezimalschreibweise.

➤ Führe `scribble_arbeit.py` aus und drücke auf die Taste [9].

➤ Mach weiter mit (`askcolor` sollte schon importiert sein)!

```
>>> farbe = askcolor()
>>> farbe
((103.40234375, 224.875, 213.83203125), '#67e0d5')
>>> farbe = farbe[1]  # zweites Element
>>> farbe
'#67e0d5'
>>> stift.fillcolor(farbe)
```

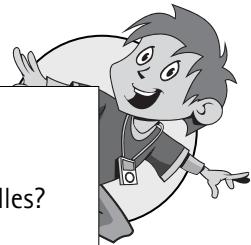
Damit hast du die ausgewählte Farbe als Füllfarbe auf den Stift übertragen. Das müssen wir nun noch ins Programm übertragen. Binden wir die Farbauswahl für Linien an "l" und die für Flächen (oder Füllen) an "f". Dazu brauchen wir eigentlich nichts Neues:

```
from turtle import Screen, Turtle
from tkinter.colorchooser import askcolor
import sys
sys.setrecursionlimit(20000)

def waehlelinienfarbe():
    farbe = askcolor()
    farbe = farbe[1]
    stift.pencolor(farbe)

def waehlefuellfarbe():
    farbe = askcolor()[1]  # dasselbe diesmal kürzer
    fuellanzeiger.fillcolor(farbe)
    if stift.filling():
        stift.fillcolor(farbe)

...
screen.onkeypress(stift.clear, "space")
screen.onkeypress(waehlelinienfarbe, "l")
screen.onkeypress(waehlefuellfarbe, "f")
screen.listen()
```



- » Füge diese Ergänzungen in scribble_arbeit.py ein.
 - » Experimentiere mit dem neuen Farbwahl-Feature! Funktioniert alles?
- Leider nicht ganz. Wenn du nämlich das Farbwahldialogfenster abbrichst, erscheint eine gründige Fehlermeldung, die sich weitschweifig darüber beschwert, dass in diesem Fall farbe den Wert None hat.
- » Füge in die beiden Funktionen je eine Abfrage ein, die bewirkt, dass die Funktionen in diesem Fall gleich mittels return verlassen werden.
 - » Alternativ könntest du den Fehler auch mit einer try ... except-Anweisung abfangen. Probiere es aus!
 - » Speichere eine Kopie von scribble_arbeit.py als scribble09.py ab.

Hilfe

Damit sind wir mit unserem Zeichenprogramm bei einem vorläufigen Ende angelangt. Fast. Doch das Programm bietet jetzt so viele Möglichkeiten, dass es nützlich wäre, diese auf einem Hilfebildschirm zu beschreiben. Üblicherweise ruft man den mit der Funktionstaste **F1** auf. Machen wir das noch:

Wir erzeugen eine eigene unsichtbare Turtle in der Form eines großen weißen Rechtecks. Wenn du auf **F1** drückst, soll sie das weiße Rechteck auf den Bildschirm stempeln – über die vielleicht vorhandene Grafik. Und darauf den Hilfetext schreiben. Wenn du danach **Esc** drückst, soll sie alles wieder löschen – und die Grafik wird wieder sichtbar.

Die Hilfe-Turtle erzeugen wir so:

```
fuellanziger.color("black", "")  
  
helper = Turtle(shape="square", visible=False)  
helper.pu()  
helper.speed(0)  
helper.fillcolor("white")  
helper.shapesize(35,27,3)  
  
screen.onclick(moveto)
```

13



Die Funktion, die den Helptext anzeigt, sieht so aus:

```
def show_help():
    screen.title("SCRIBBLE - Das Malprogramm")
    helper.stamp()
    helptext = """
        SCRIBBLE - Hilfe

Maus-Kommandos für die Zeichenfläche:
-----
Mausklick links -- zeichnet Linie zur Mausposition
Mausziehen links -- zeichnet Linie längs der Mausbewegung
Mausklick Mitte -- bewegt Stift zur Mausposition
Mausklick rechts -- schaltet Füllen ein bzw. aus

Maus-Kommandos für die Farb-Buttons:
-----
Mausklick links -- stellt Stiftfarbe ein
Mausklick rechts -- stellt Füllfarbe ein. Die Füllfarbe
                  wird unten im Füllfarbanzeiger
                  angezeigt

Tastatur-Kommandos:
-----
"1" ... "9" -- stellt Strichdicke auf die Werte 1 ... 9
Backspace -- macht die letzten Zeichenvorgänge rückgängig
Leertaste -- löscht Zeichnung
"]" -- ruft Farbauswahldialog für die Linienfarbe auf
"f" -- ruft Farbauswahldialog für die Füllfarbe auf
F1 -- ruft diese Hilfe auf
Escape -- schließt den Hilfe-Bildschirm.
"""

    helper.goto(0, -260)
    helper.write(helptext, align="center",
                font=("Courier", 12, "bold"))
    helper.home()
```

Eine andere Sorte Ereignis: Timer



Für das Löschen des Helpertextes verwenden wir einfach die Methode `helper.clear()`. Jetzt fehlen nur noch ganz wenige Ergänzungen. Ein Hinweis auf die Hilfe im Titel:

```
screen.setup(840, 600)
screen.title("SCRIBBLE - Das Malprogramm (Hilfe: F1)")
stift = Turtle()
```

Und die Bindung der beiden Funktionen an `F1` und `Esc`:

```
screen.onkeypress(show_help, "F1")
screen.onkeypress(helper.clear, "Escape")
screen.listen()
```

- Führe diese Änderungen und Ergänzungen durch und teste die Hilfe-Funktion.
- Bringe den Kopfkommentar auf den neuesten Stand und speichere eine Kopie von `scribble_arbeit.py` als `scribble10.py` ab und eine weitere Kopie als `scribble.py`, unser vorläufiges Endprodukt.

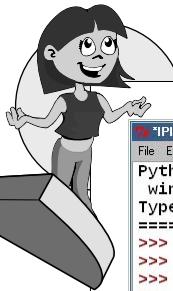
Du kannst dir sicherlich vorstellen, dass es genug Möglichkeiten gibt, das Programm noch zu verbessern. Vielleicht hast du demnächst Lust dazu. Eine erste Aufgabe wäre beispielsweise, wie erwähnt, nach den Anleitungen auf der Buch-Website eine Funktion zu implementieren, die die erstellten Grafiken abspeichert.

Eine andere Sorte Ereignis: Timer

- Starte IPI-TURTLEGRAFIK neu und mach mit!

```
>>> from turtle import Screen, Turtle
>>> screen = Screen()
>>> t = Turtle()
>>> def schritt():
    t.forward(80)
    t.left(90)
```

13



```

IPI Shell / Turtle Grafik
File Edit Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>> from turtle import Screen, Turtle
>>> screen = Screen()
>>> t = Turtle()
>>> def schritt():
    t.forward(80)
    t.left(90)

>>> screen.ontimer(schritt, 1000)
        (fun, t=0)
Installiere einen timer, der nach t Millisekunden fun aufruft.
Ln: 12 Col: 32

```

➤ Beobachte im Folgenden die Turtle:

```
>>> screen.ontimer(schritt, 1000)
>>> screen.ontimer(schritt, 1000)
>>> screen.ontimer(schritt, 1000)
```

➤ Beobachte die Turtle ganz genau und geduldig.

```
>>> screen.ontimer(schritt, 5000)
```

Verstehst du das? Vielleicht mit Hilfe des Tooltips in der IPI-Shell-Abbildung?

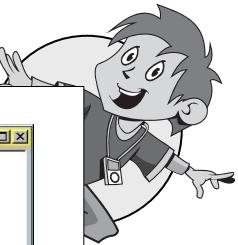
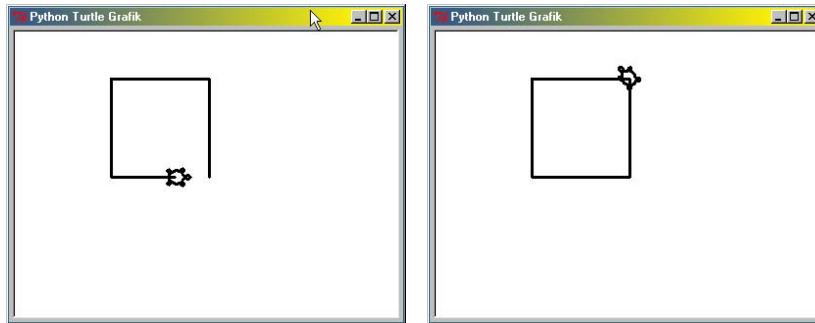
`ontimer()` ist eine Methode des `screen`-Objekts, die zwei Argumente übernimmt. Das erste muss eine Funktion `fun()` ohne Parameter sein. Das zweite ist eine Zeitangabe `t` in Millisekunden, also in Tausendstel Sekunden. `ontimer()` bewirkt einfach einen Aufruf von `fun()`, aber nicht sofort, sondern erst nach `t` Millisekunden. Bis dahin macht das Programm eine Pause! (In der Zwischenzeit kann die CPU andere Programme bedienen.) Deshalb musstest du nach dem letzten `ontimer()`-Aufruf mit `t = 5000` fünf Sekunden warten, bis die Turtle reagiert hat.

Das sieht zunächst nicht besonders aufregend aus: Wozu soll das gut sein? Das wirst du gleich sehen, wenn wir die Funktion `schritt()` in der folgenden genialen Weise erweitern:

```
>>> t.reset()
>>> def schritt():
    t.forward(80)
    t.left(90)
    screen.ontimer(schritt, 200)

>>> schritt()
```

Mach dir deine eigenen Turtle-Shapes



Damit wurde unsere Turtle zum »Perpetuum mobile«. Lässt sich nur durch Schließen des Grafik-Fensters stoppen!

So erweist sich die screen-Methode `ontimer()` als ein geeignetes Mittel, um animierte (bewegte) Grafikprogramme zu schreiben. Ein Beispiel dafür folgt etwas weiter hinten. Vorher kommt aber noch ein kurzer Einschub:

Mach dir deine eigenen Turtle-Shapes

Das Modul `turtle` kommt mit einigen wenigen vorgefertigten Turtle-Formen daher. Sie werden mit Strings bezeichnet und du kennst schon einige: "turtle", "arrow", "circle". Um zu erfahren, welche Turtle-Shapes definiert sind, gibt es in `turtle` eine screen-Methode: `getshapes()`. Bevor wir sie aufrufen können, brauchen wir allerdings ein neues Grafik-Fenster; wir haben ja das alte soeben geschlossen. Und dazu das neue screen-Objekt.

» Mach mit!

```
>>> screen = Screen()  
>>> screen.getshapes()  
['arrow', 'blank', 'circle', 'classic', 'square',  
'triangle', 'turtle']  
>>> t = Turtle()  
>>> t.shape("square")  
>>> t.left(1080)
```

Besonders viel Auswahl hast du da wirklich nicht. Das Modul `turtle` bietet dir aber die Möglichkeit, deine eigenen Turtle-Shapes zu machen. Und das geht so:

❖ Zuerst zeichnest du eine Figur, die die spätere Turtle-Gestalt werden soll. Für diese Figur müssen die Koordinaten der Eckpunkte aufgezeichnet werden. Dies leitet man mit der Anweisung `begin_poly()` ein und

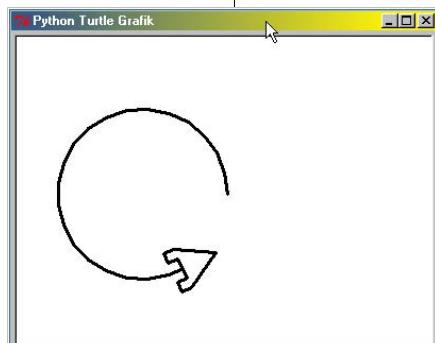
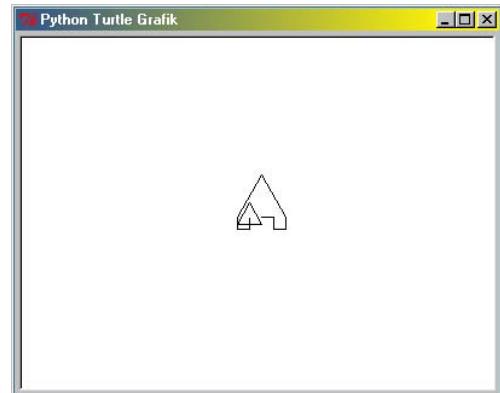
13



beendet man mit der Anweisung `end_poly()`. `get_poly()` gibt schließlich die Daten der so aufgezeichneten geometrischen Form zurück. Sieh dir das anhand des folgenden Beispiels an:

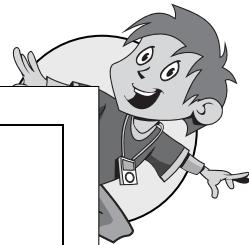
➤ Mach mit!

```
>>> t.reset()
>>> t.shape("triangle")
>>> t.rt(90); t.fd(10)
>>> t.begin_poly()
>>> t.rt(90); t.fd(10)
>>> t.lt(90); t.fd(10)
>>> t.lt(30); t.fd(40)
>>> t.lt(120); t.fd(40)
>>> t.lt(30); t.fd(10)
>>> t.lt(90); t.fd(10)
>>> t.lt(90); t.fd(10)
>>> t.end_poly()
>>> p = t.get_poly()
>>> p
((10.00,0.00), (10.00,-10.00), (20.00,-10.00), (20.00,0.00),
(0.00,34.64), (-20.00,0.00), (-20.00,-10.00), (-10.00,
-10.00), (-10.00,0.00))
```



Du hast damit ein Polygon (ein Vieleck, nicht regelmäßig) mit neun Ecken erzeugt. Beim Zeichnen eines solchen Polygons wird automatisch die letzte mit der ersten Ecke verbunden. Wir wollen nun dieses Polygon als »Shape«. Turtle-Gestalt, beim Grafik-Fenster registrieren. Es ist sinnvollerweise das Grafik-Fenster, das für alle Turtles die verfügbaren Shapes verwaltet. Dazu verwenden wir die `screen-Methode register_shape()` und geben unserer neuen Turtle-Gestalt den Namen "pfeil". Danach kann diese Figur als Turtle-Shape verwendet werden.

```
>>> screen.register_shape("pfeil", p)
>>> screen.getshapes()
['arrow', 'blank', 'circle', 'classic', 'pfeil', 'square', 'triangle',
'turtle']
>>> t.reset()
>>> t.shape("pfeil")
>>> t.pensize(3)
>>> t.right(720) # beobachte die Turtle!
>>> t.circle(80)
```



Muster 18: Erzeugen benutzerdefinierter Turtle-Shapes

1. Schritt: Erzeugung der Punktliste für die Turtle-Form mit einer Turtle *t*:

```
t.begin_poly()          Start der Aufzeichnungen der Polygonpunkte
Grafik-Anweisungen      ... zeichnen die neue Turtle-Shape
t.end_poly()             ← Ende der Aufzeichnungen der Punkte
p1 = t.get_poly()        Punktliste wird der Variablen p1 zugewiesen
```

2. Schritt: Registrierung der Shape unter einem Namen

```
screen.register_shape(namenstring, p1)
```

3. Schritt: Verwendung der Shape:

```
t.shape(namenstring)
```

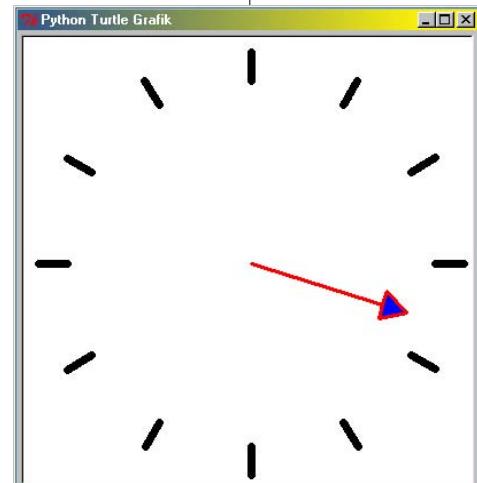
Uhr

Mit den neu erworbenen Kenntnissen über `ontimer()` und das Herstellen eigener Turtle-Shapes wollen wir nun ein Programm schreiben, das eine einfache Uhr auf dem Bildschirm darstellt.

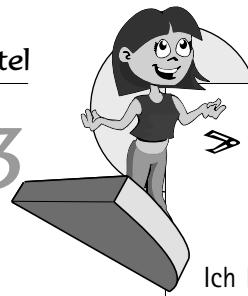
Das sieht durchaus einfach aus. Das Ganze besteht aus einem Ziffernblatt und einem (Sekunden-)Zeiger. (Überlegungen zu einer vollständigeren Uhr mit weiteren Zeigern folgen weiter hinten.) Der Sekundenzeiger muss sich drehen – wie es eine Turtle kann. Wir werden ihn daher als »Turtle mit Zeigerform« implementieren.

Da dieses Kapitel bis hierher schon ein wenig anstrengend war, gönne ich dir eine kleine Erholungspause, in der du eine Funktion für das Ziffernblatt erstellen kannst. Das ist für einen alten Turtle-Grafik-Hasen wie dich inzwischen kein Problem mehr.

⇒ Öffne ein Editor-Fenster von der IPI-SHELL aus, schreibe einen Kopfkommentar für `uhr_arbeit.py` und speichere.



13



➤ Schreibe nach den nötigen import-Anweisungen eine Funktion ziffernblatt(radius) und ein Hauptprogramm, das in ein quadratisches Turtle-Grafik-Fenster das Ziffernblatt zeichnet.

Ich habe das auch getan. Um dir zu zeigen, dass man auch mit der altbekannten anonymen Turtle eigene Shapes machen kann, habe ich die ausnahmsweise noch einmal verwendet. Außerdem funktioniert für sie nützlicherweise die Funktion jump() aus unserer Bibliothek.

Bei mir sieht der Code, wie immer eine von vielen möglichen Lösungen, wie folgt aus:

```
from turtle import *
from mytools import jump

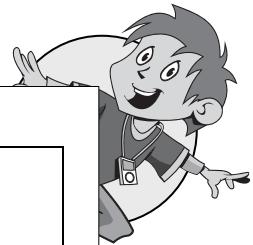
def ziffernblatt(radius):
    reset()
    pensize(7)
    for i in range(12):
        jump(radius)
        fd(25)
        jump(-radius-25)
        rt(30)

def uhr():
    screen.setup(400, 400)
    ziffernblatt(160)

if __name__ == "__main__":
    screen = Screen()
    uhr()
```

Die Anweisung screen = Screen() erzeugt wieder den globalen Namen screen, den wir auch später noch für den Aufruf von screen-Methoden benötigen werden.

Als Nächstes schreiben wir eine Funktion, die den Zeiger zeichnet: vom Ursprungsort der Turtle ausgehend und nach oben zeigend. Weil wir aber im Hinterkopf schon die Notwendigkeit notiert haben, später Zeiger anderer Größe zu erzeugen (Minuten-, Sekundenzeiger), verpassen wir der Funktion zeiger() zwei Parameter: laenge für die Länge des Schaftes und spitze für die Seitenlänge des Dreiecks, das die Zeigerspitze bildet.



```
def zeiger(laenge, spitze):
    fd(laenge)
    rt(90)
    fd(spitze/2)
    lt(120)
    fd(spitze)
    lt(120)
    fd(spitze)
    lt(120)
    fd(spitze/2)
```

- Füge den Code dieser Funktion gleich hinter den `import`-Anweisungen in das Programm `uhr_arbeit.py` ein. Führe das Programm aus und teste im Direktmodus mit dem IPI, ob diese Funktion verschiedene Zeiger korrekt zeichnen kann.

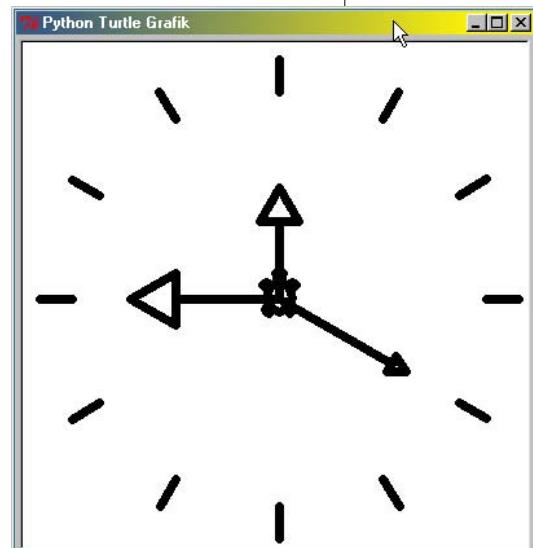
```
>>> zeiger(60,30)
>>> pu(); home(); pd()
>>> rt(120)
>>> zeiger(100, 15)
>>> pu(); home(); pd()
>>> rt(270)
>>> zeiger(80, 40)
>>> pu(); home(); pd()
```

Okay. Das funktioniert. Jetzt müssen wir eine passende Zeigerform zu einer verfügbaren Turtle-Shape machen. Im Direktmodus geht das leicht:

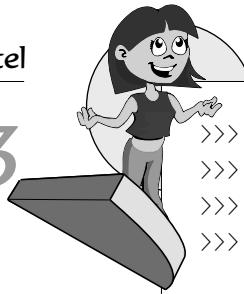
- Mach mit!

```
>>> reset()
>>> begin_poly()
>>> zeiger(130, 25)
>>> end_poly()
>>> zeiger_form = get_poly()
>>> screen.register_shape("sekundenzeiger", zeiger_form)
>>> reset()
>>> shape("sekundenzeiger")
>>> right(1080)
```

Fein, jetzt haben wir der Turtle Zeigerform gegeben. Wenn wir nun die Turtle mit `right()` drehen, dann dreht sich der Zeiger. (6° entsprechen einer Sekunde.) Und so können wir den Zeiger bewegen.



13



```
>>> speed(1)
>>> right(6)
>>> right(6)
>>> for i in range(58):
    right(6)
```

Fein, das funktioniert auch. Für das Zeichnen des Ziffernblattes und der Zeigerform haben wir die »anonyme« Turtle verwendet. Da die in Zukunft nicht die Rolle eines Uhrzeigers übernehmen soll, geben wir ihr wieder eine normale Gestalt:

```
>>> shape("turtle")
```

Als Zeiger (später vielleicht mehrere Zeiger) wollen wir wieder benannte Objekte der Klasse `Turtle` verwenden. Zunächst einmal eine Turtle namens `sekundenzeiger`. Und weil wir vielleicht mehrere unterschiedliche Zeigerformen zu `Turtle-Shapes` machen wollen, verpacken wir den entsprechenden Code auch in eine Funktion `mache_zeiger_shape()` mit Parametern für `name` der Turtle-Form sowie `laenge` und `spitze`. Wenn wir uns am obigen interaktiven Beispiel orientieren, muss diese Funktion so aussehen:

```
def mache_zeiger_shape(name, laenge, spitze):
    reset()
    begin_poly()
    zeiger(laenge, spitze)
    end_poly()
    zeiger_form = get_poly()
    screen.addshape(name, zeiger_form)
```

➤ Füge diesen Code unterhalb der Funktion `zeiger()` in das Script `uhr_arbeit.py` ein.

In der Funktion `uhr()` können wir jetzt diese Funktion wie folgt benutzen:

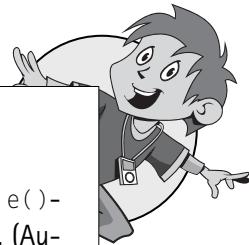
```
def uhr():
    global sekundenzeiger
    winsize(400, 400)
    shape("turtle")
    mache_zeiger_shape("sekundenzeiger", 130, 25)
    ziffernblatt(160)
    sekundenzeiger = Turtle()
    sekundenzeiger.shape("sekundenzeiger")
    sekundenzeiger.turtlesize(1, 1, 3)
    sekundenzeiger.color("red", "blue")
    sekundenzeiger.speed(1)
```

Animation des Uhrzeigers

- » Erweitere deine Funktion uhr() wie eben gezeigt.
- » Füge am Ende der Funktion ziffernblatt() eine hideturtle()-Anweisung ein, um die »anonyme« Turtle unsichtbar zu machen. (Aber du willst sie als »Markenzeichen« im Zentrum des Ziffernblattes stehen lassen.)
- » Führe das Programm aus und überprüfe, ob die Uhr richtig gezeichnet wird. Überlege, wie sich die Aufrufe der turtlesize()- und color()-Methoden in der Darstellung des Zeigers auswirken.

Ich hoffe, dass dir die Uhr halbwegs gefällt. Speichere eine Kopie davon als uhr01.py.

Du wirst aber sicherlich bedauern, dass sie noch nicht geht. Das werden wir jetzt auch noch ändern.



Animation des Uhrzeigers

Wir wollen erreichen, dass sich der Sekundenzeiger jede Sekunde um 6° nach rechts dreht. Das wird ein tick() der Uhr. Genauer: Der Zeiger soll sich um 6° nach rechts drehen. Dann soll 1000 Millisekunden gewartet werden und dasselbe wieder stattfinden, also wieder tick() aufgerufen werden. So was macht gerade screen.ontimer():

```
def tick():
    sekundenzeiger.right(6)    # 360/60
    screen.ontimer(tick, 1000)
```

Jetzt fehlt uns nur noch ein Aufruf von tick(). Setzen wir den ans Ende von uhr():

```
def uhr():
    global sekundenzeiger
    ...
    zeiger.speed(1)
    tick()
```

- » Füge die Definition der Funktion tick() und ihren Aufruf in das Script ein und teste es. Worin besteht hier das Testen? Wie lange dauert eine Minute auf dieser Uhr?

Diese Frage kannst du mittels der Funktion clock() aus dem Modul time, die du schon von Kapitel 10 kennst, leicht beantworten:

13



```
from turtle import *
from mytools import jump
from time import clock

...
def tick():
    print("{0:8.2f}".format(clock()))
    sekundenzeiger.right(6)
    screen.ontimer(tick, 1000)
```

Ich habe hier in der Formatmarke mit :8.2f das Format festgelegt: eine Gleitkommazahl (float), insgesamt 8 Zeichen breit mit 2 Nachkommastellen.

Wenn ich die Uhr mit dieser Ergänzung laufen lasse, erhalte ich im IPI-Shell-Fenster etwa folgende Ausgabe:

```
>>> 229.63
230.69
231.76
232.82
...
```

Leider, die Sekunde unserer schönen Uhr dauert etwas zu lange! Erstens, weil die Anweisungen in `tick()` etwas Zeit brauchen, und zweitens, weil `ontimer()` nicht besonders genau arbeitet. Jetzt hat aber jeder Computer eine ziemlich genaue Uhr eingebaut. Die können wir benutzen, um unsere Uhr zu steuern.

Datum und Uhrzeit ermitteln

Python hat einige Module, die mit Datum und Zeit zu tun haben. Das Modul `time` ist ein solches. Für unsere jetzigen Zwecke ist aber das Modul `datetime` besser geeignet. Dieses Modul enthält eine Klasse, die ebenfalls `datetime` heißt. Diese Klasse (!) hat eine Methode `now()`, mit der sie ein `datetime`-Objekt erzeugen kann, das die augenblicklichen Werte von Datum und Zeit enthält.

➤ Mach mit!

```
>>> from datetime import datetime
>>> jetzt = datetime.now()
>>> jetzt
datetime.datetime(2009, 8, 18, 14, 53, 54, 905000)
```

Datum und Uhrzeit ermitteln



jetzt verweist somit auf ein datetime-Objekt mit folgenden Eigenschaften:

```
>>> jetzt.year  
2009  
>>> jetzt.month  
8  
>>> jetzt.day  
18  
>>> jetzt.hour  
14  
>>> jetzt.minute  
53  
>>> jetzt.second  
54  
>>> jetzt.microsecond  
905000
```

Außerdem haben datetime-Objekte verschiedene Methoden, von denen dich vielleicht folgende interessiert:

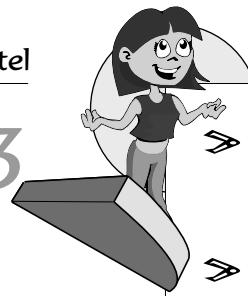
```
>>> jetzt.weekday()  
1
```

Sie gibt eine Zahl im Bereich 0 ... 6 zurück, die den Wochentag bezeichnet. Dabei bedeutet 0 Montag, 1 Dienstag usw. bis 6 Sonntag.

Damit können wir bei jedem Aufruf von tick() genau feststellen, wie spät es ist, davon den Sekundenanteil ermitteln und für die Einstellung des Sekundenzeigers nutzen. (Beachte: Eine Mikrosekunde ist eine Millionstelsekunde). Dabei ist es durchaus möglich, mehrmals pro Sekunde die Lage des Sekundenzeigers anzupassen, um eine gleichmäßige Zeigerbewegung zu erreichen:

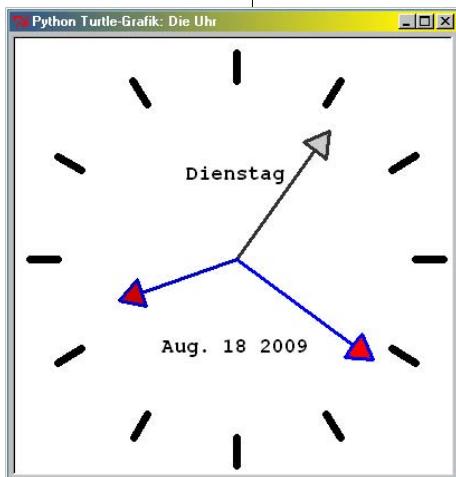
```
from turtle import *  
from mytools import jump  
from datetime import datetime  
  
def tick():  
    jetzt = datetime.now()  
    sekunden=jetzt.second + jetzt.microsecond * 0.000001  
    print("{0:8.2f}".format(sekunden))  
    sekundenzeiger.setheading(6*sekunden)  
    screen.ontimer(tick, 200)
```

13



- Ändere die import-Anweisung und den Code von tick() in dieser Weise ab. Teste dein Programm. Beobachte, dass nun deine Uhr mit der des Computers synchronisiert ist.
- Entferne zuletzt die print-Anweisung aus tick(). Sie wird nicht mehr gebraucht. Speichere eine Kopie von uhr_arbeit.py als uhr02.py ab.

Mehr Zeiger ...



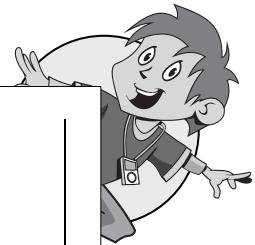
Wenn du Lust hast, kannst du nun darangehen, uhr_arbeit.py zu erweitern, bis deine Uhr zum Beispiel etwa so aussieht:

Ich will dir hier nur einen Anstoß für eine solche Erweiterung geben. Um den Minutenzeiger hinzuzufügen, brauchst du erstens den genauen Wert für die Minuten, mit Sekunden als Dezimalanteil. Das geht leicht, wenn du daran denkst, dass eine Sekunde gleich einer Sechzigstelminute ist. Zweitens brauchst du ein weiteres Zeigerobjekt minutenzeiger. Das muss in uhr() erzeugt werden. Folgende Änderungen machen einen Anfang:

```
def tick():
    jetzt = datetime.now()
    sekunden = jetzt.second + jetzt.microsecond*0.000001
    sekundenzeiger.setheading(6*sekunden)
    Minuten = jetzt.minute + sekunden/60
    minutenzeiger.setheading(6*Minuten)
    screen.ontimer(tick, 200)

def uhr():
    global sekundenzeiger, minutenzeiger
    screen.setup(400, 400)
    shape("turtle")
    mache_zeiger_shape("sekundenzeiger", 120, 25)
    mache_zeiger_shape("minutenzeiger", 130, 25)
    ziffernblatt(160)
    sekundenzeiger = Turtle(shape="sekundenzeiger")
    sekundenzeiger.turtlesize(1, 1, 3)
    sekundenzeiger.color("red", "blue")
```

Zusammenfassung



```
sekundenzeiger.speed(0)
minutenzeiger = Turtle()
minutenzeiger.shape("minutenzeiger")
tick()
```

Damit ist der Minutenzeiger noch recht farblos. Es bleibt also einiges zu tun, sowohl, was die Gestaltung der Uhr anbelangt, wie auch in puncto Stundenzeiger und Datumsangabe.

Für die Wochentags- und Datumsanzeige musst du mit der Turtle auf die Zeichenfläche schreiben. Dass du das mit der `turtle`-Funktion `write()` tun kannst, kannst du in folgender Weise interaktiv erforschen:

```
>>> reset()
>>> pu(); fd(90)
>>> write("Sonntag")
>>> bk(30)
>>> write("Montag", align="center")
>>> bk(30)
>>> write("Dienstag", align="center",
         font=("courier", 14))
>>> bk(30)
>>> write("Mittwoch", align="center",
         font=("courier", 14, "bold"))
>>> bk(50)
>>> write("11. Sept. 2001", align="center",
         font=("courier", 20, "bold"))
```



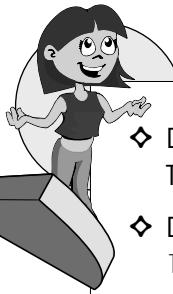
⇒ Wage dich an die Erweiterung des Uhrprogramms. Anspruchsvoll, aber lohnend! Zum Vergleich findest du eine Lösung auf der Buch-CD.

Zusammenfassung

Dieses Kapitel hat viele Neuigkeiten gebracht:

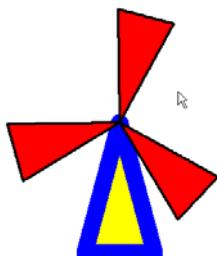
- ❖ `onclick()`, `onkeypress() + listen()` und `ontimer()` sind Methoden des `screen`-Objekts, mit denen man ereignisgesteuerte Programme erstellen kann.
- ❖ Turtle-Objekte haben ebenfalls Methoden, um auf Ereignisse reagieren zu können: `onclick()` und `ondrag()`.
- ❖ `screen.ontimer()` eignet sich besonders für die Erstellung grafischer Animationen.

13



- ❖ Das Modul `turtle` hat Funktionen zur Erzeugung benutzerdefinierter Turtle-Formen.
- ❖ Das Modul `tkinter.colorchooser` enthält die Funktion `askcolor()`, die einen Dialog für Farbauswahl zur Verfügung stellt.
- ❖ `datetime`-Objekte aus dem Modul `datetime` enthalten Informationen über Datum und Uhrzeit.

Aufgaben ...



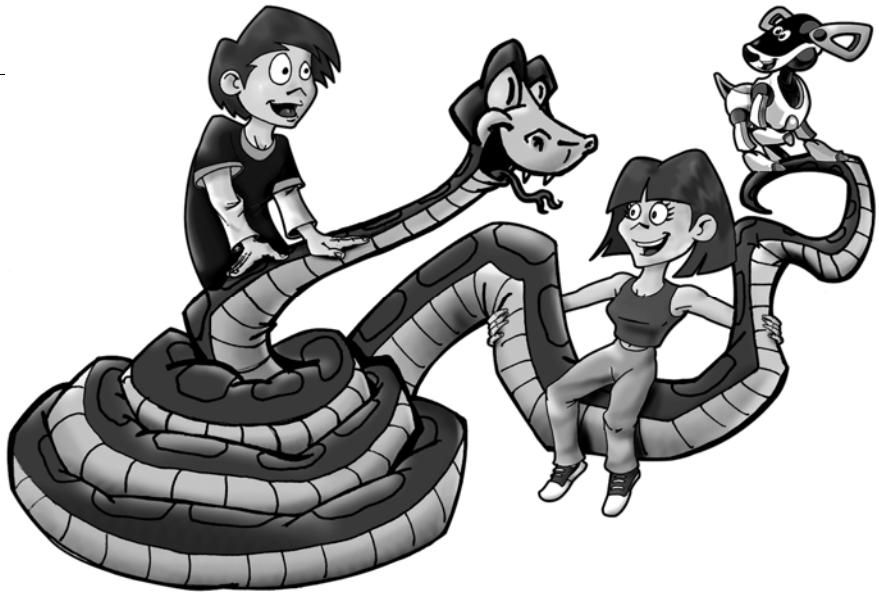
Aufgabe 1: Schreibe ein Programm `muehle.py`, das eine Windmühle zeichnet, deren Flügel sich drehen. Verwende dazu eine Turtle-Shape, die aus drei Dreiecken mit einer gemeinsamen Ecke besteht.

... und einige Fragen

1. Wie läuft ein ereignisgesteuertes Programm ab?
2. Auf welche Ereignisse kann das `screen`-Objekt reagieren?
3. Auf welche Ereignisse können Turtles reagieren?
4. Warum muss eine Funktion, die an `onclick()` als erstes Argument übergeben wird, zwei Parameter haben?
7. Welche Funktion dient zur Auswahl von Farben?
8. Welche Informationen sind in einem `datetime`-Objekt gespeichert?

14

Neue Klassen definieren

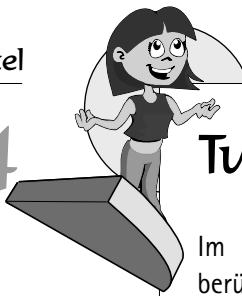


In diesem Kapitel lernst du endlich, was du alles brauchst, um deine eigenen Klassen zu definieren und damit zu selbst entworfenen Objekten zu kommen.

Dabei wirst du ...

- ◎ zunächst von unseren Turtles ausgehen und aus ihnen eine neue Klasse von »verbesserten« Turtles ableiten.
- ◎ sehen, wie die neuen Turtles eingesetzt werden können.
- ◎ eine ganz sonderbare Klasse `Bote` mit allen möglichen Variationen bis hin zu Agenten erstellen.

14



Turtles, die mehr können!

Im Kapitel 11 mussten wir bei der Programmierung von Dynaspiralen berücksichtigen, dass die Turtle-Objekte, die uns die Spiralen gezeichnet haben, unsere Bibliotheks-Funktion `jump()` nicht (als Methode) kennen. Daher mussten wir auf die Methoden `penup()`, `forward()`, `pendown()` zurückgreifen:

*Turtles haben keine
Methode `jump()`.*

```
dynaspiralen_arbeit.py - C:\py4kids\kap11\dynaspiralen_arbeit.py
File Edit Format Run Options Windows Help
# Python für Kids (4. Auflage) - Kapitel 11
# Objekte und Methoden

# Autor: Gregor Lingl
# Datum: 11. 8. 2009
# dynaspiralen_arbeit.py

from turtle import Turtle, Screen

screen = Screen()

alex = Turtle()
bert = Turtle()
bert.left(90)
bert.color("red")
carl = Turtle()
carl.color("green")
carl.left(180)
dinu=Turtle()
dinu.left(270)
dinu.color("blue")
kroeten = (alex, bert, carl, dinu)

for krot in kroeten:
    krot.hideturtle()
    krot.speed(0)
    krot.penup()      # funktioniert wie jump(50)
    krot.forward(50)  # schön wäre statt dessen:
    krot.pendown()   # krot.jump(50)
    krot.pensize(3)
    krot.right(30)

for laenge in range(100, 0, -5):
    screen.tracer(False)
    for krot in kroeten:
        krot.forward(laenge)
        krot.left(120)
    screen.tracer(True)
```

Lr: 23 Col: 33

Die Funktion `jump()` funktioniert – wie alle anderen Turtle-Grafik-Funktionen, die wir bisher programmiert haben – nur mit der namenlosen Turtle, die wir mit der Importanweisung `from turtle import *` zur Verfügung gestellt bekommen.

Wenn man mehrere Turtles benutzt, muss man jede Turtle-Grafik-Anweisung an eine bestimmte von ihnen schicken – mit unserer nun schon bekannten Punktnotation.

Turtles, die mehr können!



Genauer: Der Code von `jump()` sieht (etwas vereinfacht) so aus:

```
def jump(laenge):
    penup()
    forward(laenge)
    pendown()
```

Als wir in Kapitel 11 wollten, dass mehrere Turtles einen Sprung ausführen, mussten wir schreiben:

```
for krot in kroeten:
    krot.penup()
    krot.forward(laenge)
    krot.pendown()
```

Schöner wäre gewesen:

```
for krot in kroeten:
    krot.jump(50)
```

Das ging aber nicht, da Turtle-Objekte keine Methode `jump()` haben! Ein Mangel! Du kannst dir denken, dass sich der für Funktionen, die aus mehr Anweisungen als `jump()` bestehen, noch unangenehmer auswirkt.

Wir wollen uns nun eine erste Möglichkeit schaffen, beim Aufruf von `jump()` eine Turtle (das heißt: ein Turtle-Objekt) anzugeben, die den Sprung ausführen soll.

`jump()` braucht also eine zusätzliche Information. Wie über gibt man eine Information an eine Funktion? Als Argument! Es ist daher eine nahe liegende Idee, für das Turtle-Objekt, das zeichnen soll, einen neuen Parameter einzuführen:

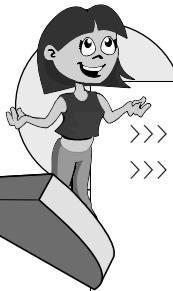
```
def jump(turtle, laenge):
    turtle.penup()
    turtle.forward(laenge)
    turtle.pendown()
```

Probieren wir das gleich aus:

➤ Mach mit:

```
>>> from turtle import Turtle
>>> alex = Turtle()
>>> bert = Turtle()
>>> bert.left(90)
```

14



```
>>> bert.pencolor("red")
>>> def jump(turtle, laenge):
    turtle.penup()
    turtle.forward(laenge)
    turtle.pendown()
```

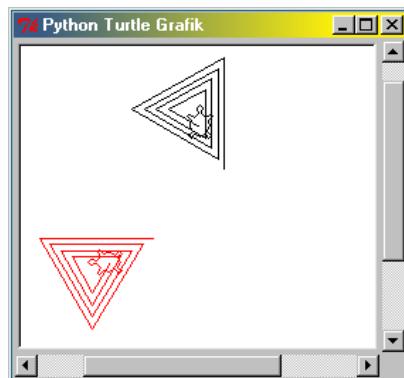
```
>>> for krot in (alex, bert):
    jump(krot, 50)
```

Ah, das funktioniert! Mutig geworden, fassen wir den »Drehschritt«, vorwärts/links, den die Turtles wiederholt für ihre Spiralen machen müssen, auch in eine kleine Funktion:

```
>>> def polyschritt(turtle, laenge, winkel):
    turtle.forward(laenge)
    turtle.left(winkel)
```

```
>>> for laenge in range(80, 20, -5):
    for krot in [alex, bert]:
        polyschritt(krot, laenge, 120)
```

Sehr ermutigend! Das bauen wir nun gleich in das Dynaspiralen-Programm ein.



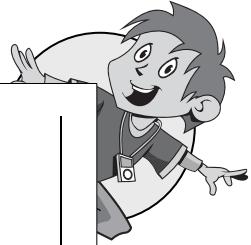
- Schließe das Turtle-Grafik-Fenster. Öffne von der IDLE aus `dynaspiralen_arbeit_.py` im Verzeichnis kap14 und speichere es als `dynaspiralen_arbeit.py`
- Füge gleich nach der `import`-Anweisung den Code der oben erprobten Funktionen `jump()` und `polyschritt()` ein.
- Ändere die Anweisungsblöcke der beiden `for`-Schleifen so ab, dass darin diese neuen Funktionen benutzt werden:

```
for krot in kroeten:
    krot.hideturtle()
    krot.speed(0)
    jump(krot, 50)
```

Turtles, die mehr können!

```
krot.pensize(3)
krot.right(30)

for laenge in range(80, 20, -5):
    for krot in kroeten:
        polyschritt(krot, laenge, 120)
```



- Speichere das Programm und führe es aus. Es sollte dasselbe Ergebnis liefern wie vor der Änderung.
- Speichere eine Kopie des Programms als dynaspiralen02.py ab.

Vielleicht empfindest du so wie ich, dass das Programm einheitlicher und schöner wäre, wenn der Code so lautete:

```
for krot in kroeten:
    krot.hideturtle()
    krot.speed(0)
    krot.jump(50)
    krot.pensize(3)
    krot.right(30)

for laenge in range(100,0,-5):
    for krot in kroeten:
        krot.polyschritt(laenge, 120)
```

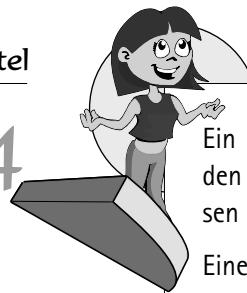
Was verlangen wir damit? Wir verlangen, dass die Turtles alex, bert, carl und dinu Botschaften verstehen sollen, die Turtle-Objekte eigentlich gar nicht verstehen: jump() und zum Beispiel polyschritt(). Wir wollen aber auch, dass sie alle Botschaften, die Turtle-Objekte verstehen, weiterhin verstehen. Du kannst das auch so formulieren: Sie sollen, im Vergleich zu gewöhnlichen Turtle-Objekten, über zusätzliche Methoden verfügen.

Denken wir an die kurz erwähnten Hunde Bello und Waldi aus Kapitel 11 zurück – gewöhnliche Hunde, zumindest wissen wir nicht mehr über sie. Wir sagten, Bello und Waldi sind Objekte der Klasse Hund.

Vielleicht kennst du auch einen Jäger mit einem Jagdhund, Cäsar. Jagdhunde verstehen einige Anweisungen, die gewöhnliche Hunde nicht verstehen. Jagdhunde bilden ebenfalls eine Klasse. Cäsar ist ein Objekt dieser Klasse.

Weil aber natürlich jeder Jagdhund auch ein Hund ist, sagen wir: Die Jagdhunde bilden eine Unterklasse der Hunde. Sie haben alle Eigenschaften und Fähigkeiten, die zu Hunden gehören – wir sagen: Sie erben alle Fähigkeiten und Eigenschaften der Klasse Hund. Und sie haben noch zusätzliche.

14



Ein Klasse kann im Übrigen durchaus mehrere Unterklassen haben. So bilden zum Beispiel die Schoßhunde oder die Blindenhunde andere Unterklassen der Klasse Hunde.

Eine ganz ähnliche Situation haben wir hier mit den Turtle-Objekten vor uns. Wir brauchen welche, die mehr Botschaften verstehen oder mehr Methoden beherrschen als gewöhnliche Turtle-Objekte. Um solche zu erhalten, brauchen wir eine Unterklasse der Klasse Turtle. Und genau eine solche wollen wir jetzt definieren.

Eine Unterklasse von Turtle

Aller Anfang ist schwer. Außer hier. Hier ist der Anfang leicht. Wir müssen lernen, wie man eine Klasse definiert und wie man angibt, dass sie eine Unterklasse einer bestimmten Klasse ist.

Unsere neue Klasse nennen wir `MyTurtle`. Sie soll eine Unterklasse von `Turtle` sein. Auch das können wir mit dem IPI untersuchen:

➤ Schließe das Turtle-Grafik Fenster und mach mit!

```
>>> class MyTurtle(Turtle):  
    pass
```

Das war's auch schon. Jetzt haben wir eine neue Klasse definiert. Sie ist Unterklasse von `Turtle`, daher kennen alle Objekte von `MyTurtle` automatisch alle Methoden von `Turtle`. Aber auch nicht mehr. Denn wir haben uns bei der Definition von `MyTurtle` wirklich auf das Allernotwendigste beschränkt, was in Python für eine zusammengesetzte Anweisung erforderlich ist: mit `pass` anzugeben, dass nichts weiter definiert wird. Sehen wir nach, ob das funktioniert hat:

```
>>> fritz = MyTurtle()  
>>> franz = MyTurtle()  
>>> fritz.pencolor("red")  
>>> fritz.forward(50)  
>>> franz.left(90)  
>>> franz.forward(50)
```

Funktioniert ja blendend! Wiederholen wir kurz: Was ist `MyTurtle`?

```
>>> MyTurtle  
<class '__main__.MyTurtle'>
```

Eine Unterklasse von Turtle



MyTurtle ist eine Klasse. Und fritz?

```
>>> fritz  
<__main__.MyTurtle object at 0x014B60F0>
```

fritz ist ein Objekt der Klasse MyTurtle. Python hat eine eingebaute Funktion, mit der man nachprüfen kann, ob ein Objekt Instanz einer Klasse ist:

```
>>> isinstance(fritz, MyTurtle)  
True
```

Ja! Ist fritz auch Objekt der Klasse Turtle?

```
>>> isinstance(fritz, Turtle)  
True
```

Ebenfalls. Warum? Weil MyTurtle eine Unterklasse von Turtle ist. (So wie Cäsar ein Objekt der Klasse Jagdhunde ist, aber auch ein Objekt der Klasse Hunde.)

So weit – so gut! Aber leider können fritz und franz noch nicht mehr als jede gewöhnliche Turtle. Oder willst du etwa fritz.jump(50) ausprobieren? Woher soll denn fritz wissen, wie jump(50) geht?

Das wollen wir unseren neuen Turtles jetzt beibringen. Das heißt, wir wollen die Definition der Klasse MyTurtle so treffen, dass Objekte dieser Klasse die Methoden jump() und polyschritt() kennen. Da diese Klassendefinition aber schon etwas umfangreicher sein wird, erstellen wir sie nicht im IPI, sondern in einer Datei, die wir aus dynaspiralen_arbeit.py entwickeln:

⇒ Öffne ein Editorfenster mit dynaspiralen_arbeit.py.

⇒ Schreibe über die Definition der ersten Funktion die Zeile:

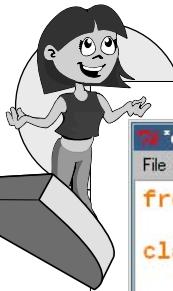
```
class MyTurtle(Turtle):
```

⇒ Um zu erreichen, dass die beiden folgenden Funktionsdefinitionen als Methoden in die Klassendefinition übernommen werden, markiere den Code der zwei Funktionen.

⇒ Wähle den Menüpunkt FORMAT|INDENT REGION oder drücke die -Taste mit demselben Effekt.

⇒ Kommentiere alle Anweisungen unterhalb von polyschritt() aus (Markieren und):

14



```
*dynaspiralen_arbeit.py - C:/py4kids/kap14/dynaspiralen_arbeit.py*
File Edit Format Run Options Windows Help
from turtle import Turtle, Screen

class MyTurtle(Turtle):
    def jump(turtle, laenge):
        turtle.penup()
        turtle.forward(laenge)
        turtle.pendown()

    def polyschritt(turtle, laenge, winkel):
        turtle.forward(laenge)
        turtle.left(winkel)

##screen = Screen()
##
##alex = Turtle()

Ln: 25 Col: 19
```

Die neue Klassendefinition für `MyTurtle`.

Die Einrückung der Funktionsdefinitionen macht diese zu Bestandteilen der Klassendefinition von `MyTurtle`. Es verhält sich hier genauso wie bei allen strukturierten Anweisungen von Python. Nur besteht hier der Körper der Klassendefinition aus mehreren Funktionsdefinitionen, die wir jetzt Methodendefinitionen nennen.

- Speichere das Programm, das die neue Klassendefinition enthält, und führe es aus. (Beachte: `dynaspiralen_arbeit.py` kann in dieser Situation noch nicht voll funktionieren, da die Anweisungen, die das Hauptprogramm bilden, noch nicht auf die neuen Verhältnisse mit der Klassendefinition umgestellt sind. Da wir sie auskommentiert haben, stört das im Moment nicht.)
- Schließe, falls noch offen, das Turtle-Grafik-Fenster. Untersuche die neue `MyTurtle`-Klasse mit dem IPL:

Zunächst erzeugen wir wieder zwei Objekte von `MyTurtle`. Der alte `fritz` und der alte `franz` haben leider die neuen Fähigkeiten nicht mitbekommen. Nur von jetzt an neu erzeugte Objekte bekommen sie:

```
>>> fritz = MyTurtle()
>>> franz = MyTurtle()
>>> fritz.pencolor("red")
>>> fritz.forward(50)
>>> franz.left(120)
>>> franz.forward(50)
```

Verlernt haben sie offenbar nichts! Und nun: Kennen sie die neuen Methoden?

Eine Unterklasse von Turtle

```
>>> fritz.jump(30)  
>>> franz.jump(30)
```

Das funktioniert! Und polyschritt?

```
>>> for i in range(3):  
    fritz.polyschritt(30,90)  
    franz.polyschritt(30,90)
```

```
>>> for laenge in range(24, 0, -2):  
    for kroete in [fritz, franz]:  
        kroete.polyschritt(laenge, 90)
```

Es ist uns also gelungen, die Funktionen jump() und polyschritt() an die Objekte der Klasse MyTurtle zu binden. Damit sind sie zu Methoden von MyTurtle-Objekten geworden!

Beachtenswert ist dabei aber Folgendes: Beim Aufruf einer Methode wird stets ein Argument weniger eingesetzt, als es Parameter in der Methodendefinition gibt. Ahnst du schon den Grund? Bevor ich ihn dir erkläre, rufe ich noch absichtlich eine Fehlermeldung hervor, aus der du natürlich wieder etwas lernen kannst. Ich setzte einfach eine falsche Zahl von Parametern in einen Methodenaufruf ein:

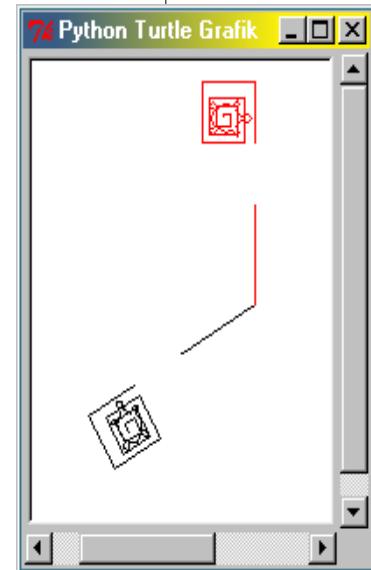
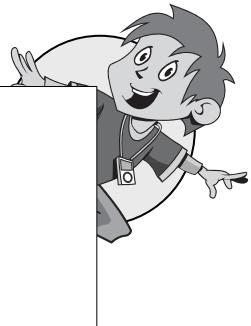
```
>>> fritz.polyschritt(1,2,3)  
Traceback (most recent call last):  
  File "<pyshell#40>", line 1, in <module>  
    fritz.polyschritt(1,2,3)  
TypeError: polyschritt() takes exactly 3 arguments (4 given)
```

Ich habe hier sinnlos drei Argumente in den Aufruf von polyschritt() eingesetzt. Nun kommt eine Fehlermeldung, die sagt: polyschritt() braucht genau drei Argumente, es wurden aber vier übergeben! Was steckt dahinter?

Dahinter steckt, dass der Python-Interpreter *intern*, wenn er eine Methode eines Objekts aufruft, immer *das Objekt selbst* als erstes Argument in den Methodenaufruf einsetzt. Konkret:

fritz.polyschritt(50,120) wird zu polyschritt(fritz,50,120).
franz.polyschritt(50,120) wird zu polyschritt(franz,50,120).

Der Aufruf fritz.polyschritt(1,2,3) findet daher vier Argumente, eines zu viel, vor: polyschritt(fritz,1,2,3).



14



Damit wird die Fehlermeldung auf einmal ganz verständlich! Und außerdem verstehst du jetzt auch, warum bei der Definition von Methoden der Parameter für das Objekt in den Methodenköpfen immer an erster Stelle stehen muss.

Nun wollen wir das Programm `dynaspiralen_arbeit.py` so umarbeiten, dass es die neue Klasse benutzt:

- »Entkommentiere« wieder alle Anweisungen, die nach der Klassendefinition stehen (markieren und **Alt+4**).
- Ersetze alle vier Konstruktor-Aufrufe `Turtle()` durch Aufrufe von `MyTurtle()`.
- Ändere die Aufrufe von `jump()` und `polyschritt()` so ab, dass sie Aufrufe von Methoden werden, also die Punktnotation verwenden. Der Programmteil muss dann genauso aussehen, wie wir es eben bei unserem IPI-Experiment ausprobiert haben.
- Speichere das Programm. Schließe das Grafik-Fenster und führe das Programm aus. Es sollte wieder dieselbe Grafik erzeugen.



Manchmal sind im Arbeitsspeicher des IPI von früherer Arbeit noch Definitionen von Funktionen vorhanden, deren Ausführung das Vorhandensein von neuen Fehlern überdeckt. Daher ist es sinnvoll, nach einer größeren Arbeit wie hier die IDLE zu schließen und neu zu starten. Man hat dann sicher einen sauberen Arbeitsspeicher. (Lies zu diesem Problem auch Anhang F ab Seite 442.)

- Schließe die IDLE mit *allen* Fenstern! Starte sie neu und lade das Programm `dynaspiralen_arbeit.py`. Führe es aus. Wenn es nicht richtig funktioniert, dann enthält es noch Fehler.
- Wenn es keine Fehler mehr enthält, speichere eine Kopie davon als `dynaspiralen03.py` ab.

Namen sind Schall und Rauch, aber auch wieder nicht!

Auch Variablennamen sind Schall und Rauch (J. W. v. Goethe: Faust, Teil I). Das heißt, du kannst in einem Programm jeden Variablennamen durch einen beliebigen anderen ersetzen, wenn du es nur an *allen* Stellen machst,

Namen sind Schall und Rauch, aber auch wieder nicht!



wo der Name verwendet wird, und wenn der Name nicht schon im selben Geltungsbereich verwendet wird.

Daher ist es ganz unproblematisch, in der Klassendefinition von MyTurtle:

```
class MyTurtle(Turtle):  
  
    def jump(turtle, laenge):  
        turtle.penup()  
        turtle.forward(laenge)  
        turtle.pendown()  
  
    def polyschritt(turtle, laenge, winkel):  
        turtle.forward(laenge)  
        turtle.left(winkel)
```

den Namen `turtle` durch einen anderen Namen zu ersetzen.

Andererseits hast du beim Durcharbeiten dieses Buches wohl bemerkt, dass Namen für Objekte keineswegs beliebig gewählt werden sollten. Vielmehr versucht man als Programmierer, »sprechende« Variablennamen zu verwenden, die dem Leser des Programms erklären, was mit ihnen bezeichnet wird.

Nun steht in der Klassendefinition von MyTurtle der Name `turtle` in allen Methodendefinitionen an der Stelle, an der beim Aufruf das aufrufende Objekt *selbst* eingesetzt wird: an der ersten Stelle der Parameterliste. Immer und für jede Methode, die zu einem Objekt gehört, gilt, dass das Objekt an die erste Stelle der Parameterliste eingesetzt wird!

Es hat sich daher die *Vereinbarung* durchgesetzt, für diesen ersten Parameter von Methodendefinitionen stets den Namen `self` zu verwenden. Damit soll eben klargestellt werden, dass dieser Parameter auf das Objekt selbst verweist, für das die Methode aufgerufen wird.

➤ Ersetze (am besten mit `EDIT|REPLACE...`) in der Klassendefinition von MyTurtle den Namen `turtle` an allen Stellen (außer in der `import`-Anweisung) durch den Namen `self`. Damit dabei die `import`-Anweisung und auch der Name MyTurtle unverändert bleibt, musst du im `EDIT|REPLACE...-Dialog` die Optionen `MATCH CASE` (Groß-/Kleinschreibung beachten) und `WHOLE WORD` (nur ganze Wörter ersetzen) anhaken! Damit bist du nun (etwa) bei folgendem Programm angekommen (bitte umblättern):

14



```
from turtle import Turtle, Screen

class MyTurtle(Turtle):

    def jump(self, laenge):
        self.penup()
        self.forward(laenge)
        self.pendown()

    def polyschritt(self, laenge, winkel):
        self.forward(laenge)
        self.left(winkel)

screen = Screen()

alex = MyTurtle()
bert = MyTurtle()
bert.left(90)
bert.color("red")
carl = MyTurtle()
carl.color("green")
carl.left(180)
dinu = MyTurtle()
dinu.left(270)
dinu.color("blue")
kroeten = (alex, bert, carl, dinu)

for krot in kroeten:
    krot.hideturtle()
    krot.speed(0)
    krot.jump(50)
    krot.pensize(3)
    krot.right(30)

for laenge in range(100,0,-5):
    screen.tracer(False)
    for krot in kroeten:
        krot.polyschritt(laenge, 120)
    screen.tracer(True)
```

➤ Speichere eine Kopie des Programms als `dynaspiralen04.py` ab.

- Dieses Programm tut genau dasselbe wie unser Ausgangsprogramm `dynaspiralen02.py`. Und dennoch hast du bei seiner Entwicklung viele neue und wichtige Dinge gelernt:



Um Objekte mit gewünschten Eigenschaften und Fähigkeiten erzeugen zu können, musst du eine entsprechende Klasse definieren. Von Vorteil kann es dabei sein, wenn die neue Klasse Unterklasse einer schon vorhandenen Klasse sein soll. In diesem Kapitel haben wir `MyTurtle` als Unterklasse von `Turtle` definiert. Umgekehrt nennen wir `Turtle` eine Oberklasse von `MyTurtle`.

Die Objekte der Unterklasse erben dann alle Fähigkeiten, das heißt alle Methoden der Oberklasse.

Eine Klassendefinition beginnt mit einer Kopfzeile der Form

```
class Klassenname(Oberklasse):
```

Dahinter folgt – wie immer bei zusammengesetzten Anweisungen – eingerückt der Körper der Klassendefinition.

In ihm werden zusätzliche Methoden festgelegt. Die Methodendefinitionen müssen folgende Eigenschaften haben:

- ❖ Sie müssen im Körper der Klassendefinition eingerückt stehen.
- ❖ Sie müssen als ersten Parameter einen Namen für das Objekt selbst haben, für das die Methode aufgerufen wird. Dafür wird immer der Name `self` verwendet.
- ❖ Methodendefinitionen haben also immer mindestens einen Parameter: `self!`



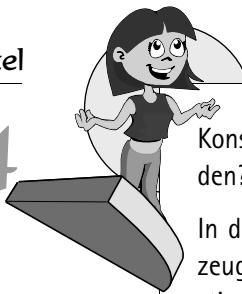
In der Definition haben die Methoden (scheinbar) einen Parameter mehr als die Anzahl der Argumente, die beim Aufruf eingesetzt werden. Doch bei der Ausführung der Methode ist das Objekt, für das die Methode aufgerufen wird, das Argument, das für den ersten Parameter eingesetzt wird. Er steht entsprechend der speziellen Syntax für Methodenaufrufe durch einen Punkt getrennt vor dem Methodennamen (Punktnotation).

Das war ein guter Anfang zum Thema Programmierung von Klassen. Doch dazu gibt es noch mehr zu sagen.

Der Konstruktor

Du weißt schon, dass der Konstruktor einer Klasse aufgerufen wird, um ein Objekt dieser Klasse zu erzeugen. Wir haben aber für `MyTurtle` keinen

14



Konstruktor programmiert. Muss ein Konstruktor nicht programmiert werden?

In der Regel doch. Der Konstruktor enthält Programmcode, der bei der Erzeugung eines Objekts ausgeführt wird. Durch diesen Code können bestimmte Eigenschaften des Objekts eingestellt werden oder auch verschiedene Effekte bewirkt werden. So hat die Klasse Turtle einen programmierten Konstruktor, der unter anderem ein Turtle-Grafik-Fenster öffnet, wenn noch keines vorhanden ist. In diesem Fenster erscheint dann das Turtle-Objekt. Das hast du ja stets beobachtet, wenn du mit dem IPI eine Turtle erzeugt hast. Außerdem werden im Konstruktor allerhand Eigenschaften der Turtle festgelegt: wo sie sitzt, wie sie orientiert ist, ihre Farbe, ihre Strichdicke und so weiter.

Ganz dasselbe geschieht auch, wenn du ein MyTurtle-Objekt erzeugst, obwohl wir keinen Konstruktor für die Klasse MyTurtle programmiert haben. Wie kommt das?

Das kommt so: Wenn eine abgeleitete Klasse keinen Konstruktor hat, dann wird bei der Konstruktion eines Objekts dieser Klasse automatisch der Konstruktor der Oberklasse aufgerufen.

Wenn wir aber wollen, dass bei der Erzeugung eines MyTurtle-Objekts zusätzliche Eigenschaften festgelegt oder zusätzliche Aktionen ausgeführt werden, dann müssen wir dafür einen eigenen Konstruktor schreiben.

Das wollen wir jetzt tun. Der neue Konstruktor soll ein gleichartiges Objekt wie der Konstruktor von Turtle erzeugen, das aber schon eine bei der Erzeugung wählbare Stiftfarbe und Orientierung haben kann. Farbe und Orientierung sollen dem Konstruktor als Argument übergeben werden. Das heißt, die Anweisung:

```
fritz = MyTurtle("red", 90)
```

macht dasselbe wie bisher schon fritz = MyTurtle() und stellt zusätzlich die Stiftfarbe von fritz auf Rot und die Orientierung auf 90° ein. Wir werden den Konstruktor so gestalten, dass die Farbe ein Argument mit dem Standardwert "black" ist und die Orientierung ein Argument mit dem Standardwert 0.

Der Konstruktor jeder Klasse ist eine »spezielle Methode« mit einem festgelegten Namen: `__init__`. Dieser Name hat zwei Unterstriche vor und zwei Unterstriche nach `init`. Wie du schon weißt, gibt es in Python eine ganze Reihe von Namen für »spezielle Methoden«, die mit diesem »2-mal-2-Unterstrich-Verfahren« gebildet werden.

Wir arbeiten weiterhin mit unserem kleinen Dynaspiralen-Programm:

Der Konstruktor

- Starte IDLE neu und lade dynaspiralen_arbeit.py. Schreibe in den Kopfkommentar: »Mit Konstruktor«.
- Füge unter dem Kopf der Klassendefinition folgenden Code für den (noch leeren) Konstruktor ein:

```
def __init__(self):  
    pass
```

Beachte, dass auch er als ersten Parameter `self` braucht.

- Speichere das Programm und versuche, es auszuführen.

Das Ergebnis ist: ein Turtle-Fenster ohne Turtle! Und eine schwer verständliche Fehlermeldung! Grund: Es gibt kein Turtle-Objekt! Denn wenn `MyTurtle` einen eigenen Konstruktor hat, dann wird *nicht automatisch* der Konstruktor der Oberklasse aufgerufen. Doch dagegen gibt es ein Mittel: Wir rufen ihn selbst auf.

Nun ist eine Besonderheit zu beachten: Da wir noch kein Objekt der Klasse `Turtle` haben, steht uns der Konstruktor nicht als Methode eines Objekts zur Verfügung, sondern nur als Bestandteil der Klassendefinition von `Turtle`. Daher muss er wie folgt aufgerufen werden:

```
class MyTurtle(Turtle):  
    def __init__(self):  
        Turtle.__init__(self)
```

Lies es so: Das Objekt *selbst* wird als ein Turtle-Objekt konstruiert, indem der Konstruktor `__init__()` der Klasse `Turtle` aufgerufen wird.

- Speichere das Programm und führe es aus. Es funktioniert jetzt so wie früher.

Wir haben damit noch nichts gewonnen, außer ...? – außer der Möglichkeit, noch weitere Aktionen bei der Konstruktion eines `MyTurtle`-Objekts ausführen zu lassen, indem wir einfach weitere Anweisungen in den Konstruktor hineinschreiben. Wir können nach dem Aufruf von `Turtle.__init__()` bereits auf alle Turtle-Methoden über `self` zugreifen, wie es auch in den danach folgenden Methodendefinitionen geschieht!

Den Plan, den `MyTurtle`-Objekten gleich bei der Konstruktion eine Farbe zu verpassen, realisieren wir so:



14



```
def __init__(self, farbe="black", winkel=0):
    Turtle.__init__(self)
    self.color(farbe)
    self.left(winkel)
```

Der Konstruktor von MyTurtle bekommt zwei zusätzliche Parameter, und das neu erzeugte MyTurtle-Objekt verwendet gleich die von Turtle geerbten Methoden `color()` und `left()`, um die Farbe einzustellen.

➤ Ändere den Code von `__init__()` wie angegeben ab. Speichere das Programm und führe es aus.

Es sollte funktionieren wie gehabt. Natürlich nutzt unser Programm die neu gewonnenen Möglichkeiten noch nicht voll aus. Aber weil wir beide Parameter mit Standardwerten versehen haben, funktioniert der Aufruf `MyTurtle()` noch immer.

➤ Ändere den Standardwert auf "yellow". Damit wird jede neue MyTurtle anfangs einmal auf Gelb eingestellt. Wenn ihre Farbe in der Folge nicht geändert wird, bleibt sie so. Kannst du voraussagen, was nun das Ergebnis eines Programmlaufes sein wird? Probiere es aus!

➤ Stelle danach die Standardfarbe wieder auf Schwarz ein.

Im Programmcode von `dynaspiralen_arbeit.py` siehst du, dass drei von den Kröten durch einen Aufruf der Methode `color()` Farbe bekommen und sich dann auch noch um einen bestimmten Winkel drehen. Dies können wir nun gleich bei der Konstruktion erledigen:

```
alex = MyTurtle()
bert = MyTurtle("red", 90)
carl = MyTurtle("green", 180)
dinu = MyTurtle("blue", 270)
kroeten = [alex, bert, carl, dinu]
```

➤ Führe diese Änderung in `dynaspiralen_arbeit.py` aus und führe das Programm aus.

➤ Verwende nun dasselbe Verfahren, um bei der Konstruktion von MyTurtle-Objekten auch die Strichstärke einstellen zu können. Stelle damit schon bei der Konstruktion der vier Kröten die Strichdicke auf 3 ein.

Wenn du das nicht auf dich allein gestellt versuchen möchtest, dann halte dich an die folgenden Tipps:

Noch ein einfaches Beispiel: Boten

- Füge in der Parameterliste des Konstruktors einen weiteren Parameter `strich` mit Standardwert 1 an.
- Füge als letzte Anweisung im Konstruktor einen Aufruf von `self.pensize()` mit dem Parameternamen `strich` als Argument an.
- Setze bei den vier MyTurtle-Konstruktor-Aufrufen als drittes Argument 3 ein. Beachte, dass nun `alex` auch ein Farb-Argument braucht. Wie wär's zur Abwechslung mit "pink"?
- Entferne den `krot.pensize()`-Aufruf aus der ersten `for`-Schleife.
- Speichere das Programm und teste es. Wenn es zufriedenstellend funktioniert, speichere eine Kopie als `dynaspiralen05.py`.



Wir haben jetzt ein schönes Stück Arbeit geleistet und ich bin ziemlich sicher, dass dir jetzt ganz schön der Kopf raucht. Wir haben die neue Klasse `MyTurtle` von der bestehenden `Turtle`-Klasse abgeleitet. Die Objekte dieser neuen Klasse werden mit einem »verbesserten« oder leistungsfähigeren Konstruktor erzeugt und sie kennen zusätzliche Methoden.

Da steckt viel von den Grundideen der objektorientierten Programmierung drin. Doch oft ist bei der Durcharbeitung nur eines Beispiels nicht gleich ersichtlich, was eigentlich das Wesentliche an der Sache ist. Ich möchte daher hier einen Abschnitt einschieben, in dem noch einmal die Grundkonzepte an einem anderen ganz einfachen Beispiel von Anfang an durchprobiert werden. Dabei wirst du sogar einige neue Aspekte kennen lernen.

Noch ein einfaches Beispiel: Boten

Mit diesem Beispiel will ich dir ausschließlich nochmals die Grundbegriffe des Programmierens von Objekten veranschaulichen. Und zwar von der Pike auf! Auf den praktischen Nutzen kommt es mir bei diesem Beispiel nicht an.

Dafür ist es so klein, dass wir alles mit dem interaktiven Interpreter machen können. Trotzdem findest du die Klassen, die in dieser Übung definiert werden, auch auf der Buch-CD in der Datei `boten.py` zum Ausführen, Ausdrucken, Durchlesen oder wozu immer.

Boten sind Leute, die eine Botschaft überbringen oder eine Meldung ausgeben. Wir wollen eine Klasse `Bote` ganz neu definieren, deren Objekte das tun können.

14



➤ Starte IDLE (PYTHON GUI) und mach mit!

```
>>> class Bote:
    def melde(self):
        print("Ich melde dir:", "Hallo!")
```

Objekte dieser Klasse haben nur sehr bescheidene Fähigkeiten:

```
>>> herold = Bote()
>>> herold.melde()
Ich melde dir: Hallo!
>>> melder = Bote()
>>> melder.melde()
Ich melde dir: Hallo!
>>>
```

Die Boten kennen nur eine Methode, etwas zu melden, und melden immer dasselbe. So kann das nicht bleiben. Sie sollen uns doch eine Botschaft überbringen. Also verbessern wir die Methode `melde()`:

```
>>> class Bote:
    def melde(self, text):
        print("Ich melde dir:", text)

>>> herold = Bote()
>>> herold.melde("Feuer am Dach!")
Ich melde dir: Feuer am Dach!
>>> herold.melde("Hurrikan im Anzug!")
Ich melde dir: Hurrikan im Anzug!
```

Das ist schon ein Fortschritt. Der Methode `melde()` kann als Argument übergeben werden, was gemeldet werden soll. Bedauerlich ist allerdings, dass sich diese Boten nicht merken, was sie melden sollen. Für jede Meldung muss man es ihnen vorsagen. Besser wäre doch, sie könnten einen Meldungstext – ja! – speichern!

Das Problem dabei ist, dass wir ja aus einer Klasse `Bote` viele Boten-Objekte machen können oder wollen. Und jedes soll sich seinen eigenen Meldungstext merken. Das heißt, jedes Objekt (jede Instanz von `Bote`) braucht seine eigene Variable für den Meldungstext.

➤ Das geht, und das macht man so:

```
>>> class SchlauerBote:
    def merke(self, text):
        self.botschaft = text
    def melde(self):
        print("Ich melde dir:", self.botschaft)
```

Noch ein einfaches Beispiel: Boten



```
>>> schlaumeier = SchlauerBote()  
>>> schlaumeier.merke("Die Aktien fallen!")  
>>> schlaumeier.melde()  
Ich melde dir: Die Aktien fallen!  
>>> vifzac = SchlauerBote()  
>>> vifzac.merke("Das Wetter wird kalt!")  
>>> vifzac.melde()  
Ich melde dir: Das Wetter wird kalt!  
>>> schlaumeier.melde()  
Ich melde dir: Die Aktien fallen!  
>>>
```

Wir sagen: Der Name `botschaft` ist an das Objekt gebunden. Jedes Objekt hat einen eigenen Namen `botschaft`, und jeder dieser Namen kann auf einen anderen Wert verweisen. Die `botschaft` von `schlaumeier` verweist auf einen anderen String als die `botschaft` von `vifzac`. Kurz gesagt: Jeder Bote kennt und merkt sich seine eigene Botschaft!

Wir bezeichnen solche Variablen, die zu einem bestimmten Objekt gehören, als *Instanzvariable*. Instanzvariablen sind *Attribute* des Objekts.

Der Wert der Instanzvariablen `botschaft` kann jederzeit mit der Methode `merke` verändert werden:

```
>>> vifzac.merke("Das Wetter wird warm!")  
>>> vifzac.melde()  
Ich melde dir: Das Wetter wird warm!  
>>>
```

In der Tat erlaubt es Python sogar, den Wert einer Instanzvariablen direkt anzusehen:

```
>>> vifzac.botschaft  
'Das Wetter wird warm!'  
>>>
```

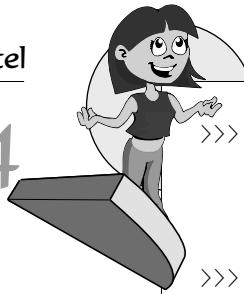
Und er kann auch direkt durch eine Wertzuweisung verändert werden:

```
>>> vifzac.botschaft = "Sauwetter!"  
>>> vifzac.melde()  
Ich melde dir: Sauwetter!
```

Beides ist in der Technik des objektorientierten Programmierens nicht ratsam. Wenn wir mit einer Klasse arbeiten, lernen wir zuerst, welche Methoden die Klasse hat. Denke hier doch an die Klasse `Turtle` zurück.

Jetzt machen wir Folgendes (hast du so etwas Ähnliches schon mal gesehen?):

14



```
>>> class SehrSchlauerBote(SchlauerBote):
    def merke_dazu(self, zusatz):
        self.botschaft=self.botschaft+" "+zusatz
```

```
>>> angelo = SehrSchlauerBote()
>>> angelo.merke("Kalt wird's!")
>>> angelo.melde()
Ich melde dir: Kalt wird's!
```

Der sehr schlaue Bote kann merken und melden. Obwohl nichts Derartiges in seiner Klassendefinition steht. Weil er ja alles kann, was ein schlauer Bote kann. Er *ist* ein schlauer Bote. Aber er ist auch ein sehr schlauer Bote, denn er kann noch mehr:

```
>>> angelo.merke_dazu("Warm anziehen!")
>>> angelo.melde()
Ich melde dir: Kalt wird's! Warm anziehen!
```

Damit wäre ein nur schlauer Bote überfordert, denn:

```
>>> merkur = SchlauerBote()
>>> merkur.merke("Stau auf allen Autobahnen!")
>>> merkur.merke_dazu("Reise verschieben!")
Traceback (most recent call last):
```

```
  File "<pyshell#62>", line 1, in <module>
    merkur.merke_dazu("Reisen verschieben!")
AttributeError: SchlauerBote object has no attribute
'merke_dazu'
```

Die Klasse SehrSchlauerBote **erbt alle Methoden von** SchlauerBote und kennt noch eine **zusätzliche**, `merke_dazu`. Schlaue Boten wissen davon aber natürlich nichts.

Einen ernsten Mangel haben unsere Botenklassen, der sich so zeigt:

```
>>> zweistein = SehrSchlauerBote()
>>> zweistein.melde()
Traceback (most recent call last):
  File "<pyshell#64>", line 1, in <module>
    zweistein.melde()
  File "<pyshell#15>", line 5, in melde
    print("Ich melde dir:", self.botschaft)
AttributeError: SehrSchlauerBote object has no attribute
'botschaft'
```

Die Instanzvariable `botschaft` von `zweistein` verweist auf nichts. Sie ist noch nicht *initialisiert*. Das ist kein guter Programmierstil. Initialisierung von Instanzvariablen gehört in den Code eines Konstruktors. Das erledigen wir gleich für `SchlauerBote` ...:

Noch ein einfaches Beispiel: Boten

```
>>> SchlauerBote  
<class '__main__.SchlauerBote'>
```

... und schreiben eine verbesserte Version der Klassendefinition:

```
>>> class SchlauerBote:  
    def __init__(self):  
        self.botschaft = "Hi!"  
    def merke(self, text):  
        self.botschaft = text  
    def melde(self):  
        print("Ich melde dir:", self.botschaft)
```

Nun haben wir also wieder eine Klasse SchlauerBote:

```
>>> SchlauerBote  
<class '__main__.SchlauerBote'>
```

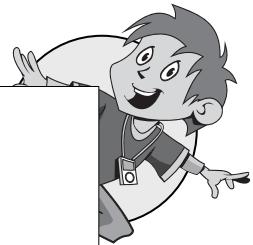
... aber eine andere, neu definierte. Damit die Klasse SehrSchlauerBote nicht mehr von der alten, sondern von dieser neuen erbt, müssen wir sie nochmals definieren:

```
>>> class SehrSchlauerBote(SchlauerBote):  
    def merke_dazu(self, zusatz):  
        self.botschaft = self.botschaft+zusatz  
  
>>> zweistein = SehrSchlauerBote()  
>>> zweistein.melde()  
Ich melde dir: Hi!
```

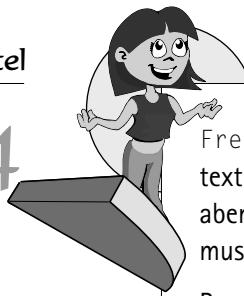
Nicht gerade sehr schlau! Aber besser als ein Programmabsturz. (Hinter den Kulissen ist es ja doch schlau, denn der Konstruktor von SehrSchlauerBote ruft *automatisch* den Konstruktor von SchlauerBote auf!)

Eine weitere Demo: Wir wollen uns eine Klasse FreundlicherBote machen, die von SchlauerBote abgeleitet ist:

```
>>> class FreundlicherBote(SchlauerBote):  
    def __init__(self, grusstext):  
        SchlauerBote.__init__(self)  
        self.gruss = grusstext  
    def gruesse(self):  
        print(self.gruss)  
  
>>> sunny = FreundlicherBote("Schönen Tag wünsch' ich!")  
>>> sunny.gruesse()  
Schönen Tag wünsch' ich!  
>>> sunny.melde()  
Ich melde dir: Hi!
```



14



FreundlicherBote hat einen eigenen Konstruktor, da er mit einem Grußtext aufgerufen wird und eine Instanzvariable initialisieren muss. Damit aber auch die Initialisierungsschritte von SchlauerBote gemacht werden, muss zusätzlich der Konstruktor der Oberklasse aufgerufen werden.

Beachte dazu: Das Objekt, das erzeugt und initialisiert wird, ist `self`. Die Form des Aufrufes des Konstruktors der Oberklasse ist nicht wie üblich

```
self.methodename(argumente...)
```

sondern:

```
Oberklassename.__init__(self, argumente...)
```

Dieser Aufruf braucht aber auch die Information, welches Objekt initialisiert werden soll, und daher musst du ihm `self` als erstes Argument übergeben:

```
SchlauerBote.__init__(self, ...)
```

Man sagt: `__init__` wird hier »als Klassenmethode« der Klasse `SchlauerBote` aufgerufen.



Wenn du eine Methode als »Klassenmethode«, d.h. mit dem Klassennamen vor dem Punkt aufrufst, musst du ihr das Objekt, für das sie aufgerufen werden soll, als erstes Argument übergeben.

Dies spielt auch eine Rolle in der nächsten Übung, die wir mit der Klasse `FreundlicherBote` anstellen:

```
>>> class FreundlicherBote(SchlauerBote):
    def __init__(self, grusstext):
        SchlauerBote.__init__(self)
        self.gruss = grusstext
    def gruesse(self):
        print(self.gruss)
    def melde(self):
        self.gruesse()
        SchlauerBote.melde(self)
```

```
>>> sunny = FreundlicherBote("Schönen Tag wünsch' ich!")
>>> sunny.melde()
Schönen Tag wünsch' ich!
Ich melde dir: Hi!
```

Hier haben wir die Methode `melde()` neu definiert, um zu erreichen, dass jeder freundliche Bote bei jeder Meldung vorher automatisch grüßt. Daran sind zwei Dinge bemerkenswert:

Noch ein einfaches Beispiel: Boten



- ❖ Diesmal wurde in der Klassendefinition von FreundlicherBote die Methode `melde()` als Klassenmethode der Oberklasse `SchlauerBote` aufgerufen und hat dementsprechend `self` als Argument übergeben bekommen. Warum? In der Definition der Methode `melde()` in der Klasse `FreundlicherBote` bedeutet der Name `self.melde` die Methode `melde` von `FreundlicherBote`. An dieser Stelle wird aber ein anderes `melde` gebraucht, nämlich die in der Oberklasse definierte Methode.
- ❖ In der Klasse `FreundlicherBote` wurde damit die von `SchlauerBote` geerbte Methode `melde()` mit einer neuen Version von `melde()` überschrieben. Objekte der Klasse `SchlauerBote` verfügen nach wie vor über die alte Methode `melde()`.

```
>>> sunny.merke("OOP ist eine steile Sache!")
```

```
>>> sunny.melde()
```

Schönen Tag wünsch' ich!

Ich melde dir: OOP ist eine steile Sache!

```
>>> tim = SchlauerBote()
```

```
>>> tim.merke("Morgen passiert's!")
```

```
>>> tim.melde()
```

Ich melde dir: Morgen passiert's!

Du siehst: Tim ist nicht freundlich!

Abschließend möchte ich dir noch zeigen, wie Objekte miteinander kommunizieren können. Hier einfacher: wie Boten miteinander Informationen austauschen können.

Ich möchte nämlich unsere schlauen Boten zu Agenten ausbilden.

Selbstverständlich ist ein Agent ein schlauer Bote. Wir werden die Agentenklasse also von unseren schlauen Boten ableiten. Aber Agenten können noch mehr – sie geben Botschaften an andere Boten weiter und sind auch imstande, andere Boten zu belauschen!

➤ Mach mit!

```
>>> class Agent(SchlauerBote):  
    def weitersagen(self, bote):  
        bote.merke(self.botschaft)  
    def belausche(self,bote):  
        self.botschaft = bote.botschaft
```

Jetzt kommen zwei Agenten:

```
>>>james = Agent()
```

```
>>>austin = Agent()
```

14



```
>>>james.melde()  
Ich melde dir: Hi!  
und gehen gleich an die Arbeit!  
>>> james.belausche(tim)  
>>> james.melde()  
Ich melde dir: Morgen passiert's!  
>>> austin.melde()  
Ich melde dir: Hi!  
>>> james.weitersagen(austin)  
>>> austin.melde()  
Ich melde dir: Morgen passiert's!  
>>>
```

Nicht schlecht! Wir haben Objekte erzeugt, die offenbar untereinander Informationen austauschen können! `james` hat `austin` verraten, was `tim` zu melden hatte!

Entscheidend für diese Fähigkeit der Zusammenarbeit von Boten (Objekten) ist, dass Objekte als Argumente an Methoden übergeben werden können: James kann mit Austin zusammenarbeiten, nämlich ihm etwas weitersagen, weil in James' Methodenaufruf von `weitersagen()` `austin` als Argument eingesetzt wurde!

Da capo al fine!

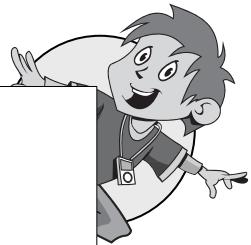
Von Anfang an noch mal! So etwas schreibe ich ganz selten. Aber was du in diesem Kapitel bis jetzt vorgesetzt bekommen hast, ist schon ziemlich starker Tobak. Ich halte es daher für ziemlich wahrscheinlich, dass du einiges davon noch nicht *ganz* verstanden hast. Wenn das so ist, rate ich dir, dieses Kapitel von Anfang an nochmals durchzugehen und genau zu durchdenken. Du wirst jetzt mit einem besseren Vorverständnis an die Sache herangehen und ich bin überzeugt, dass dir dabei noch einige Lichter aufgehen werden.

Zusammenfassung

- ❖ Klassendefinitionen werden mit dem reservierten Wort `class` eingeleitet.
- ❖ Die `class`-Anweisung ist eine zusammengesetzte Anweisung.
- ❖ Objekte werden durch Aufruf des Konstruktors einer Klasse erzeugt.
- ❖ Objekte haben Eigenschaften – Instanzvariablen.

Eine Aufgabe

- ❖ Objekte haben Fähigkeiten – Methoden.
- ❖ Instanzvariablen und Methoden nennt man *Attribute* der Objekte.
- ❖ Methoden sind an ein Objekt gebundene Funktionen.
- ❖ Der erste Parameter einer Methode erhält beim Aufruf als Argument stets das Objekt selbst, für das die Methode aufgerufen wird.
- ❖ In Python gilt (deshalb) die Vereinbarung, in Methodendefinitionen den ersten Parameter stets mit `self` zu bezeichnen.
- ❖ Klassen können von anderen Klassen, so genannten »Elternklassen«, abgeleitet werden.
- ❖ Abgeleitete Klassen erben alle Eigenschaften und Fähigkeiten der »Elternklassen« ...
- ❖ ... außer sie werden in ihrer Definition »überschrieben«.



Eine Aufgabe

Aufgabe: (a) Definiere eine Klasse `Bankkonto`. Objekte dieser Klasse sollen nur eine Eigenschaft haben: den Kontostand. Bei der Konstruktion soll auf das Konto gleich ein Geldbetrag eingelagert werden können. Wird kein Betrag eingelagert, dann ist eben der Kontostand anfangs 0.

Bankkonten – die Objekte der Klasse `Bankkonto` – sollen zunächst drei Methoden haben: `kontostand_abfrage()`, `einzahlen()`, `abheben()`.

Erzeuge einige `Bankkonto`-Objekte und lege Geld ein oder hebe es ab.

(b) Definiere zusätzlich eine Methode `ueberweise()`, die einen Betrag und ein (anderes) Konto als Parameter hat. Mit ihr soll Geld von einem Konto an ein anderes übertragen werden. (Denke dabei an das Beispiel der Agenten aus diesem Kapitel, die Botschaften übertragen können.)

... und nach diesem Kapitel keine Fragen

Nachdem du es sogar zweimal durcharbeiten musstest.

15

Moorhuhn

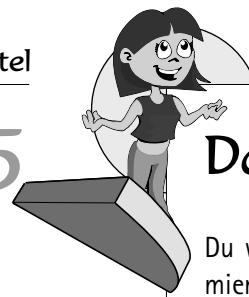


Ein ausgewachsenes Spielprogramm wollen wir in diesem letzten Kapitel des Buches erarbeiten: eine einfache Version des wohlbekannten Moorhuhn-Spiels.

Dabei wirst du ...

- ◎ lernen, wie man mit `turtle` mit Bildern arbeitet.
- ◎ eine weitere Art, Animationen zu programmieren, kennen lernen.
- ◎ erfahren, was Schlüsselwortargumente sind, wozu und wie sie gebraucht werden.
- ◎ ein vollständig auf der Definition von Klassen aufgebautes Programm erstellen.
- ◎ einen kleinen Blick aus der Welt von `turtle` in die Welt von Tkinter werfen.
- ◎ Klänge in dein Spielprogramm einbinden.

15

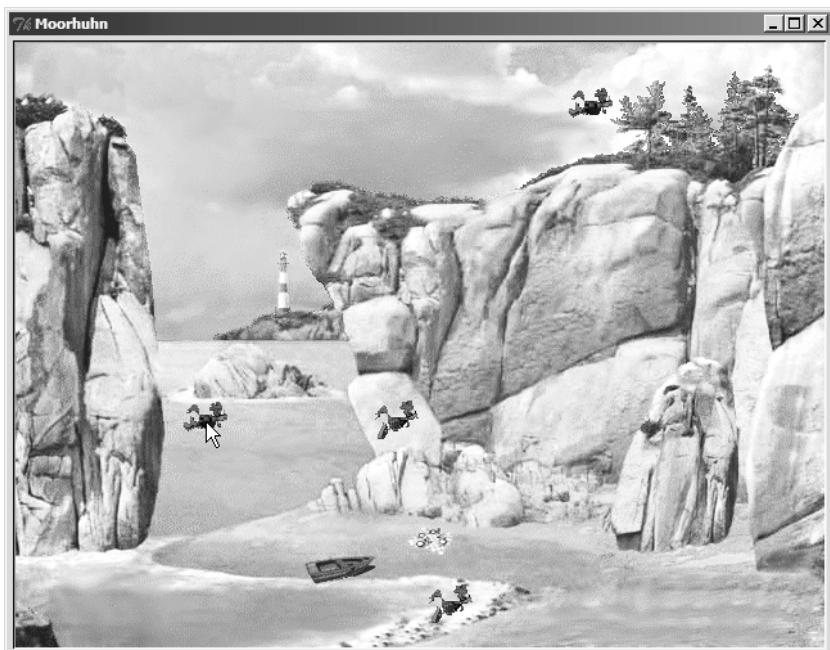


Das Spiel

Du wirst eine einfache Version des bekannten Moorhuhn-Spiels programmieren. Das ist ein Spiel, bei dem vor allem die Reaktionsschnelligkeit herausgefordert wird. Es geht darum, Moorhühner, die über eine Landschaft fliegen, mit Mausklicks zu treffen und damit abzuschießen. Der Spieler oder die Spielerin hat eine vorgegebene Anzahl von Schüssen zur Verfügung. Die Anzahl der erzielten Treffer und die Anzahl der abgegebenen Schüsse werden angezeigt.

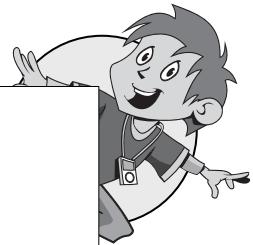
Eine Spielsituation sieht so aus:

Spielsituation
beim Moorhuhn-
Spiel



Besser als jede Beschreibung in Worten hilft dir sicher, das Spiel einfach ein paar Mal zu spielen. Auf der CD zum Buch findest du im Verzeichnis kap15 das Programm `moorhuhn_demo.pyw`. Das ist eine in Byte-Code übersetzte Version des Spiels. Den Python-Code für dieses Spiel wollen wir im Folgenden gemeinsam entwickeln.

➤ Spiele einige Male das Spiel Moorhuhn. (Schalte den Lautsprecher deines Computers ein.) Achte auf den Spielablauf und die Spiellogik. Versuche anschließend, in der Art eines Brainstormings Programmierideen zu entwickeln, die du brauchen könntest, um `moorhuhn.py` selbst zu programmieren.



Zunächst zwei spezielle technische Details

Derartige Programme haben wir bis jetzt noch nicht erstellt, und du wirst dir die Frage stellen, ob wir so etwas mit unserem Modul turtle hinkriegen oder ob wir dazu zusätzliche grafische Hilfsmittel benötigen. Ich kann dich beruhigen – bis auf wenige Kleinigkeiten werden wir mit turtle auskommen.

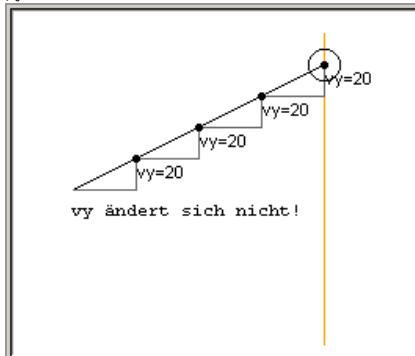
Bevor wir uns aber an die Programmierung des Spiels machen, möchte ich in zwei kurzen Abschnitten spezielle technische Fragen behandeln, die dabei eine Rolle spielen, damit wir später bei der Entwicklung des Spiels davon nicht abgelenkt werden.

Die Bewegung der Hühner: Flug und Absturz

Zunächst will ich dir zeigen, wie man eine Turtle im Grafik-Fenster kontrolliert in eine bestimmte Richtung bewegt.

➤ Starte IPI-TURTLEGRAFIK, importiere (ausnahmsweise noch einmal) alle Funktionen aus turtle und mach mit!

```
>>> from turtle import *
>>> reset()
>>> shape("circle")
>>> pu(); goto(-180, 0); pd()
>>> vx, vy = 40, 20
>>> x, y = position()
>>> while x < 0:
    x = x + vx
    y = y + vy
    goto(x,y)
    dot(5)
```

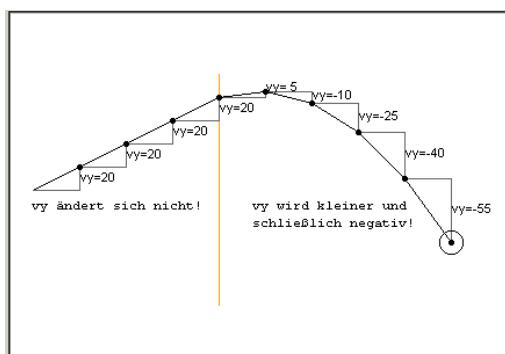


Zunächst ist die Geschwindigkeit der Turtle konstant ...

Erklärung: Weil die Orientierung der Turtle hier keine Rolle spielt, benutzen wir die kreisförmige Turtle-Shape. Die eben eingegebenen Anweisungen haben Folgendes bewirkt: Zuerst hast du die Turtle ohne zu zeichnen an den linken Fensterrand bewegt. Dann hast du zwei Zahlenwerte vx und vy festgelegt. Sie spielen etwa die Rolle der Geschwindigkeit in x-Richtung bzw. in y-Richtung. (Du kannst die Rolle dieser Zahlenwerte direkt untersuchen, indem du die gleichen Anweisungen mit anderen Zahlenwerten nochmals ausführst.) Danach hat die Turtle vier Bewegungsschritte schräg nach rechts oben ausgeführt – gleichzeitig um vx nach rechts und um vy nach oben – und nach jedem Schritt einen Punkt gezeichnet.

Eine kleine Änderung – von vy – in jedem Durchgang der while-Schleife bewirkt nun, dass die Turtle in eine gekrümmte Bahn einschwenkt. Und weil die Änderung der Geschwindigkeit dabei konstant ist – im folgenden Beispiel ist sie 15 –, ist die Bahn eine »Wurfparabel«. (Lust auf »Physik für Kids«?)

```
>>> while x < 160:
    vy = vy - 15
    x = x + vx
    y = y + vy
    goto(x, y)
    dot(5)
```



... doch dann ändert sie sich mit jedem Schritt.



Bilder und Turtle-Grafik

Wenn wir im Moorhuhn-Spiel für die Hühner Turtles verwenden wollen, müssen wir ihnen kleinen Bildchen als »Shapes« geben. Das geht in der Tat, wenn die Bilder im Format *.gif vorliegen.

Bevor du mit der Arbeit weitermachst, vergewissere dich, dass die Dateien `landschaft.gif`, `huhn01.gif` und `huhn02.gif` im Verzeichnis `C:\py4kids\kap15` vorhanden sind.

Wir müssen nun zuerst das Arbeitsverzeichnis für Python richtig einstellen:

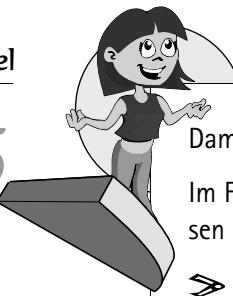
- Starte IPI-TURTLEGRAFIK neu und mach mit! (Die Pfadangaben musst du an dein Betriebssystem und an deine Verzeichnis-Struktur anpassen.)

```
>>> import os  
>>> os.getcwd()  
'C:\\py4kids'  
>>> os.chdir("C:/py4kids/kap15")  
>>> os.getcwd()  
'C:\\py4kids\\kap15'  
>>> os.listdir(".")  
['applaus.wav', 'daneben.wav', 'getroffen.wav',  
'huhn01.gif', 'huhn02.gif', 'jubel.wav', 'landschaft.gif']
```

Hier hast du gleich drei recht wichtige Funktionen aus dem Modul `os` kennengelernt. `os` steht als Abkürzung für *operating system*, Betriebssystem.

- ❖ Die Funktion `getcwd()` gibt das aktuelle Arbeitsverzeichnis aus (`getcwd` steht für *get current working directory*).
- ❖ Die Funktion `chdir()` wechselt das aktuelle Arbeitsverzeichnis auf das als Argument angegebene Verzeichnis (`chdir` steht für *change directory*).
- ❖ Die Funktion `listdir()` gibt eine Liste mit den Einträgen in dem Verzeichnis aus, das als Argument angegeben wird.

Dabei steht `..` immer für das aktuelle Arbeitsverzeichnis. Daher gibt der Aufruf `os.listdir(..)` eine Liste der Dateien (und Unterverzeichnisse) im aktuellen Arbeitsverzeichnis aus. So überzeugst du dich, dass die beiden Dateien `"huhn01.gif"` und `"huhn02.gif"` tatsächlich im aktuellen Arbeitsverzeichnis vorhanden sind. Wenn du im IPI/in der IDLE(PYTHON GUI) ein Script ausführst, ist das aktuelle Arbeitsverzeichnis immer das Verzeichnis, in dem das Script steht.



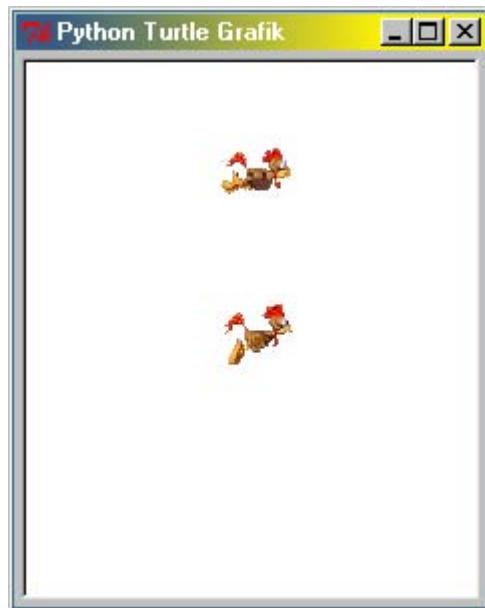
Damit ist es leicht, die beiden .gif-Bildchen zu Turtle-Shapes zu machen.

Im Folgenden wollen wir dazu aus dem Modul `turtle` wieder nur die Klassen `Screen` und `Turtle` importieren.

➤ Starte IPI-TURTLEGRAFIK neu und mach mit!

```
>>> from turtle import Screen, Turtle
>>> screen = Screen()
>>> huhn1 = Turtle()
>>> screen.register_shape("huhn01.gif")
>>> screen.getshapes()
['arrow', 'blank', 'circle', 'classic', 'huhn01.gif',
'square', 'triangle', 'turtle']
>>> huhn1.shape("huhn01.gif")
>>> huhn1.pu(); huhn1.forward(80)
>>> screen.register_shape("huhn02.gif")
>>> huhn2 = Turtle(shape="huhn02.gif")
```

Moorhühner mit Turtle-Shapes aus *.gif-Dateien



Hier haben wir die Namen von gif-Dateien als »Shapes« zunächst mit `screen.register_shape()` registriert und dann – wie gehabt – im Konstruktor der Klasse `Turtle` verwendet.

Wenn `register_shape()` einen Shape-Namen als Argument bekommt, der mit ".gif" endet, wird dieser als Name einer Bilddatei aufgefasst und diese Bilddatei als Turtle-Shape registriert. Danach kann die Methode `shape()` aufgerufen werden, um der Turtle diese Gestalt zu verleihen.

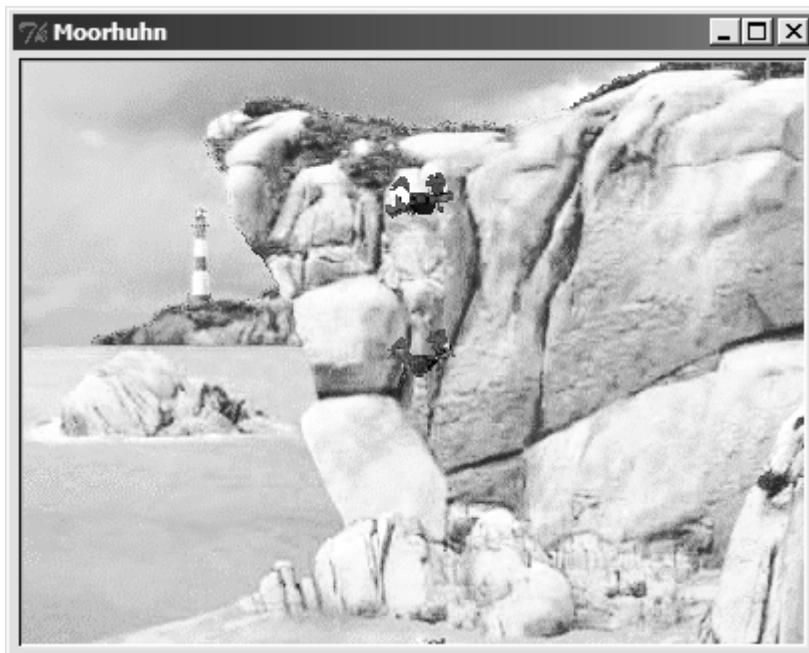


Turtles mit einer durch eine *.gif-Bilddatei festgelegten Gestalt können alle Methoden, insbesondere auch `left()` und `right()`, ausführen. Das Bildchen dreht sich aber dabei *nicht*. Daher kann man bei solchen »*.gif-Shapes« die Orientierung der Turtle nicht an der Turtle-Gestalt erkennen!

Erheblich einfacher geht die Einrichtung eines Hintergrundbildes. Das Objekt `screen` hat eine Methode `bgpic()`, um ein Hintergrundbild zu installieren, das in einer *.gif-Datei im selben Verzeichnis liegen muss wie das Script selbst. Auch diese Methode verlangt als Argument den Dateinamen.

➤ Mach weiter mit!

```
>>> screen.bgpic("landschaft.gif")
```



Moorhühner vor einer Landschaft.

Die Benutzeroberfläche, die Klasse MoorhuhnSpiel

Die Benutzeroberfläche des Moorhuhn-Spiels besteht aus einem Grafik-Fenster mit Hintergrundbild. Auf dem Grafik-Fenster werden Meldungen (z.B. über den Spielstand) ausgegeben und zudem bewegen sich darauf als Hühner verkleidete Turtles. Der Benutzer kann das Spiel mit der Leertaste in

15



Gang setzen und dann mit Mausklick-Ereignissen spielen. Mausklicks stellen Schüsse dar. Wird ein Moorhuhn getroffen, ändert es seine Bewegung: Es fällt zu Boden. Wenn alle Schüsse abgegeben worden sind, wird der Spielerin oder dem Spieler die erzielte Trefferquote mitgeteilt.

Wir wollen bei der Programmierung dieses Spiels unsere in den letzten Kapiteln erworbenen Kenntnisse über Klassen und Objekte einsetzen. Wir sind nun schon gewohnt, all die Funktionen, die wir früher mit `from turtle import *` zur Verfügung hatten, als Methoden der Turtles oder des `screen`-Objekts zu verwenden.

Methoden des `screen`-Objekts beziehen sich nicht auf eine Turtle, sondern auf das Grafik-Fenster, z.B. `onkey()`, `register_shape()` oder `listen()`. Vorsichtig müssen wir mit `onclick()` verfahren, weil dies der Name einer Turtle-Methode wie auch einer Methode von `screen` ist.

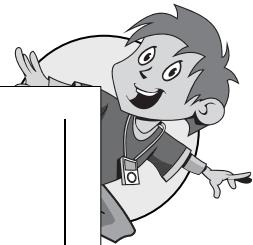
Wir wollen auf das ganze Moorhuhn-Spiel bezogene Dinge mit einer speziellen Klasse `MoorhuhnSpiel` verwalten. Wir werden sie dafür benutzen, das Grafik-Fenster und den Hintergrund zu erzeugen, die Hühner-Shapes zu registrieren, Meldungen auf den Bildschirm zu schreiben und vieles andere mehr.

Wie geben der Klasse `MoorhuhnSpiel` ein `screen`-Attribut, das Grafik-Fenster, gerade so groß wie das Hintergrundbild, und für die Ausgabe von Meldungen eine `schreiber`-Turtle als Attribut und eine Methode `melde()`, der man den Text als Argument übergibt, den diese anzeigen soll:

```
from turtle import Screen, Turtle

class MoorhuhnSpiel:
    """Kombiniert die Bestandteile des Moorhuhnspiels.
    """

    def __init__(self):
        self.screen = Screen()
        self.screen.setup(640, 480)
        self.screen.title("Moorhuhn")
        self.screen.bgpic("landschaft.gif")
        self.screen.register_shape("huhn01.gif")
        self.screen.register_shape("huhn02.gif")
        # Der Schreib-Gehilfe
        self.schreiber = Turtle(visible=False)
        self.schreiber.speed(0)
```



```
self.schreiber.penup()
self.schreiber.goto(-290, -220)
self.schreiber.pencolor("yellow")

def melde(self, txt):
    """Gibt Text txt im Grafik-Fenster aus.
    """
    self.schreiber.clear()
    self.schreiber.write(txt,
                        font=("Courier", 18, "bold"))

def run(self):
    self.melde("Start mit Leertaste!")
```

Wie du siehst, habe ich der Klasse MoorhuhnSpiel auch noch eine Methode `run()` verpasst, mit der das Spiel, oder eigentlich eine Spielserie, in Gang gesetzt wird. Bis jetzt liefert diese Methode erst nur einen Hinweis, wie das Spiel gestartet wird.

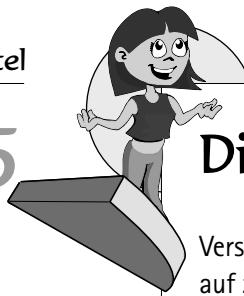
Damit können wir das »Hauptprogramm« folgendermaßen formulieren:

```
if __name__ == "__main__":
    spiel = MoorhuhnSpiel()
    spiel.run()
```

- Öffne vom IPI aus ein Editor-Fenster für eine Datei `moorhuhn_arbeit.py`. Schreibe einen Kopfkommentar und sichere die Datei im Verzeichnis kap15.
- Füge die `import`-Anweisung, die Klassendefinitionen von Moorhuhn-Spiel sowie den Hauptprogrammteil mit dem Aufruf zur Erzeugung eines MoorhuhnSpiel-Objektes und anschließend dem Aufruf der Methode `run()` wie in den obigen beiden Listings angegeben an.
- Schließe ein allenfalls offenes Grafik-Fenster und führe das Programm aus.

Es wird nun eine Instanz der Klasse MoorhuhnSpiel erzeugt. Ein Grafik-Fenster öffnet sich, und du siehst das Hintergrundbild und auch die Meldung in gelber Schrift. Es gibt aber noch keinerlei Spiel-Aktionen.

15



Die Spiellogik

Versuchen wir jetzt ansatzweise, die Spiellogik einzurichten. Das Spiel soll auf zwei Arten von Ereignissen reagieren:

- ❖ auf Drücken der Leertaste mit einem neuen Spiel, aber nur wenn nicht gerade ein Spiel im Gange ist; wir werden am Anfang eines Spiels die Leertaste deaktivieren und sie am Ende wieder aktivieren.
- ❖ auf Mausklick mit der linken Maustaste mit einem Schuss, aber nur wenn das Spiel noch im Gange ist, d. h. noch nicht alle Schüsse abgegeben wurden. Wir werden also am Anfang des Spiels den Bildschirm-Klick aktivieren und ihn am Ende des Spiels deaktivieren.

Das könnte so gehen:

```
class MoorhuhnSpiel:
    ...
    def schuss(self, x, y):
        """Wird bei Mausklick auf das Grafikfenster
        aufgerufen, wenn das Spiel läuft.
        """
        ...
    def spiel(self):
        self.screen.onkeypress(None, "space")
        self.screen.onclick(self.schuss)
        ### Spiel ausführen ###
        ...
        self.screen.onclick(None)
        self.screen.onkeypress(self.spiel, "space")

    def run(self):
        self.melde("Start mit Leertaste!")
        self.screen.onkeypress(self.spiel, "space")
        self.screen.listen()
```

➤ Füge diesen Code in die Klasse MoorhuhnSpiel ein und vergewissere dich, dass das Spiel ausführbar bleibt.

Beachte, dass an das Mausklick-Ereignis die Methode `self.schuss()` gebunden wird, die zwei (freie) Parameter, `x` und `y`, hat, wie es erforderlich ist – obwohl wir noch nicht wissen, ob wir diese brauchen werden. In gleicher Weise ist die parameterfreie Methode `self.spiel()` an den Tastendruck auf die Leertaste gebunden.



Solange `schuss()` und `spiel()` nichts tun, können wir nicht feststellen, ob das funktioniert. Also fügen wir ein Minimum an Aktion hinzu: Wir zählen die Schüsse. Nach fünf Schüssen soll das Spiel aus sein. Ein paar Kommentare helfen uns sicher, im Folgenden den Überblick nicht zu verlieren.

```
def schuss(self, x, y):
    """
    ...
    """

    self.schuesse = self.schuesse + 1
    self.melde("SCHUSS {0}".format(self.schuesse))

def spiel(self):
    # Initialisierung
    self.screen.onkeypress(None, "space")
    self.screen.onclick(self.schuss)
    self.melde("SPIEL LÄUFT!")
    self.schuesse = 0

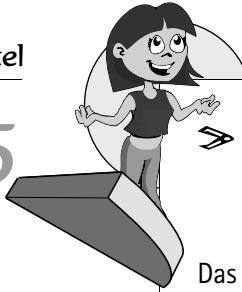
    # Ausführung
    while self.schuesse < 5:
        print("*", end=" ") # damit etwas Zeit vergeht
        self.screen.onclick(None)

    # Abschluss und Ergebnis
    self.screen.onkeypress(self.spiel, "space")
    self.melde("DAS SPIEL IST AUS! - Leertaste!!!")
```

Wie verläuft `spiel()` in dieser Fassung? Zuerst erscheint die Meldung »Spiel läuft!« Das Attribut `self.schuesse` dient der Spielkontrolle: Es wird zunächst auf 0 gesetzt und dann mit jedem Schuss erhöht. Wenn es 5 erreicht hat, ist das Spiel – mit einer dementsprechenden Meldung – aus. Und: Du kannst auch gleich ein neues Spiel mit der Leertaste starten!

- Füge diese Codezeilen für die Methoden `spiel()` und `schuss()` der Klasse `MoorhuhnSpiel` ein.
- Speichere das Programm und teste es: Starte es durch Drücken der Leertaste und schieße es durch Klicken mit der Maus in das Grafik-Fenster.

Wie ist der Test verlaufen? Hast du dich überzeugt, dass die Leertaste während des Spiels nicht aktiv ist? Und die Maus nur während des Spiels, aber nicht vorher und auch nicht nachher? Bei mir war alles o.k.



⇒ Wenn auch bei dir alles funktioniert, speichere eine Kopie als `moorhuhn01.py` ab. Wenn in der weiteren Entwicklung etwas schiefgehen sollte, kannst du auf diese Version zurückgreifen.

Das Spiel funktioniert zwar bis jetzt richtig, aber mit den Schüssen gibt es natürlich noch ein kleines Problem: Es fehlt ihnen das Ziel: die Moorhühner. Deshalb sollten wir uns nun diesen zuwenden.

Die Klasse »Huhn«

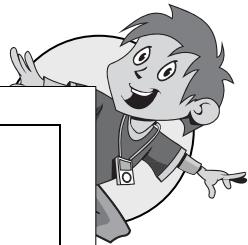
Unsere technischen Vorbereitungen vom Anfang des Kapitels legen nahe, die Klasse Huhn von der Klasse Turtle abzuleiten. Damit können die Hühner alles, was Turtles auch können.

Wenn wir Moorhühner im Computerprogramm abbilden oder modellieren, interessieren uns natürlich nur die Eigenschaften, die für das Spielgeschehen eine Rolle spielen. Beispielsweise interessieren wir uns nicht für das Alter der Moorhühner.

- ❖ Welche *Eigenschaften* der Moorhühner sind also für das Moorhuhn-Spiel wichtig? Erstens haben sie eine Geschwindigkeit – gegeben durch zwei Zahlen `vx` und `vy`. Zweitens können sie von einer Kugel getroffen werden und daher tot sein. Das können wir mit einer booleschen Variablen als *Attribut* beschreiben.
- ❖ Welche *Fähigkeiten* der Moorhühner sind für das Moorhuhn-Spiel wichtig? Ein Moorhuhn muss links am Grafikfensterrand erscheinen. Dort soll es eine aus passenden Bereichen zufällig gewählte Position und Geschwindigkeit erhalten. Es wäre ja urlangweilig, wenn alle Hühner auf der gleichen Bahn flögen. Und sie sollen natürlich nicht tot sein. Wir können Moorhühner, die das Fenster verlassen, weiterverwenden, indem wir sie mit einer Methode `zumstart()` schnell und ungetragen zum linken Fensterrand bewegen.
- ❖ Ein Moorhuhn muss einen Bewegungsschritt ausführen und immer wieder wiederholen können. Dies werden wir mit einer *Methode* `schritt()` realisieren.
- ❖ Und dann ist da noch die Turtle-Shape wichtig. Bei unserem Spiel soll sie ein Bildchen aus einer gif-Datei sein. Das machen wir wie weiter vorn in der interaktiven Sitzung gezeigt.

Aus dieser Beschreibung ergibt sich zunächst folgende Klassendefinition von Huhn. (Die `import`-Anweisung gehört natürlich an den Anfang des Programms!)

Die Klasse »Huhn«



```
from turtle import Screen, Turtle
import random
# ...
class Huhn(Turtle):
    """Objekte dieser Klasse sind Moorhühner im
    Moorhuhn-Spiel. Sie 'fliegen' geradlinig und -
    wenn getroffen - auf einer Parabelbahn.
    """
    def __init__(self, bilddatei):
        Turtle.__init__(self, bilddatei)
        self.hideturtle()
        self.penup()
        self.speed(0)

    def zumstart(self):
        self.hideturtle()
        self.setpos(-340, random.randint(-120,120))
        self.vx = random.randint(6,11)
        self.vy = random.randint(-3,3)
        self.showturtle()
        self.tot = False

    def schritt(self):
        x, y = self.position()
        x = x + self.vx
        y = y + self.vy
        self.goto(x,y)
```

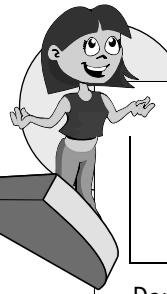
➤ Füge den Code der Huhn-Klasse in moorhuhn_arbeit.py ein.

Ob diese Methoden ausreichen, wird sich herausstellen, wenn wir in der MoorhuhnSpiel-Methode spiel() die Hühnersteuerung programmieren.

➤ Zunächst wollen wir jedoch im Konstruktor von MoorhuhnSpiel die Hühner für das Spiel erzeugen. Füge dort folgende Zeile ein:

```
class MoorhuhnSpiel:

    def __init__(self):
        ...
        self.schreiber.pencolor("yellow")
        # Die Moorhühner
```



```
self.huehner = (Huhn("huhn01.gif"),
                 Huhn("huhn02.gif"))
...
```

Damit hast du ein Tupel von zwei Moorhühnern in das Spiel eingebaut. Die sollten nun natürlich in der Methode `spiel()` auch gebraucht werden. Beim Spielanfang müssen sie zum Start gehen und dann `schritt()`-weise nach rechts fliegen und ...

```
def spiel(self):

    # Initialisierung
    self.screen.onkeypress(None, "space")
    self.screen.onclick(self.schuss)
    self.melde("SPIEL LÄUFT!")
    self.schuesse = 0
    for huhn in self.huehner:
        huhn.zumstart()

    # Ausführung
    while self.schuesse < 5:
        for huhn in self.huehner:
            huhn.schritt()
        self.screen.onclick(None)
    ...
```

➤ Beachte, dass die Zeile mit dem `print()`-Aufruf gelöscht wurde. Wird nicht mehr gebraucht! Speichere das Programm und führe es auch aus.

Es funktioniert zwar ohne Fehlermeldungen, unsere Hühner fliegen eifrig, kommen aber leider nicht von selbst zum Start zurück, wenn sie rechts, oben oder unten verschwunden sind. Das müssen wir reparieren. Dazu müssen wir aber die Hühner fragen können, ob sie das Grafik-Fenster verlassen haben. Wir verpassen ihnen noch eine Methode `raus()`, die uns `True` zurückgibt, wenn sie aus dem Grafik-Fenster raus sind, und `False` sonst:

Schießen und treffen



```
class Huhn(Turtle):
    ...
    def raus(self):
        x, y = self.position()
        return x > 340 or abs(y) > 250
```

Wenn `abs(y)`, der Betrag der y-Koordinate des Huhns größer als 250 ist, ist es entweder oben oder unten raus!

Hier siehst du erstmals, dass logische Ausdrücke wie `x > 340` und `abs(y) > 250` mit logischen Operatoren verbunden werden können. Verwendest du `or` (oder), dann muss mindestens einer der beiden Teilausdrücke wahr sein, damit der ganze Ausdruck wahr ist. Verwendest du `and` (und), dann müssen es beide sein.

Damit ergänzen wir die `while`-Schleife in `spiel()`:

```
# Ausführung
while self.schuesse < 5:
    for huhn in self.huehner:
        huhn.schritt()
        if huhn.raus():
            huhn.zumstart()
    ...

```

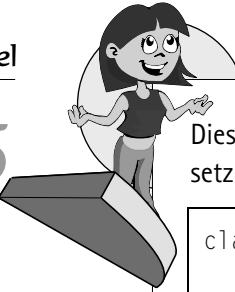
⇒ Teste das Spiel. Wenn es funktioniert, speichere eine Kopie als `moorhuhn02.py` ab.

Es sollte funktionieren wie die Fassung 1: Nach Druck auf die Leertaste startet das Spiel. Du schießt fünf Mal und das Spiel ist aus. Währenddessen werden nun nicht mehr Sternchen ausgegeben, sondern die beiden Moorhühner fliegen unaufhörlich über das Spielfeld, unberührt von deinen Mausklicks. Das wollen wir nun ändern.

Schießen und treffen

Zunächst sorgen wir dafür, dass die Hühner die Mausklicks spüren:

- ❖ Unsere Moorhühner müssen sterben, wenn sie einen Treffer erhalten. Eine Methode `getroffen()`, die an Mausklicks auf die Hühner gebunden wird, kann das erledigen.



Diese Methode `getroffen()` der Klasse `Huhn` ist besonders einfach: Wir setzen das Attribut `tot` auf wahr, also True.

```
class Huhn(Turtle):
    ...
    def getroffen(self, x, y):
        self.tot = True
```

In der `MoorhuhnSpiel.spiel()`-Methode können wir diese Methode an Mausklicks auf die Hühner (Turtles) binden:

```
def spiel(self):
    ...
    for huhn in self.huehner:
        huhn.zumstart()
        huhn.onclick(huhn.getroffen)

    # Ausführung:
    ...
```

Jetzt können wir – wie weiter vorn gezeigt – die toten Hühner auf Parabelbahnen zu Boden fallen lassen. Das erfordert eine Einfügung in der `Huhn.schritt()`-Methode, die die Fallgeschwindigkeit (in y-Richtung) erhöht:

```
def schritt(self):
    if self.tot:
        self.vy = self.vy - 0.25
    x, y = self.position()
    x = x + self.vx
    y = y + self.vy
    self.goto(x,y)
```

➤ Füge die neuen Codezeilen aus den letzten Listings ein und teste das Spiel. Stürzen die getroffenen Hühner zu Boden? Vielleicht bist du (so wie ich) nicht besonders geschickt und die Vögel fliegen dir zu schnell? Dann nimm für `self.vx` und `self.vy` in `schritt()` kleinere Zahlenwerte.

Es sollte nun noch mitgezählt werden, wie viele Hühner getroffen wurden. Dazu führen wir in der Klasse `MoorhuhnSpiel` ein Attribut `treffer` ein, das zu Beginn eines Spiels auf null gesetzt und jedes Mal, wenn ein Huhn

Schießen und treffen



getroffen wird, um eins erhöht wird. Damit kann am Ende die Anzahl der erzielten Treffer ausgegeben werden.

- ❖ Der erste Teil dieser Ergänzung ist äußerst einfach: Wir setzen im Initialisierungs teil von `spiel()` ein Attribut `treffer` auf den Wert 0.

```
...  
self.schuesse = 0  
self.treffer = 0  
for huhn in self.huehner:  
    ...
```

- ❖ Nun müssen aber auch noch die getroffenen Hühner dieses Attribut der Klasse Moorhuhnspiel erhöhen. Dazu müssen sie diese Klasse kennen. Das erreichen wir so, dass wir ihnen schon bei der Erzeugung das Moorhuhnspiel selbst – eigentlich: einen Verweis darauf – als Argument übergeben:

```
class Huhn(Turtle):  
    """...  
    """  
    def __init__(self, bilddatei, moorhuhnspiel):  
        Turtle.__init__(self, shape=bilddatei)  
        self.spiel = moorhuhnspiel  
        self.hideturtle()
```

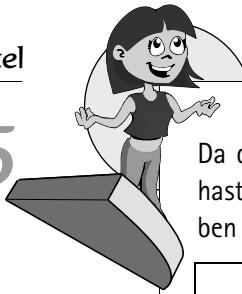
Nun müssen wir aber auch noch tatsächlich das MoorhuhnSpiel *selbst* als Argument bei der Erzeugung der Hühner einsetzen:

```
class MoorhuhnSpiel:  
    def __init__(self):  
        ...  
        # Die Moorhühner  
        self.huehner = (Huhn("huhn01.gif", self),  
                       Huhn("huhn02.gif", self))
```

Insgesamt nicht viel Arbeit, aber sehr wirksam. Nun hat jedes Huhn über `self.spiel` Zugriff auf alle Attribute des Moorhuhnspiels. Wir nützen dies aber nur für das Attribut `treffer`. Eine einfache und naheliegende Idee ist folgende:

```
def getroffen(self, x, y):  
    self.tot = True  
    self.spiel.treffer = self.spiel.treffer + 1
```

15



Da du als Spieler sicher gerne eine Rückmeldung hättest, ob du getroffen hast, ändern wir auch noch die Meldung, die nach jedem Schuss ausgegeben wird:

```
def spielstand(self):
    return "Treffer/Schüsse: {0}/{1}".format(
        self.treffer, self.schuesse)

def schuss(self, x, y):
    """
    ...
    """
    self.schuesse = self.schuesse + 1
    self.melde(self.spielstand())

def spiel(self):
    ...
    self.screen.onkeypress(self.spiel, "space")
##    self.melde("DAS SPIEL IST AUS! - Leertaste!!!")
```

➤ Führe diese Änderungen des Codes durch. Beachte, dass für den Spielstand-String eine eigene kleine Methode `spielstand()` verwendet werden wird, und auch, dass ich die Meldung im Abschlussteil von `spiel()` auskommentiert habe, damit die von `schuss()` stehen bleibt.

Zunächst scheint die Sache zu funktionieren. Doch wenn du auf ein Huhn mehrfach schießt und auch mehrmals triffst, bevor es das Grafik-Fenster verlassen hat, werden alle diese Treffer gezählt. Das soll nicht sein! Auch Hühner sterben nur einmal!

Doch auch dieses Problem ist wieder einfach zu lösen: Wenn ein Huhn schon tot ist, soll es auf einen Schuss einfach nicht mehr reagieren:

```
def getroffen(self, x, y):
    if self.tot:
        return
    self.tot = True
    self.spiel.treffer = self.spiel.treffer + 1
```

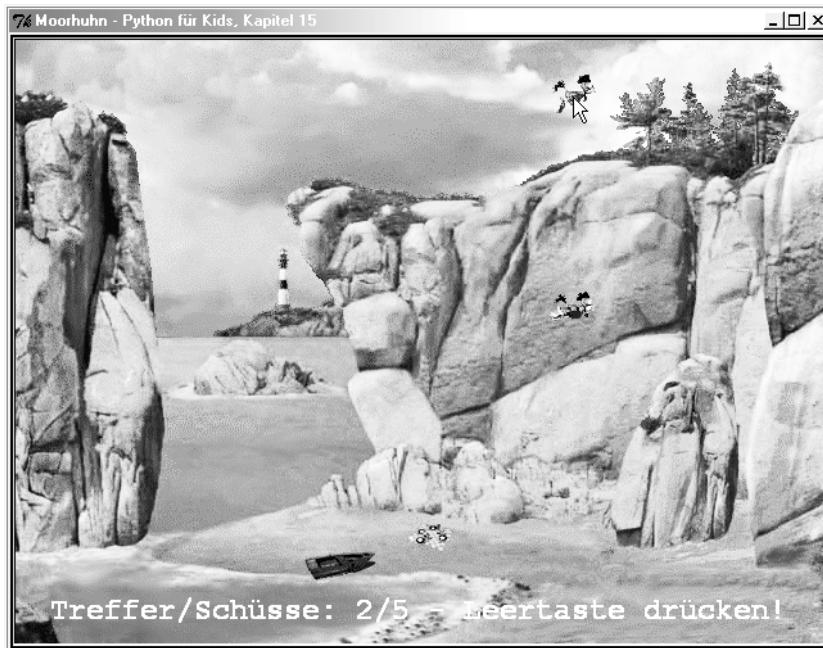
➤ Füge diese Codezeilen ein und teste das Spiel. Du wirst bemerken, dass nun Treffer nur noch einmal gezählt werden.

Fine tuning Moorhuhn

Ändern wir jetzt noch die Ausgabe am Ende des Spiels so, dass ein Hinweis angefügt wird, die Leertaste zu drücken. Dazu ist bloß die vorhin auskommentierte Meldung im Abschlussteil der Methode `spiel()` durch folgende zwei Anweisungen zu ersetzen:

```
# Abschluss und Ergebnis  
self.screen.onkeypress(self.spiel, "space")  
self.melde(self.spielstand() +  
           " - Leertaste drücken!")
```

- Teste das Spiel. Wenn es funktioniert, speichere eine Kopie als `moorhuhn03.py` ab.



So könnte ein Spiel ausgehen (ich bin kein besonders guter Spieler).

Du hast nun bereits ein voll funktionsfähiges Moorhuhn-Spiel programmiert. Einige Kleinigkeiten lassen aber noch zu wünschen übrig. Damit wollen wir uns im Folgenden befassen.

Fine tuning Moorhuhn

Du hast wahrscheinlich beobachtet, dass bei Spielende die letzten beiden verbleibenden Hühner plötzlich über der Landschaft anhalten – weil fünf Schüsse abgegeben worden sind. Besser wäre, wenn sie noch weiter, aus dem Grafik-Fenster raus, flögen. Das lässt sich leicht machen:



Das programmieren wir mit zwei pythonesken Feinheiten.

- ❖ Erstens *list-comprehension*: [huhn.raus() for huhn in self.huehner] ist eine Liste. Weil raus() eine Huhn-Methode ist, die True oder False liefert, enthält diese Liste nur Wahrheitswerte, und zwar für jedes Huhn einen.
- ❖ Zweitens hat Python eine Funktion all() eingebaut, die eine Liste als Argument übernehmen kann und nur dann True zurückgibt, wenn alle Listenelemente wahr sind. (all funktioniert auch mit anderen Sequenzen und dynamischen Wertevorräten.)

Es ist immer gut, sich solche neuen Dinge mit dem IPI anzusehen, um sich mehr Klarheit zu verschaffen. Das solltest du dir für deine zukünftige Praxis als Python-Programmierer merken. Hier zum Beispiel:

➤ Mach mit!

```
>>> 3 < 6 # boolescher Ausdruck
True
>>> [i < 2 for i in range(4)] # list comprehension
[True, True, False, False]
>>> all([i < 2 for i in range(4)])
False
>>> [i < 5 for i in range(4)]
[True, True, True, True]
>>> all([i < 5 for i in range(4)])
True
```

Zuerst deaktivieren wir die onclick()-Bindung, damit sie auf eventuelle weitere Mausklicks nicht mehr reagieren (#1). Solange nicht alle Hühner raus sind (#2), lassen wir sie einfach weitere Schritte machen (#3):

```
def spiel(self):
    ...
    # Ausführung
    while self.schuesse < 5:
        ...
            huhn.zumstart()
        for huhn in self.huehner: #1
            huhn.onclick(None)
        self.screen.onclick(None)
        while not all([huhn.raus() for huhn in self.huehner]): #2
            for huhn in self.huehner: #3
                huhn.schritt()
```

Konstanten

Damit ist dieses Problem schon erledigt. Wenn du magst, kannst du gleich hinter der ersten while-Schleife in `spiel()` noch folgende Anweisung einfügen:

```
huhn.zumstart()  
self.melde(self.spielstand() +  
           " - DAS SPIEL IST AUS!")  
# Nix geht mehr!
```

Sie wird angezeigt, bis alle Hühner das Grafik-Fenster verlassen haben. So erfährt die Spielerin, dass weiterzuschießen sinnlos ist.

- Führe die beschriebenen Änderungen aus und teste dann das Programm. Verschwinden die Hühner am Ende aus der Bildfläche? Wenn ja, speichere eine Kopie als `moorhuhn04.py` ab.

Konstanten

Auf meinem Rechner ist das Spiel (für mich) ziemlich schwer zu spielen. Die Hühner fliegen recht schnell. Auf deinem Rechner kann das ganz anders aussehen – vielleicht sind sie dir sogar zu langsam. Vielleicht möchtest du auch mehr als fünf Schüsse pro Spiel haben.

Diese Punkte betreffen grundlegende Einstellungen des Spiels. Es ist sinnvoll, solche Einstellungen mit Hilfe von *Konstanten* festzulegen. Konstanten sind Namen, die auf fixe Werte, in diesem Fall Zahlenwerte, verweisen. In Python ist es üblich, Konstanten in Großbuchstaben zu schreiben und ihre Werte am Programmanfang, gleich nach den `import`-Anweisungen, festzulegen.

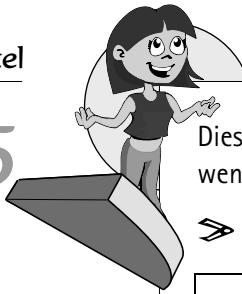
In der letzten Fassung unseres Spiels werden pro Spiel 5 SCHUESSE abgegeben. Und das Spieltempo wird letztlich durch die Attribute `vx` und `vy` der Hühner festgelegt. Wählen wir die beiden etwa halb so groß, läuft das Spiel viel langsamer ab. Dies wollen wir nun mit einer Konstanten TEMPO regeln. Und zwar wie folgt:

- Füge in `moorhuhn_arbeit.py` nach den `import`-Anweisungen und vor der ersten Klassendefinition folgende Wertzuweisungen ein:

```
SCHUESSE = 5  
TEMPO = 1
```



15



Diese Konstanten müssen nun an den passenden Stellen im Programm verwendet werden:

➤ In der Methode `zumstart()` verwende TEMPO wie folgt:

```
self.vx = randint(6,11) * TEMPO
self.vy = randint(-3,3) * TEMPO
```

➤ Ergänze in `schrift()`:

```
if self.tot:
    self.vy = self.vy - 0.25 * TEMPO
```

Die Anzahl der Schüsse eines Spieldurchgangs wird nur an einer Stelle in der Methode `spiel()` gebraucht.

➤ Ersetze im Kopf der ersten while-Schleife in `spiel()` die 5 durch SCHUESSE:

```
...
# Ausführung
while self.schuesse < SCHUESSE:
    for huhn in self.huehner:
        ...
        ...
```

Nun kannst du das Programm leicht – bloß durch Änderungen von zwei Zahlenwerten am Anfang des Programms – an deine Bedürfnisse und an deinen Rechner anpassen.

➤ Probiere einmal verschiedene Werte für TEMPO etwa im Bereich von 0.5 bis 3 aus.

Cursor

Zum Zielen und Schießen würde sich ein anderer Mauscursor als der standardmäßig eingestellte Pfeil besser eignen. Doch da stoßen wir an die Grenzen von `turtle`. Dieses Modul hat keine Mittel zur Veränderung des Cursors. Aber es hat Mittel, um über seine eigenen Grenzen zu gelangen. Das zeige ich dir an diesem Beispiel – wieder ein kleines Tor in die Welt von Tkinter.

Die Zeichenfläche, auf der die Turtles leben, ist – wie du schon aus dem Kapitel über `scribble` weißt – ein Tkinter-Canvas. Dieses Canvas-Objekt



hat neben der `postscript()`-Methode noch eine ganze Latte von anderen Methoden, über die du in einer Tkinter-Dokumentation nachlesen kannst. Eine davon heißt `config()` und kann zur Festlegung der Cursor-Gestalt verwendet werden.

Tkinter-Dokumentationen findest du auf der Buch-CD, von der Buch-Website aus oder durch Googeln. Hier ist zum Beispiel eine Liste der verfügbaren Cursor-Formen zu finden:

<http://infohost.nmt.edu/tcc/help/pubs/tkinter/cursors.html>

Den Zugriff auf das Canvas-Objekt, das `screen` verwendet, ermöglicht die `screen-Methode` `getcanvas()`.

➤ Schließe alle offenen Grafik-Fenster und mach mit! (In einem neu gestarteten IPL musst du zuerst `Turtle` und `Screen` aus `turtle` importieren.)

```
>>> t = Turtle()  
>>> screen = Screen()  
>>> leinwand = screen.getcanvas()  
>>> leinwand.config(cursor="X_cursor")
```

Wenn du nun die Maus über die Zeichenfläche bewegst, nimmt der Mauscursor die Gestalt eines X an. So können wir mit einer schlauen Anweisung den Mauscursor für das Moorhuhn-Fenster umstellen.

➤ Füge folgende Anweisung in den Konstruktor von `MoorhuhnSpiel` ein, am besten ans Ende der Gruppe der Anweisungen mit `self.screen`:

```
self.screen.getcanvas().config(cursor="X_cursor")
```

Töne

Um ein besseres Action-Gefühl zu erzeugen, braucht ein Spiel Töne! Python kommt mit einem Modul `winsound` daher, der auf Systemen mit Windows als Betriebssystem erlaubt, `*.wav`-Klangdateien abzuspielen. Zum Abschluss will ich dir zeigen, wie man das für unser Spiel nutzt. (Benutzer anderer Betriebssysteme können sich etwa `Pygame`, einen Werkzeugkasten zum Programmieren von Spielen, installieren und erhalten damit auch auf einfache Weise Zugriff auf die Sound-Komponenten ihres Computers. Auf der Buch-Website <http://python4kids.net> findest du mehr Informationen dazu.)

15



Das Modul winsound hat eine Funktion PlaySound(). Diese Funktion verlangt zwei Argumente: den Dateinamen der Sounddatei, die gespielt werden soll, und eine der im Modul winsound definierten Konstanten, die darüber entscheiden, wie der Sound gespielt werden soll. Wir wollen, dass das Abspielen des Klanges das Spiel nicht unterbricht oder anhält, und dafür ist die Konstante SND_ASYNC zuständig. Das wollen wir jetzt ausprobieren:

- Vergewissere dich, dass die Klangdateien `getroffen.wav`, `daneben.wav`, `jubel.wav` und `applaus.wav` im aktuellen Arbeitsverzeichnis `C:\py4kids\kap15` vorhanden sind und mach mit!

```
>>> import winsound
>>> winsound.PlaySound("applaus.wav", winsound.SND_ASYNC)
>>> winsound.PlaySound("daneben.wav", winsound.SND_ASYNC)
```

Ein bisschen viel zu schreiben. Daher definieren wir uns zuerst eine Methode `klang()` der Klasse MoorhuhnSpiel:

```
def klang(self, soundfile):
    winsound.PlaySound(soundfile, winsound.SND_ASYNC)
```

- Ergänze die `import`-Anweisungen:

```
import random, winsound
```

- Füge `klang()` als letzte Methode vor der Methode `run()` der Klasse MoorhuhnSpiel hinzu.

Für die Dateinamen der Klangdateien führen wir Konstanten ein.

- Schreibe die folgenden Konstanten, diesmal sind es Strings, unter die zwei schon vorhandenen am Anfang der Programmdatei:

```
GETROFFEN = "getroffen.wav"
DANEBEN = "daneben.wav"
GUT = "jubel.wav"
NAJA = "applaus.wav"
```

Das ist vor allem nützlich, wenn du irgendwo Klangdateien entdeckst (oder gar selbst erzeugst), die dir besser gefallen und die du statt diesen hier einsetzen willst. Dann brauchst du nur den Wert der entsprechenden Konstanten abzuändern!

Töne



Jetzt brauchen wir einfach nur noch Aufrufe der Methode `klang()` an den passenden Stellen:

Erstens im Spiel: (wenig) Applaus, wenn wenig Hühner getroffen wurden, sonst Jubel.

➤ Füge ans Ende der Methode `spiel()` folgende Anweisung an:

```
trefferrate = self.treffer / self.schuesse
if trefferrate > 0.55:
    self.klang(GUT)
else:
    self.klang(NAJA)
```

Und dann noch die Klänge für die Schüsse. Sie sollen verschieden klingen, je nachdem, ob ein Huhn getroffen wurde oder der Schuss danebenging. Hier ist ein klein wenig Tricksen angesagt:

Der entsprechende Aufruf von `klang()` gehört in die Methode `schuss()`. Wir wählen – als Voreinstellung – die Klangdatei DANEBEN und schreiben eine entsprechende Anweisung in den Konstruktor von `MoorhuhnSpiel`.

```
self.huehner = (Huhn("huhn01.gif", self),
                 Huhn("huhn02.gif", self))
self.schussklang = DANEBEN
```

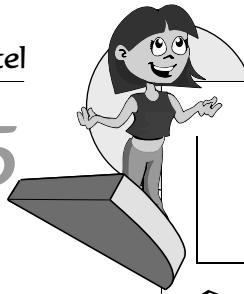
Nur wenn der Schuss (mindestens) ein Huhn traf, ersetzen wir sie durch GETROFFEN. Diese Änderung kann aber nur in der `getroffen()`-Methode des Huhns vorgenommen werden.

```
def getroffen(self, x, y):
    if self.tot:
        return
    self.tot = True
    self.spiel.schussklang = GETROFFEN
    self.spiel.treffer = self.spiel.treffer + 1
```

Nun muss der Klang noch gespielt und auf jeden Fall wieder auf den Standardwert DANEBEN gesetzt werden.

```
def schuss(self, x, y):
    """
    ...
    """
    self.schuesse = self.schuesse + 1
```

15



```
self.klang(self.schussklang)
self.melde(self.spielstand())
self.schussklang = DANEBEN
```

- Füge diese Änderungen in die Methoden `getroffen()` und `schuss()` ein.
- Speichere das Programm, führe es aus, beseitige Fehler, falls vorhanden, und teste es gründlich.
- Passe die Werte der Konstanten `TEMPO` und `SCHUESSE` deinen Bedürfnissen an. Zum Testen waren ja fünf Schüsse recht praktisch, aber bei einem echten Spiel werden so um die 20 wohl angemessener sein.
- Speichere eine Kopie des Programms als `moorhuhn05.py` ab.

Anmerkung: Dass das Treffer-Zählen und Klang-Abspielen richtig funktioniert, hat eine wichtige Voraussetzung, über die ich mich nicht lange ausgelassen habe: Ein Mausklick auf ein Huhn löst zwei Ereignisse aus: das Klick-Ereignis auf das Huhn (die Turtle) und das Klick-Ereignis auf das »darunter liegende« Grafik-Fenster. Entscheidend ist hier, dass die Sache in Tkinter so eingerichtet ist, dass das Klick-Ereignis auf das Huhn (`getroffen!`) zuerst verarbeitet wird. Nur deshalb weiß `schuss()` (danach), dass ein Treffer vorliegt.

Moorhuhn als selbstständig ausführbares Programm

Du hast nun ein schönes Stück Arbeit geleistet und willst vielleicht manchmal Moorhuhn spielen, ohne das Script jedes Mal in die IDLE zu laden. Es wäre doch angenehm, es einfach durch Doppelklick im Explorer oder auf ein Icon am Desktop starten zu können. Das ist prinzipiell möglich, wenn auf deinem Rechner Python installiert ist. (Das dürfte vermutlich der Fall sein.)

- Versuche, das Programm `moorhuhn05.py` durch Doppelklick zu starten.

Das Ergebnis ist enttäuschend. Zwar geht das Grafik-Fenster auf, doch ehe du dich versiehst, hat es sich auch schon wieder geschlossen.

Erklärung: Damit ein ereignisgesteuertes Programm laufen kann, muss eine Schleife in Gang gesetzt werden, die das Eintreffen von Ereignissen abwar-



tet, diese Ereignisse registriert und dann die entsprechenden Aktionen ausführt. Wenn du ein solches Programm mit dem IPI-TURTLEGRAFIK entwickelst, dann läuft eine solche Schleife bereits. Sie steuert die IDLE selbst, die ja auch ein ereignisgesteuertes Python-Programm ist. Und sie steuert, wenn du es mit **F5** ausführst, auch dein Programm.

Wenn du nun dein Moorhuhn-Programm für sich ausführst, wird ein MoorhuhnSpiel-Objekt erzeugt und deshalb der Code des Konstruktors der Klasse MoorhuhnSpiel ausgeführt. Doch dann wird das Programm sang- und klanglos beendet. Das Fenster geht wieder zu. Um das zu verhindern, musst du ans Ende des Hauptprogramms einen Aufruf der Funktion `mainloop()` stellen, die vom `screen`-Objekt als Methode zur Verfügung gestellt wird.

- Schließe die IPI-SHELL und alle zugehörigen Editor-Fenster.
- Starte diesmal IDLE(PYTHON GUI) und lade wieder (über den Menüpunkt FILE|RECENT FILES) `moorhuhn_arbeit.py` in ein Editor-Fenster.
- Speichere sofort das Programm unter dem Namen `moorhuhn.py` ab.
- Ergänze die `run()`-Methode um den Aufruf von `mainloop()`:

```
def run(self):  
    self.melde("Start mit Leertaste!")  
    self.screen.onkeypress(self.spiel, "space")  
    self.screen.listen()  
    self.screen.mainloop()
```

- Führe das Programm durch Drücken der Taste **F5** aus.

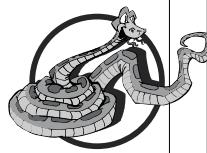
In der IDLE(PYTHON GUI) werden Programme nicht im selben Prozess wie die IDLE selbst ausgeführt, sondern in einem eigenen Unterprozess (*subprocess*). Daher braucht ein ereignisgesteuertes Grafikprogramm hier einen eigenen `mainloop()`-Aufruf. In dieser Form wird es aber im IPI nicht korrekt laufen, weil sich dann zwei mainloops in die Quere kommen.

Dafür kannst du es nun auch als Stand-alone-Programm (bei vorhandenem Python-Interpreter) ausführen.

- Starte vom Windows-Explorer aus durch Doppelklick das Programm `moorhuhn.py` im Verzeichnis `C:\py4kids\kap15`.

Du erkennst: Ein schwarzes Eingabeaufforderungsfenster geht auf. Dort läuft der Python-Interpreter (sieh den Fenstertitel an) und führt `moorhuhn.py` aus. Dabei öffnet sich das Grafik-Fenster für das Moorhuhnspiel.

15



Auf das Eingabeaufforderungsfenster können wir eigentlich verzichten, oder? Das erreichst du auch leicht:

- ⇒ Ändere den Dateinamen `moorhuhn.py` ab auf `moorhuhn.pyw`. Starte das Programm nochmals durch Doppelklick.

Stand-alone-Programme brauchen »mainloop()«

Grafik-Programme auf der Basis von `turtle` (oder von Tkinter), die alleine, ohne Entwicklungsumgebung, laufen sollen, brauchen als *letzte Anweisung* einen Aufruf der `screen()`-Methode `mainloop()`. Damit wird die »Hauptschleife« zur Verarbeitung anfallender Ereignisse in Gang gesetzt. Der Name `mainloop` kann von `turtle` auch als Funktion importiert werden. Programme mit eigener `mainloop()` laufen auch unter IDLE(PYTHON GUI), jedoch ist dort *interaktive* Grafik vom Python-Interpreter aus nicht möglich.

Es ist empfehlenswert, Grafik-Programme mit dem IPI-TURTLEGRAFIK zu entwickeln und mit IDLE(PYTHON GUI) zu testen.

Wenn für solche Programme die Dateierweiterung `*.pyw` anstelle von `*.py` verwendet wird, öffnet sich beim Start gleich das Grafik-Fenster.

Wenn du genauer wissen willst, warum zwei verschiedene Konfigurationen von IDLE überhaupt nötig sind, lies darüber in Anhang F (Seite 442) nach.

- ⇒ Spiele noch ein bisschen Moorhuhn. Wenn du keine Lust mehr hast, schließe das Moorhuhnspiel-Fenster.

Damit bist du am Ende dieses Buches angelangt. Du hast ein tolles Stück Arbeit geleistet, indem du es bis hierher durchgearbeitet hast. Ich gratuliere dir zum Abschluss ganz herzlich dazu!

Du hast jetzt den Einstieg ins Programmieren erfolgreich geschafft und beherrschst die Grundlagen von Python. Du bist jetzt gut darauf vorbereitet, dich in die umfangreiche Modul-Bibliothek von Python einzuarbeiten, die für die unterschiedlichsten Zwecke vorgefertigte Klassen bereitstellt.

Wenn du mit anderen Programmiersprachen arbeitest, wirst du bemerken, dass dir das, was du hier gelernt hast, auch dabei zugutekommt.

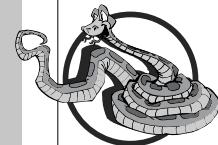
Außerdem hast du die Erfahrung gemacht, dass Programmieren mit Python Spaß macht. Vielleicht bekommst du schon morgen Lust, das Moorhuhn-Programm um neue Features zu erweitern. (Wie wäre es mit verschiedenen Schwierigkeitslevels?)

Hier kommen nun keine Zusammenfassung, keine Fragen und keine Aufgaben mehr, sondern nur gute Wünsche für deine Zukunft als Programmiererin oder als Programmierer. Ich bin sicher, die Chancen sind sehr groß, dass du für zukünftige Programmieraufgaben häufig Python benutzen wirst.

Clara Pythias letzter Tipp:

Schau von Zeit zu Zeit mal rein auf <http://python4kids.net>. Es gibt dort

- ❖ Infos zu dem Buch,
- ❖ heiße Tipps für häufig auftretende Probleme,
- ❖ Beispielprogramme zu alten und neuen Problemstellungen,
- ❖ weitere Lernmaterialien,
- ❖ Neuigkeiten aus der Python-Welt
- ❖ und nicht zuletzt kannst du uns dort auch deine Meinung schreiben über Python und über dieses Buch.



Anhang A



Python installieren (Windows, Linux, Mac OS X)

Um mit dem Buch *Python für Kids* optimal zu arbeiten, musst du ...

1. ... eine Standardinstallation von Python auf deiner Festplatte vornehmen.
2. ... mit einigen weiteren Schritten die Installation für die Arbeit mit dem Buch anpassen.

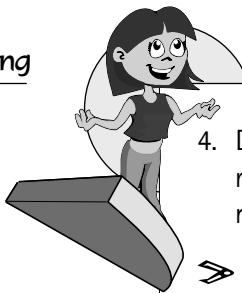
Installation von Python 3 unter Windows

1. Stelle sicher, dass mindestens 50 MB freier Speicherplatz auf der Festplatte C: vorhanden sind.
2. Angenommen, X: ist dein CD-ROM-Laufwerk. Lege die Buch-CD ins CD-ROM-Laufwerk und führe das Programm X:\Python31\python-3.1.1.msi aus. Akzeptiere alle Voreinstellungen, indem du etwa drei Mal auf die NEXT-Schaltfläche klickst und dann einmal auf FINISH. Das Ergebnis ist eine Standardinstallation von Python im Verzeichnis C:\Python31.
3. Teste die Installation, indem du START|ALLE PROGRAMME|PYTHON3.1|IDLE(PYTHON GUI) anklickst. Die Programmierumgebung IDLE wird gestartet und hinter der Bereitschaftsanzeige >>> blinks der Cursor. Gib dort Folgendes ein:

```
>>> print(1 + 1)
```

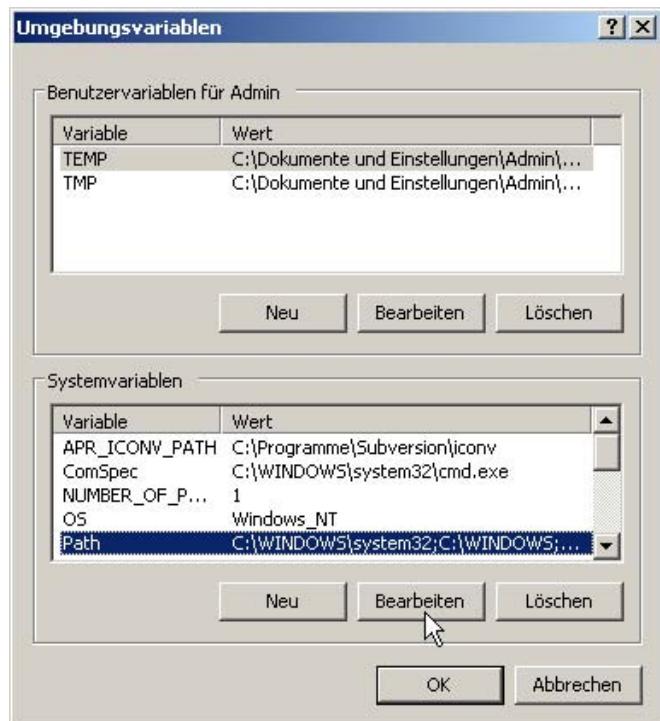
und drücke . Wenn nun als Ergebnis 2 erscheint, ist es gut so.

The screenshot shows a Windows application window titled "Python Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main window displays the Python 3.1.1 interpreter. It shows the version information: "Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32". Below this, it says "Type "copyright", "credits" or "license()" for more information.". A command line input is shown with the text "2" and the cursor at the end of the line. In the bottom right corner of the window, there is a status bar with "Ln: 5 Col: 4".



4. Damit Python auf deinem Computer in jeder Situation gefunden wird, muss das Python-Verzeichnis C:\Python31 nun in der Umgebungsvariablen PATH eingetragen werden. Dazu gehst du so vor:

- Klicke auf START|SYSTEMSTEUERUNG.
- Wechsle zur »klassischen Ansicht«, doppelklicke auf das Icon SYSTEM und dann auf die Karteikarte ERWEITERT.
- Klicke auf die Schaltfläche UMGEBUNGSVARIABLEN.
- Markiere im Feld SYSTEMVARIABLEN den Eintrag für PATH und klicke auf die Schaltfläche BEARBEITEN.



- Füge ans Ende des Eintrags von PATH ;C:\Python25 an. Klicke drei Mal auf die Schaltfläche OK.





- » Überprüfe, ob der Eintrag korrekt erfolgt ist, indem du mit START|ALLE PROGRAMME|ZUBEHÖR|EINGABEAUFLÖRUNG die Eingabeaufforderung öffnest. Tippe dort python ein.

```
C:\ Eingabeaufforderung - python
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Dokumente und Einstellungen\Python4Kids>python
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.

>>> -
```

Wenn nun der Python-Interpreter gestartet wird, dann hast du ein funktionierendes Python-System. Damit kannst du Python-Programme ausführen, wie es in Anhang B beschrieben ist.

- » Schließe das Fenster wieder.

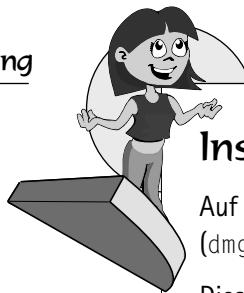
Installation von Python 3 unter Linux

Linux Distributionen kommen in der Regel mit einem installierten Python. Es wird aber noch einige Zeit dauern bis die Standardinstallation Python 3 ist.

Die meisten Linux-Distributionen haben ein Software-Installations-Programm, das auf eine umfangreiche Software-Sammlung im Internet zugreifen kann. Weil dies aber für die verschiedenen Distributionen recht unterschiedlich eingerichtet ist, kann das hier nicht für alle im Detail beschrieben werden.

Für Ubuntu 9.04, beispielsweise, ist dieses Programm die Synaptic-Paketverwaltung. Mit ihr kannst du zu der Zeit, wo ich das schreibe – im Oktober 2009 – Python 3.0 installieren. Das ist okay für dieses Buch. Wenn du das liest, gibt's möglicherweise schon Python 3.1.

Falls aber nicht: Auf der Buch-CD und der Buch-Website findest du den Quellcode für Python 3.1.1 und eine Erklärung, wie du ihn selbst kompilieren und installieren kannst.



Installation von Python 3 unter Mac OS X

Auf der Buch-CD findest du Python 3.1.1 für Mac OS X als Disk-Image (dmg-Datei): `python-3.1.1.dmg`.

Dies wird wie üblich installiert: Du öffnest im Finder `python-3.1.1.dmg`. Nach einem Doppelklick auf `Python.mpkg` wirst du durch die Installation geführt.

Anpassung der Installation für die Arbeit mit dem Buch

Für einige Betriebssysteme habe ich `setup`-Scripts erzeugt, die du auf der Buch-CD im Ordner `py4k_setup` oder auf der Buch-Website findest. Sieh nach, ob das Passende für dich dabei ist.

Für den Fall, dass du ein Skript, das auf deinem System fehlerfrei funktioniert, nicht findest, erkläre ich hier, wie die wichtigsten Schritte mit der Hand ausgeführt werden können. Diese sind:

- ⇒ Es müssen zwei Dateien und ein Verzeichnis von der Buch-CD an verschiedene Orte auf deiner Festplatte kopiert werden.
- ⇒ Es muss ein Desktop-Icon oder ein Menü-Eintrag erzeugt werden, mit dem die IDLE im passenden Modus für interaktive Turtle-Grafik gestartet wird.

Anpassung für Windows ausführen:

Im Folgenden soll X: dein CD-ROM-Laufwerk bezeichnen. Die Kopiervorgänge kannst du bequem im Windows-Explorer mit der Maus ausführen:

- ⇒ Von X:\turtlegrafik kopiere die Dateien `turtle.cfg` und `turtle_docstringdict_german.py` in das Verzeichnis Python31\ Lib. Damit wird das Turtlegrafik-Modul in der richtigen Konfiguration für die Arbeit mit dem Buch importiert.
- ⇒ Von X:\turtlegrafik kopiere das ganze Verzeichnis py4kids von der CD-ROM auf das Laufwerk C:\, oder besser noch in dein Home-Verzeichnis (EIGENE DATEIEN). Es enthält zu den einzelnen Buchkapiteln Unterverzeichnisse, in denen du deine Programme abspeichern wirst. In diesen sind aber auch Programme, die du zur Arbeit benötigst.

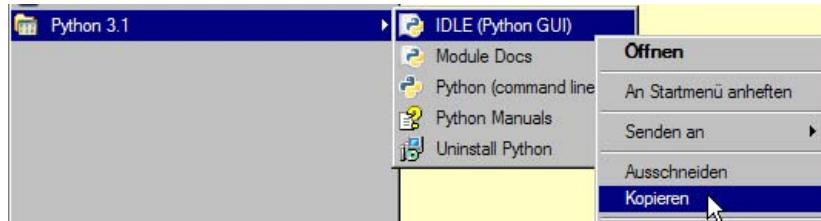
Anpassung der Installation für die Arbeit mit dem Buch



Nun wollen wir noch zwei Icons für IDLE auf dem Desktop einrichten.

Zuerst erstelle eine Verknüpfung zu IDLE (PYTHON GUI) auf dem Desktop:

- Klicke mit der *rechten* Maustaste auf START|ALLE PROGRAMME| PYTHON3.1|IDLE (PYTHON GUI) und dann im aufklappenden Kontextmenü auf KOPIEREN.

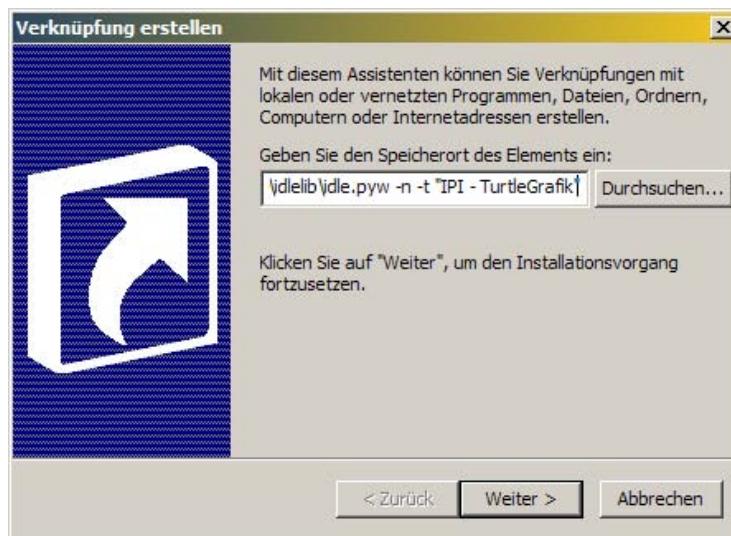


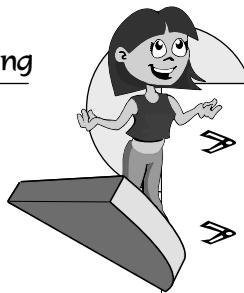
- Klicke mit der rechten Maustaste auf eine freie Stelle des Desktops und im aufklappenden Kontextmenü auf EINFÜGEN.
- Doppelklicke auf das neu eingefügt Icon. IDLE startet.

Zuletzt – aber für uns am wichtigsten – richten wir ein **Icon für die Arbeit mit Turtle-Grafik** ein:

- Klicke mit der rechten Maustaste auf eine leere Stelle des Desktops und wähle im aufklappenden Kontextmenü NEU|VERKNÜPFUNG .
- Es erscheint der Dialog VERKNÜPFUNG ERSTELLEN. Gib als Speicherort des Elements Folgendes ein (etwas länglich):

C:\Python31\pythonw.exe C:\Python31\Lib\idlelib\idle.pyw
-n -t "IPI - TurtleGrafik"



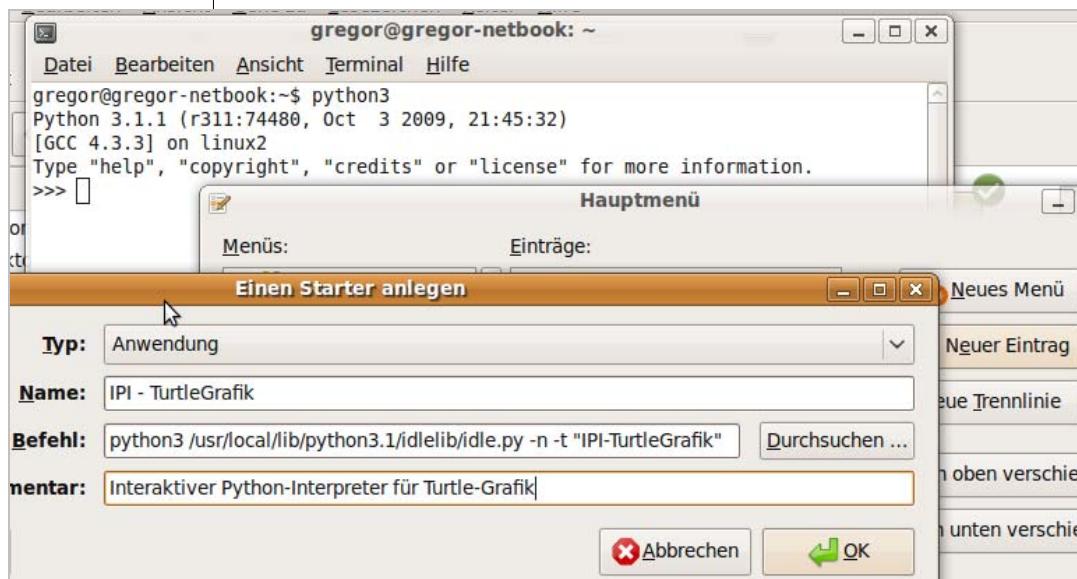


- ⇒ Klicke auf WEITER > und gib dann als Namen für die Verknüpfung IPI-TurtleGrafik ein. Klicke auf FERTIGSTELLEN.
- ⇒ Doppelklicke auf das Icon IPI – TURTLEGRAFIK. Das startet die IDLE wie in Kapitel 2 beschrieben.

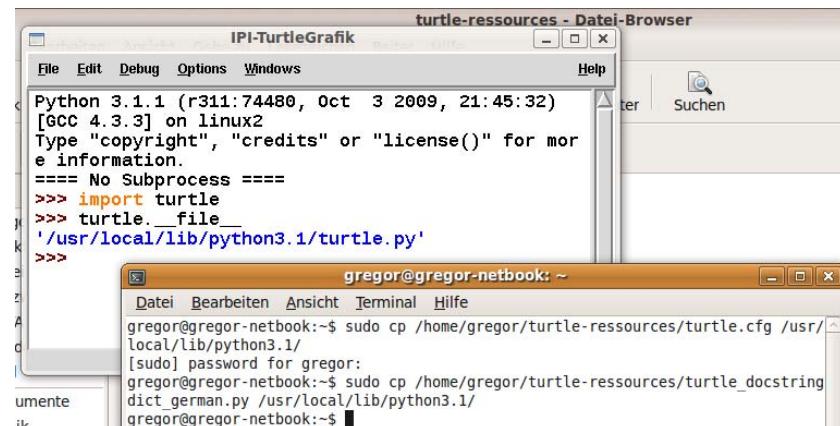
Anpassung für Linux ausführen – am Beispiel Ubuntu 9.04:

Nach der Installation von Python 3 kannst du in einem Terminal das Programm python3 starten. Wir wollen nun einen Menü-Eintrag erzeugen, der IDLE im -n Modus für interaktive Turtlegrafik startet.

- ⇒ Wähle den Menüpunkt SYSTEM|EINSTELLUNGEN|HAUPTMENÜ.
- ⇒ Erzeuge an passender Stelle im Menübaum (z.B. unter Entwicklung) einen neuen »Starter« wie folgt:



- ⇒ Jetzt sind noch die Dateien turtle.cfg und turtle_docstringdict_german.py aus dem Verzeichnis /turtlegrafik der CD in das Verzeichnis zu kopieren, in dem turtle.py wohnt. Wo das ist, erfährt man am leichtesten, indem man turtle importiert und nach dem __file__-Attribut fragt. Dann kann man das Kopieren in einem Terminal mit sudo ausführen.



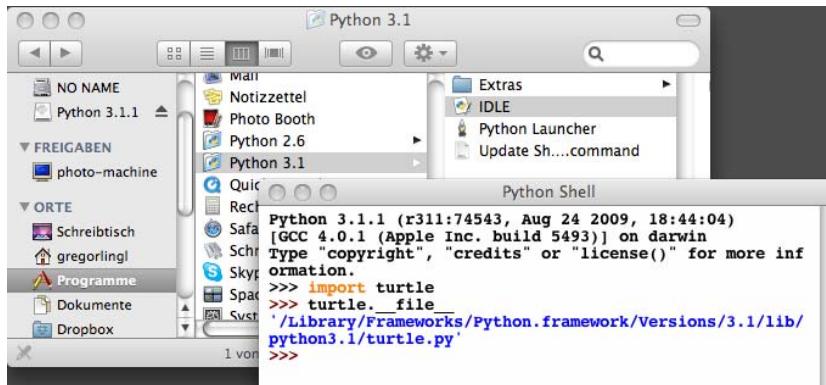
- Dann solltest du noch das Verzeichnis py4kids in dein home-Verzeichnis kopieren.

Ausführlichere Hinweise findest du auf der Buch-Website.

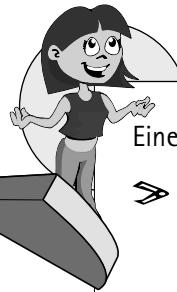
Anpassung für Mac OS X ausführen:

Da Mac OS X ein Unix-System ist, kannst du im Prinzip so vorgehen, wie im Linux-Abschnitt beschrieben. Nach der Installation findest du im Finder unter PROGRAMME|PYTHON 3.1 die IDLE

Auch hier erfährst du den Speicherort von turtle.py wie unter Linux:

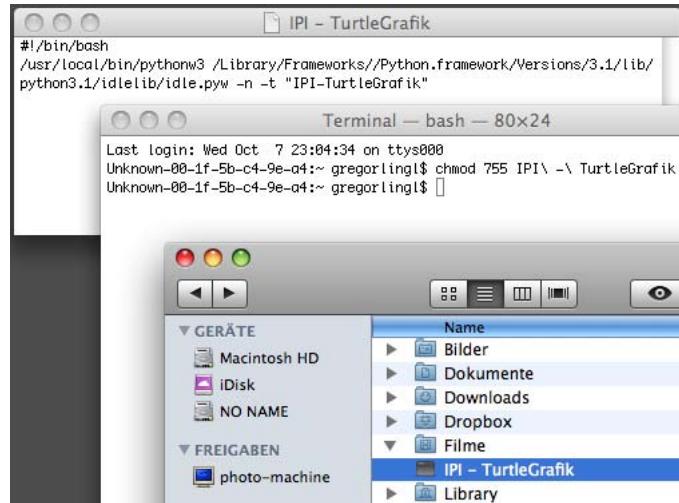


- Kopiere genau in dieses Verzeichnis die Dateien turtle.cfg und turtle_docstringdict.py von der Buch-CD.
- Ebenso solltest du das Verzeichnis py4kids in dein home-Verzeichnis kopieren und für die Arbeit mit dem Buch verwenden.



Eine IDLE ohne Subprocess bekommst du mit zwei Schritten:

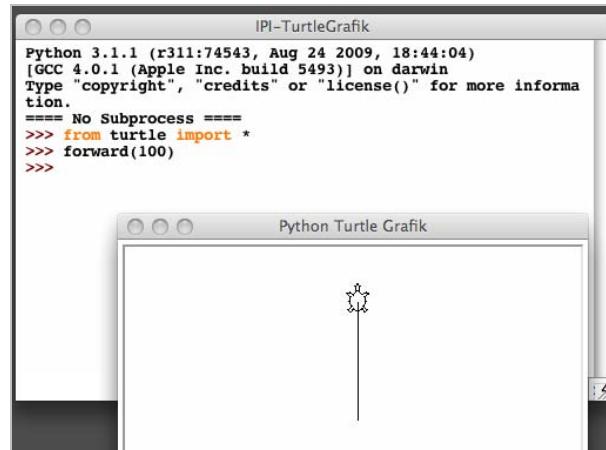
- Schreibe mit einem Texteditor ein bash-Skript, das die IDLE startet – wie in der Abbildung gezeigt. Speichere es in deinem persönlichen Verzeichnis als IPI-TURTLEGRAFIK ab.



- Mache dieses Skript in einem Terminal ausführbar mit dem UNIX-Befehl:

```
chmod 755 IPI\ -\ TurtleGrafik
```

Nun kannst du die IDLE im richtigen Modus für interaktive Turtlegrafik starten. Das gelingt im Finder durch Doppelklick auf dieses Skript:



Ausführlichere Hinweise findest du auf der Buch-Website.

Anhang B



Python-Programme ausführen

Für diesen Anhang setze ich voraus, dass du eine Standardinstallation von Python, wie in Anhang A beschrieben, durchgeführt und insbesondere auch das Python-Verzeichnis im PATH eingetragen hast.

Der Python-Interpreter ...

... kann auch ohne die Entwicklungsumgebung IDLE über das Startmenü – START|ALLE PROGRAMME|PYTHON3.1|PYTHON(COMMAND LINE) – gestartet werden.

Daraufhin meldet sich der Python-Interpreter mit einem Eingabe-Prompt >>> genauso wie in der Python Shell der IDLE. Er kann auch genauso benutzt werden:

```
Python (command line)
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> print("31 / 4 =", 31 / 4)
31 / 4 = 7.75
>>> print("31 // 4 =", 31 // 4)
31 // 4 = 7
>>> ^Z
```

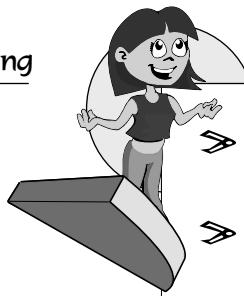
Start des Python-Interpreters von der Windows-Eingabeaufforderung aus.

Du beendest Python, indem du **Strg**+**Z** und dann **Enter** eingibst.

Ausführen von Python-Programmen mit python.exe

Häufig wirst du Python-Programme ausführen wollen, ohne die IDLE zu starten. Wie das geht, zeige ich dir am Beispiel von `hi.py` aus Kapitel 1, unter der Annahme, dass du Kapitel 1 durchgearbeitet und dieses Programm schon geschrieben hast:

- Starte die Windows-Eingabeaufforderung etwa über das Menü START|ALLE PROGRAMME|ZUBEHÖR|EINGABEAUFDORDERUNG.



- ⇒ Wechsle mit dem Kommando cd C:\py4kids\kap01 in das Verzeichnis mit hi.py.
- ⇒ Gib am Prompt der Eingabeaufforderung python hi.py ein. Damit führt der Python-Interpreter, das ist das Programm python.exe, das Python-Programm hi.py aus.

Der Pythoninterpret
er hat hi.py ausgeführt.

```

C:\> Eingabeaufforderung
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Dokumente und Einstellungen\Python4Kids>cd c:\py4kids\kap01

C:\py4kids\kap01>python hi.py
Hi Kleiner!
Wie viel ist eins und eins?
Ganz leicht!
1 + 1 = 2

C:\py4kids\kap01>

```

Ausführen von Python-Programmen durch Doppelklick

- ⇒ Versuche hi.py – wie jedes andere Python-Programm – auch auszuführen, indem du es im Windows-Explorer doppelt anklickst.

Ein »DOS-Fenster« geht kurz auf, schließt sich aber sofort wieder, bevor du noch die Ausgabe lesen kannst. Es ist genauso lange offen, wie die Ausführung des Programms braucht. Dagegen gibt es ein Mittel:

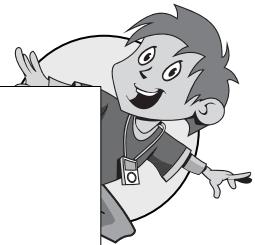


Wenn du Python-Programme so ausführen willst, musst du ihnen als letzte Anweisung eine Eingabeaufforderung anfügen. Es genügt zum Beispiel:

```
input("Schließen mit <Return>")
```

Das Programm wartet damit auf ein und wird erst nach dieser Eingabe beendet. Erst dann schließt sich das Eingabeaufforderungs-Fenster.

- ⇒ Probiere das aus, indem du hi.py um diese input()-Anweisung erweiterst und dann noch einmal durch Doppelklick ausführst.



Ausführen von Python-Programmen mit grafischer Benutzeroberfläche (GUI)

Dies geht ebenso mit Doppelklick im Windows-Explorer. GUI-Programme brauchen aber unbedingt als letzte Anweisung einen Aufruf von `mainloop()` oder `screen.mainloop()` (aus dem Modul `turtle` oder `Tkinter`). Beachte dazu *Clara Pythias Python-Special zu »mainloop«* in Kapitel 15, Seite 424.

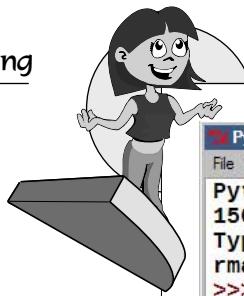
Anhang C

Was in Python 2.x anders ist

Guido van Rossum, der Erfinder von Python, und die Python Community haben nach gründlicher Diskussion mit der Veröffentlichung von Python 3 erstmals einen Versions-Schritt vollzogen, der nicht rückwärtskompatibel ist. Ein wesentliches Ziel war dabei, die Sprache einheitlicher und kompakter zu machen.

Die Änderungen, die die Syntax der Sprache betreffen, halten sich durchaus in Grenzen. Wenn du weiterhin mit Python arbeitest, wirst du aber sicherlich auf Programme stoßen, die noch in Python 2.x geschrieben sind. Dann solltest du schon ein wenig darüber Bescheid wissen, was sich geändert hat.

Damit du dir ein erstes Bild machen kannst und weißt, worauf du achten musst, gebe ich dir hier eine Zusammenstellung der wichtigsten Änderungen in Form zweier paralleler interaktiver Python-Sitzungen mit demselben Inhalt: die eine mit Python 2.6, die andere mit Python 3.1



```

Python Shell
File Edit Shell Debug Options Windows Help
Python 2.6.2 (r262:71605, Apr 14 2009, 22:40:02) [MSC v.
1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more info
rmation.

>>>
>>> # print ist eine Anweisung (und reserviertes Wort)
>>>
>>> print "Hello world!"
Hello world!
>>>
>>> # range() erzeugt eine Liste
>>>
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> type(range(10))
<type 'list'>
>>>
>>> # Dasselbe gilt für einige andere Typen:
>>>
>>> d = {"a":1, "b":2}
>>> d.keys()
['a', 'b']
>>> type(d.keys())
<type 'list'>
>>>
>>> # Division ganzer Zahlen mit / liefert ganze Zahlen
>>>
>>> 28 / 4
7
>>> 29 / 4
7
>>> # Es gibt zwei Ganzzahl-Datentypen: int und long
>>>
>>> (2**10, 2**100)
(1024, 1267650600228229401496703205376L)
>>> print type(2**10), type(2**100)
<type 'int'> <type 'long'>
>>>
>>> # unicode ist ein eigener Datentyp:
>>>
>>> print u'\u20ac', unichr(9786)
€ ©
>>> type(u'\u20ac')
<type 'unicode'>
>>>
>>> # Es gibt Klassen 'alten Typs' und 'neuen Typs'
>>>
>>> class A:                                # alter Typ
      pass

>>> class N(object):                         # neuer Typ
      pass

>>> a = A(); n = N()
>>> print type(A), type(a)
<type 'classobj'> <type 'instance'>
>>> print type(N), type(n), type(3)
<type 'type'> <class '__main__.N'> <type 'int'>
>>>

```

Was in Python 2.x anders ist

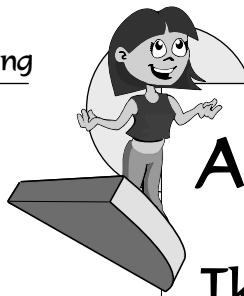


```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.
1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more info
rmation.

>>>
>>> # print() ist eine Funktion
>>>
>>> print("Hello world!")
Hello world!
>>>
>>> # range() erzeugt einen dynamischen Wertevorrat
>>>
>>> range(10)
range(0, 10)
>>> type(range(10))
<class 'range'>
>>>
>>> # Für einige weitere Typen gilt ähnliches:
>>>
>>> d = {"a":1, "b":2}
>>> d.keys()
dict_keys(['a', 'b'])
>>> d.items()
dict_items([('a', 1), ('b', 2)])
>>> print(type(d.keys()), type(d.items()))
<class 'dict_keys'> <class 'dict_items'>
>>>
>>> # Der Divisionsoperator / liefert immer Kommazahlen
>>>
>>> 28 / 4
7.0
>>> 29 / 4
7.25
>>>
>>> # Ganze Zahlen können beliebig lang sein
>>>
>>> (2**10, 2**100)
(1024, 1267650600228229401496703205376)
>>> print(type(2**10), type(2**100))
<class 'int'> <class 'int'>
>>>
>>> # Strings sind standardmäßig unicode
>>>
>>> print('\u20ac', chr(9786))
€ €
>>> type('\u20ac')
<class 'str'>
>>>
>>> # Die Klassenhierarchie wurde vereinheitlicht:
>>> # Es gibt nun nur mehr Klassen 'neuen Typs'.
>>> # Sie werden standardmäßig von object abgeleitet.
>>>
>>> class N:
    pass

>>> n = N()
>>> print(type(N), type(n), type(3))
<class 'type'> <class '__main__.N'> <class 'int'>
>>> |
```

Ln: 57 Col: 4



Anhang D

Tkinter-Farben

Bei deiner Arbeit mit der Turtle-Grafik wirst du Lust bekommen, verschiedene Farben zu verwenden. Immer nur rot, grün, blau, gelb ist langweilig.

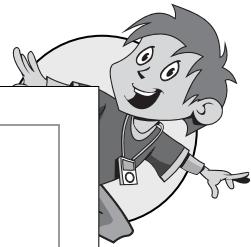
Zwei Standardfarben sind `black` und `white`. Dazwischen liegen *Grautöne*. `gray` ist ein mittleres Grau. Es gibt aber 101 feine Abstufungen von schwarzem Grau, `gray0`, bis zu weißem Grau, `gray100` – zum Beispiel `gray62`.

Jede der in der Liste weiter unten angeführten *Mischfarben* gibt es in vier Abstufungen. Ich erkläre dir das anhand der Farbe `blue`:

<code>color("blue")</code>	ergibt ein strahlendes Blau
<code>color("blue1")</code>	ergibt genau dasselbe wie "blue"
<code>color("blue2")</code>	ist etwas dunkler
<code>color("blue3")</code>	ist noch etwas dunkler
<code>color("blue4")</code>	ist ziemlich dunkel

Hier die vollständige Liste der vorgefertigten Farbtöne:

Snow	LightCyan	tan
seashell	PaleTurquoise	chocolate
AntiqueWhite	CadetBlue	firebrick
bisque	turquoise	brown
PeachPuff	cyan	salmon
NavajoWhite	DarkSlateGray	LightSalmon
LemonChiffon	aquamarine	orange
cornsilk	DarkSeaGreen	DarkOrange
ivory	SeaGreen	coral
honeydew	PaleGreen	tomato
LavenderBlush	SpringGreen	OrangeRed
MistyRose	green	red
azure	chartreuse	DeepPink
SlateBlue	OliveDrab	HotPink
RoyalBlue	DarkOliveGreen	pink
blue	khaki	LightPink
DodgerBlue	LightGoldenrod	PaleVioletRed
SteelBlue	LightYellow	maroon
DeepSkyBlue	yellow	VioletRed
SkyBlue	gold	magenta
LightSkyBlue	goldenrod	orchid



SlateGray	DarkGoldenrod	plum
LightSteelBlue	RosyBrown	MediumOrchid
LightBlue	IndianRed	DarkOrchid
LightCyan	sienna	purple
PaleTurquoise	burlywood	MediumPurple
LightBlue	wheat	Thistle

Anhang E

Weitere Informationen zu Python

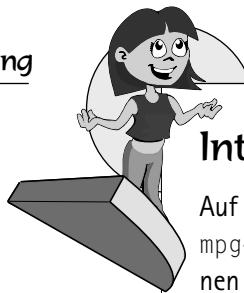
Die Originalquelle für Informationen über Python wie auch für das Herunterladen der jeweils aktuellsten Python-Version ist die Python-Website:
<http://python.org>.

Informationen zum Inhalt dieses Buches, zur Buch-CD, zum turtle-Modul und weitere Links sind zu finden auf <http://python4kids.net>.

Deutschsprachige Internetseiten zu Python

Die folgenden Informationsquellen beziehen sich (noch) hauptsächlich auf Python 2.x. Das ist sicherlich kein ernsthaftes Problem, erfordert aber ein bisschen Aufmerksamkeit für die Unterschiede zu Python 3.

- ❖ Die Zentrale für Unterrichtsmedien im Internet hat in ihrem Wiki eine gute Python-Seite mit vielen *Links* zu verschiedenen Quellen:
<http://wiki.zum.de/Python>
- ❖ Unter <http://www.rg16.at/~python/how2think/> findest du eine Einführung ins Programmieren: *Wie ein Informatiker denken lernen ... mit Python*, die deutschsprachige Fassung von *How To Think Like a Computer Scientist*. Sie ist ganz anders gemacht als *Python für Kids*, aber auch ein sehr sanfter Einstieg.
- ❖ *Programmieren lernen* von Alan Gauld ist die deutsche Übersetzung eines bekannten englischen Python-Tutorials. Zum Weiterlesen nach *Python für Kids* sicher gut geeignet:
<http://www.freenetpages.co.uk/hp/alan.gauld/german/>



Introducing Python

Auf der Buch-CD findest du ein englischsprachiges Video über Python als mpg-Datei. Es dauert etwa 20 Minuten. Auf dem Video sprechen Schülerinnen und Schüler einer amerikanischen Schule über ihre Erfahrungen mit Python. Außerdem treten Guido und einige weitere »Größen« der Python-Szene auf und erklären, warum sie in der Open-Source-Bewegung für Python arbeiten und was sie für seine Stärken halten. Interessant *und* unterhaltsam.

Anhang F

IDLE auf zwei Arten verwenden

Warum gibt es denn zwei Arten, die IDLE zu verwenden? Das will ich dir zum Abschluss noch an einem Beispiel vorführen.

Der große Unterschied

- ⇒ Starte die IPI-Shell und öffne ein Editor-Fenster. Schreibe in das Fenster folgenden Programmtext:

```
# IPI-Test / no subprocess
a = 5; b = 7
print(a, "+", b, "=", a + b)
```

- ⇒ Speichere das Programm unter dem Namen **ipitest.py** ab und führe es aus. Du erhältst folgende Ausgabe:

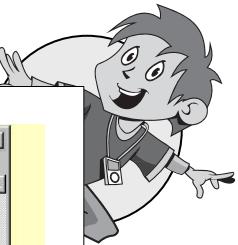
5 + 7 = 12

- ⇒ Ändere die erste Zeile ab und füge eine weitere Zeile an, so dass das Programm so aussieht:

```
c = 10; b = 7
print a, "+", b, "=", a + b
print b, "+", c, "=", b + c
```

- ⇒ Speichere es erneut und führe es aus. Dann hast du folgende Situation:

Der große Unterschied



```
ipitest.py - C:/py4kids/anhangF/ipitest.py
File Edit Format Run Options Windows Help
# IPI-Test / no subprocess
c = 10; b = 7
print(a, "+", b, "=", a + b)
print(b, "+", c, "=", b + c)

IPI Shell / Turtle Grafik
File Edit Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32
bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>>
5 + 7 = 12
>>>
5 + 7 = 12
7 + 10 = 17
>>> |
```

Ln: 9 Col: 4

➤ Führe nun genau denselben Vorgang mit der IDLE (Python-GUI) aus.
Das führt zu folgender Situation:

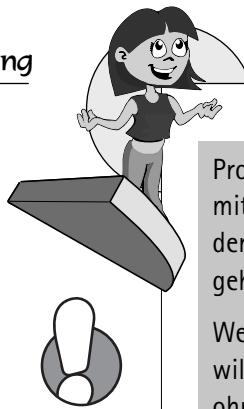
```
idletest.py - C:/py4kids/anhangF/idletest.py
File Edit Format Run Options Windows Help
# IDLE (Python GUI)
c = 10; b = 7
print(a, "+", b, "=", a + b)
print(b, "+", c, "=", b + c)

Python Shell
File Edit Shell Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32
bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
5 + 7 = 12
>>> ===== RESTART =====
>>>
Traceback (most recent call last):
  File "C:/py4kids/anhangF/idletest.py", line 3, in <module>
    print(a, "+", b, "=", a + b)
NameError: name 'a' is not defined
>>> |
```

Ln: 12 Col: 4

Die IDLE hat hier einen Programmfehler entdeckt, nämlich dass der Name `a` nicht existiert. (Du hast ihn bei der Programmänderung entfernt.) Das ist der IPI-SHELL verborgen geblieben, weil sie nicht bei jeder Programm-ausführung mit RESTART alle vorhandenen Namen löscht. Bei der zweiten Programmausführung war daher der Name `a` mit dem Wert 5 im Arbeitsspeicher noch vorhanden. Bei der Entwicklung großer Programme, wo Fehler nicht so leicht zu sehen sind wie in diesem Beispiel, kann das ganz schön problematisch werden. Erst wenn du die IPI-SHELL schließt, erneut startest und wieder `ipitest.py` ausführst, wird sie den Fehler melden.

Was ist die Lehre daraus?



Programmentwicklung ist mit der IDLE (Python-GUI) sicherer. Du kannst mit ihr auch Grafik-Programme entwickeln, doch muss dann am Ende der Programme immer ein `mainloop()`-Aufruf stehen. Interaktive Grafik geht damit nicht.

Wenn du bei der Programmentwicklung Grafik interaktiv verwenden willst, wie wir es in diesem Buch stets gemacht haben, musst du die IDLE ohne Subprozess, also die IPI-SHELL, verwenden. Das bringt den Nachteil mit sich, dass man Programmfehler leichter übersehen kann. Ausweg: den IPI von Zeit zu Zeit neu starten!

Die Konfiguration der IDLE ohne Subprozess, – also die Konfiguration des IPI – ist in Anhang A beschrieben (Seite 430f.).

Anhang G

Das Modul turtle.py: Die Referenz

Das Turtle-Grafik-Modul `turtle.py` stellt Funktionen und Klassen bereit, die gestatten, es sowohl in prozeduraler wie auch in objektorientierter Weise zu verwenden.

In dieser Referenz werden alle *Funktionen* angegeben, mit denen du die Turtle oder das Grafik-Fenster steuern kannst.

All diese Funktionen können auch als *Methoden* einer Turtle oder als Methoden des Grafik-Fensters aufgerufen werden. Du kannst in deinen Programmen beliebig viele Turtle-Objekte verwenden, aber immer nur ein Grafik-Fenster. Dieses einzigartige Objekt hat in diesem Buch stets den Namen `screen`. (In der Computer-Fachsprache sagt man: `screen` ist ein Singleton.)

⇒ Starte IPI-TURTLEGRAFIK und mach mit!

```
>>> from turtle import Screen, Turtle  
>>> screen = Screen()  
>>> screen.bgcolor("pink")  
>>> tina = Turtle()  
>>> tina.pensize(8)  
>>> tina.color("red", "orange")
```

I. Funktionen für die Kontrolle der Turtle

Wenn du mit nur einer (namenlosen) Turtle und mit Turtle-Grafik-Funktionen arbeiten willst, musst du alle Funktionen importieren:

```
from turtle import *
```

Wenn du mit Turtle-Objekten, dem screen-Objekt und mit ihren Methoden arbeiten willst, genügt es, mit folgender import-Anweisung die beiden Klassen zu importieren:

```
from turtle import Screen, Turtle
```

Im Folgenden werden zunächst alle Funktionen beschrieben, mit denen du die Turtle steuern kannst. Sie leiten sich von entsprechenden, meist gleichnamigen Turtle-Methoden her und sind nach ihren Aufgaben in Gruppen zusammengefasst. Danach werden die Funktionen für die Steuerung des Grafik-Fensters beschrieben. Diese leiten sich von Methoden des screen-Objekts her.

Für alle Funktionen (und Methoden), die du importiert hast, kannst du eine ausführlichere Erklärung in der PYTHON-SHELL mit der Funktion help() erhalten. (Siehe Kapitel 2, Seite 54.) Zum Beispiel:

```
>>> help(goto)  
>>> help(tina.pencolor)  
>>> help(screen.onkeypress)
```

I. Funktionen für die Kontrolle der Turtle

1. Turtle-Bewegung

1.a Bewegen und zeichnen

`forward(distance) | fd`

distance: eine Zahl

Bewegt die Turtle um die Strecke *distance* (in Pixel) vorwärts, und zwar in die Richtung, in die sie zeigt.

`backward(distance) | bk`

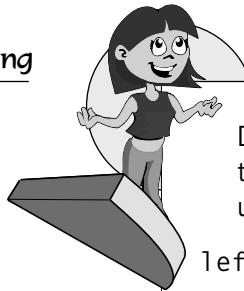
distance: eine Zahl

Bewegt die Turtle um die Strecke *distance* (in Pixel) rückwärts.

`right(angle) | rt`

angle: eine Zahl (ein Winkel)





Dreht die Turtle um *angle* Winkeleinheiten nach rechts. (Voreingestellte Winkeleinheit ist Grad; das kann über die Funktionen `degrees()` und `radians()` geändert werden.)

`left(angle) | lt`

angle: eine Zahl (ein Winkel)

Dreht die Turtle um *angle* Winkeleinheiten nach links. (Siehe auch: `right()`)

`goto(x, y=None) | setpos | setposition`

Argumente: *x, y*: zwei Zahlen oder

x: ein Paar/Vektor von zwei Zahlen (*y* = *None*)

Bewegt die Turtle zum Punkt mit den absoluten Koordinaten (*x,y*).

Wenn der Zeichenstift unten ist, wird eine Strecke gezeichnet. Die Orientierung der Turtle wird nicht geändert.

`setx(x)`

x: eine Zahl

Setzt die erste Koordinate der Turtle auf *x*. Die zweite Koordinate bleibt unverändert.

`sety(y)`

y: eine Zahl

Setzt die zweite Koordinate der Turtle auf *y*. Die erste Koordinate bleibt unverändert.

`setheading(to_angle) | seth`

to_angle: eine Zahl (ein Winkel)

Dreht die Turtle so, dass sie in Richtung des angegebenen Winkels *to_angle* orientiert ist.

Hier sind einige gebräuchliche Richtungen in Grad:

'standard'-mode:	'logo'-mode:
0 - ost	0 - nord
90 - nord	90 - ost
180 - west	180 - süd
270 - süd	270 - west

`circle(radius, extent=None, steps=None)`

radius: eine Zahl

extent (optional): eine Zahl (ein Winkel)

steps (optional): eine ganze Zahl

Zeichnet einen Kreis(bogen) mit gegebenem *radius*. Der Kreismittelpunkt ist *radius* Einheiten links von der Turtle, wenn *radius* > 0 ist, andernfalls rechts von der Turtle. Der Winkel *extent* gibt an, welcher Teil des Kreises gezeichnet wird. Wenn *extent* fehlt, wird ein voller

I. Funktionen für die Kontrolle der Turtle



Kreis gezeichnet. Kreise zeichnet die Turtle als Vielecke mit vielen Ecken, so dass sie rund ausschauen. Die Zahl *steps* legt fest, wie viele Ecken berechnet werden. Wenn sie nicht angegeben wird, wird sie automatisch optimal berechnet.

`dot(size=None, *color)`

size: eine Zahl größer oder gleich 1

color: ein Farbstring oder ein numerisches RGB-Farbtupel

Zeichnet einen Punkt mit dem Durchmesser *size*, in der Farbe *color*. Wenn *color* nicht angegeben wird, wird die Turtle-Farbe verwendet. Wenn auch *size* nicht angegeben wird, wird `pensize() + 4` verwendet.

`home()`

Bewegt die Turtle zur Ausgangsposition und richtet sie in die Ausgangsorientierung (abhängig vom mode) aus.

`stamp()`

Erzeugt einen »Stempelabdruck« der Turtle-Form auf der Zeichenfläche an der aktuellen Turtle-Position und gibt eine *id* für diesen Abdruck zurück, so dass mit `clearstamp(id)` dieser Abdruck wieder gelöscht werden kann.

`clearstamp(stamp_id)`

stamp_id: Zahl, Rückgabewert eines `stamp()`-Aufrufs

Löscht den Stempelabdruck mit der angegebenen *stamp_id*.

`clearstamps(n=None)`

n: positive oder negative ganze Zahl

Wenn *n* nicht angegeben wird, werden alle Abdrücke der Turtle gelöscht. Wenn *n > 0* ist, werden die ersten *n* Abdrücke gelöscht, wenn *n < 0* ist, die letzten *n*.

`undo()`

Macht die letzte Turtle-Aktion rückgängig. Kann wiederholt aufgerufen werden. Die Anzahl der Aktionen, die rückgängig gemacht werden können, hängt von der Größe des `undo`-Puffers ab. (Kann in der Konfigurationsdatei `turtle.cfg` eingestellt werden.)

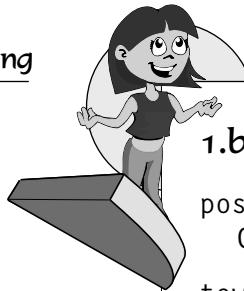
`speed(speed=None)`

speed: ganze Zahl im Bereich 0 ... 10, mit folgender Bedeutung:

0: keine Animation, Turtle »springt« zum Zielpunkt.

1 ... 10: langsam ... schnell

Setzt die Geschwindigkeit der Turtle auf *speed* oder gibt deren Wert zurück.



1.b Abfrage des Zustands der Turtle

`position()` | `pos`

Gibt die aktuelle Position der Turtle zurück: (x, y) (als Vektor)

`towards($x, y=None$)`

Argumente:

x, y : zwei Zahlen (Koordinaten eines Punktes) oder

x : ein Paar oder Vektor von zwei Zahlen (und $y=None$) oder

x : ein Turtle-Objekt (eine Turtle) (und $y=None$)

Gibt den Winkel zwischen der Linie von der Turtle-Position zum angegebenen Punkt und der Startorientierung der Turtle zurück (je nach Modus – "standard" oder "logo").

`xcor()`

Gibt die x-Koordinate der Turtle zurück.

`ycor()`

Gibt die y-Koordinate der Turtle zurück.

`heading()`

Gibt die aktuelle Orientierung der Turtle zurück.

`distance($x, y=None$)`

Argumente:

x, y : zwei Zahlen oder

x : ein Paar oder Vektor von zwei Zahlen (und $y=None$) oder

x : ein Turtle-Objekt (eine Turtle) (und $y=None$)

Gibt die Entfernung des Punktes (x, y) von der Turtle zurück.

1.c Einstellungen und Maßeinheiten

`degrees(fullcircle=360.0)`

Setzt die Einheit für die Winkelmessung auf Grad. Optionales Argument: `fullcircle` ist Anzahl der 'Grade' eines vollen Winkels, Standardwert ist 360.0).

`radians()`

Setzt die Einheit für die Winkelmessung auf Radian (Bogenmaß).

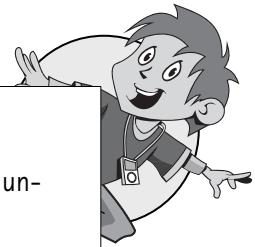
2. Steuerung des Zeichenstiftes

2.a Eigenschaften für das Zeichnen

`pendown()` | `pd` | `down`

Setzt den Zeichenstift nach unten. Zeichnet bei nachfolgenden Bewegungen.

I. Funktionen für die Kontrolle der Turtle



`penup() | pu | up`

Hebt den Zeichenstift an. Zeichnet nicht bei nachfolgenden Bewegungen.

`pensize(width=None) | width`

width: positive Zahl

Setzt die Strichdicke auf *width*, wenn *width* angegeben ist; andernfalls wird die Strichdicke zurückgegeben. Wenn *resizemode* auf 'auto' gestellt ist und die Turtleshape ein Polygon ist, dann wird das Polygon auch mit der Strichdicke *width* dargestellt.

`pen(pen=None, **pendict)`

pen: ein Dictionary mit folgenden Schlüssel-Wert-Paaren:

"shown"	:	True/False
"pendown"	:	True/False
"pencolor"	:	Farbstring oder Farb-Zahlentripel
"fillcolor"	:	Farbstring oder Farb-Zahlentripel
"pensize"	:	positive Zahl
"speed"	:	ganze Zahl im Bereich 0..10
"resizemode"	:	"auto" or "user" or "noresize"
"outline"	:	positive Zahl
"stretchfactor"	:	(positive Zahl, positive Zahl)
"tilt"	:	Zahl (integer oder float)
"shearfactor"	:	positive Zahl

Setzt die Eigenschaften des Zeichenstiftes entsprechend des als Argument übergebenen Dictionarys *pen* und/oder der an `**pendict` übergebenen Schlüsselwort-Argumente. Wenn keine Argumente angegeben werden, wird das *pen*-Dictionary zurückgegeben.

`isdown()`

Gibt True zurück, falls der Zeichenstift unten ist, False sonst.

2.b Farben einstellen und abfragen

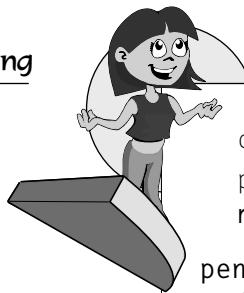
Im Folgenden steht *s* stets für einen Farbstring (siehe Anhang D) und *r*, *g*, *b* für drei Zahlen zur numerischen Beschreibung des Rot-, Grün- und Blauanteils von RGB-Farben. *r*, *g* und *b* müssen im Bereich 0 .. colormode liegen. Die Farbe Lachsrosa etwa wird nach `colormode(1)` durch das Tripel (1.0, 0.5, 0.25) angegeben, nach `colormode(255)` durch das Tripel (128, 64, 32).

`color(*args)`

Es gibt Aufruf-Formate mit 0, 1, 2, 3 oder 6 Argumenten.

`color(s), color((r, g, b))`, `color(r, g, b)`: Setzt die Zeichenfarbe und die Füllfarbe auf den gegebenen Wert.

`color(s1, s2), color((r1, g1, b1), (r2, g2, b2))` oder



`color(r1, g1, b1, r2, g2, b2)`: Gleichwertig mit `pencolor(s1)` und `fillcolor(s2)` – und entsprechend für die anderen Formate.

`pencolor(*args)`

Aufruf-Formate: `pencolor()` oder `pencolor(s)`, `pencolor((r,g,b))`, `pencolor(r,g,b)`

Setzt die Zeichenfarbe oder gibt ihren Wert zurück, wenn kein Argument übergeben wird.

`fillcolor(*args)`

Aufruf-Formate: `fillcolor()` oder `fillcolor(s)`, `fillcolor((r,g,b))`, `fillcolor(r,g,b)`

Setzt die Füllfarbe oder gibt ihren Wert zurück, wenn kein Argument übergeben wird.

2.c Füllen

`filling()`

Gibt den Füll-Zustand ab. Gibt `True` zurück, wenn Füllen mit `begin_fill()` eingeschaltet wurde, `False` sonst.

Infolge eines Bugs steht `filling()` in Python 3.1.1 nur als Methode der Klasse `Turtle` zur Verfügung, nicht aber als Funktion.

`begin_fill()`

Beginnt das Zeichnen einer Figur, die gefüllt werden soll.

`end_fill()`

Füllt die Figur, die nach dem Aufruf von `begin_fill()` gezeichnet wurde, mit der Füllfarbe.

2.d Weitere Funktionen für die Grafik

`reset()`

Löscht die Zeichnungen der Turtle vom Grafik-Fenster. Setzt die Turtle in den Mittelpunkt des Fensters und alle Attribute auf ihre Anfangswerte (mit Ausnahme der Turtle-Gestalt).

`clear()`

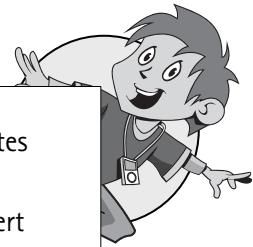
Löscht die Zeichnungen der Turtle vom Grafik-Fenster. Die Turtle wird nicht bewegt, Zeichnungen anderer Turtles werden nicht beeinflusst.

`write(arg,move=False,align='left', font=('Arial',8,'normal'))`

`arg`: String, der geschrieben werden soll

`move`: `True` oder `False`

I. Funktionen für die Kontrolle der Turtle



`align: 'left', 'center' oder 'right'` zur Ausrichtung des Textes
`font:` ein Dreiertupel zur Beschreibung der Schriftart

Schreibt Text an die aktuelle Turtle-Position, entsprechend dem Wert von `align` in der für `font` angegebenen Schriftart. Wenn `move True` ist, wird die Turtle zum rechten unteren Ende des Textes bewegt.

3. Zustand der Turtle

`showturtle() | st`
Macht die Turtle sichtbar.

`hideturtle() | ht`
Macht die Turtle unsichtbar.

`isvisible() | shownp`
Gibt True zurück, wenn die Turtle sichtbar ist, False sonst.

`shape(name=None)`
`name:` ein String, der eine registrierte Turtleshape bezeichnet.
Setzt die Gestalt der Turtle auf die Gestalt mit dem angegebenen `name`.

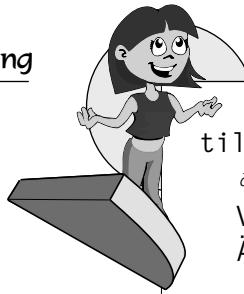
`resizemode(rmode=None)`
`rmode: "auto", "user", "noresize"`
Setzt resizemode auf einen der drei Werte mit folgender Wirkung:
"auto": Turtle-Größe passt sich automatisch der pensize an.
"user": Turtle-Größe wird vom Benutzer mit shapesize() festgelegt.
"noresize": keine Größenänderung der Turtle

Wenn kein Argument angegeben ist, wird der aktuelle resizemode zurückgegeben.

Anmerkung: Wird shapesize() mit Argumenten aufgerufen, so setzt dies automatisch den resizemode auf den Wert "user".

`shapesize(stretch_wid=None, stretch_len=None,
outline=None) | turtlesize`
`stretch_wid, stretch_len, outline:` drei positive Zahlen
Setzt Streckfaktoren für die Turtle-Shape und die Randstärke oder gibt diese zurück. `stretch_len` betrifft Streckung in die Länge, `stretch_wid` die in die Breite.

`tiltangle(angle=None)`
`angle:` eine Zahl (ein Winkel)
Verdreht die Turtleshape auf `angle` bezogen auf ihre Orientierung oder gibt den Verdrehungswinkel `angle` zurück. Ändert nicht die Orientierung der Turtle.



tilt(*angle*)

angle: eine Zahl (ein Winkel)

Verdreht die Turtle um *angle* bezogen auf ihren aktuellen Tilt-Winkel.
Ändert *nicht* die Orientierung der Turtle.

shearfactor(*shear=None*)

shear: eine Zahl (ein Winkel)

Setzt den Scherungsfaktor für die Turtle-Shape oder gibt ihn zurück,
wenn *shear* nicht angegeben ist. Die Turtle-Gestalt wird einer Sche-
rung unterworfen, wobei *shear* der Tangens des Scherungswinkels ist.
Ändert *nicht* die Orientierung der Turtle.

shapetransform(*t11=None*, *t12=None*, *t21=None*,

***t22=None*)**

t11, *t12*, *t21*, *t22* sind vier Zahlen, die eine Transformationsma-
trix festlegen. Wenn alle vier nicht angegeben sind, wird die Transforma-
tionsmatrix als Vierertupel zurückgegeben. Die Turtle-Shape wird ent-
sprechend dieser Matrix transformiert und die Streckfaktoren, der Ver-
drehungswinkel und der Scherungsfaktor werden entsprechend einge-
stellt. Ändert *nicht* die Orientierung der Turtle.

get_shapepoly()

Gibt die aktuelle Turtle-Form als Tupel von Koordinatenpaaren zurück.

4. Turtle-Ereignisse

onclick(*fun*, *btn=1*)

fun: Funktion mit 2 Argumenten

btn: 1 oder 2 oder 3 für die drei Maustasten

Bindet den Aufruf von *fun* an das Mausklick-Ereignis auf die Turtle. Die
Koordinaten des angeklickten Punktes werden beim Aufruf von *fun* als
Argumente übergeben.

onrelease(*fun*, *btn*)

fun, btn: wie bei onclick()

Bindet den Aufruf von *fun* an das Ereignis »Maustaste loslassen auf der
Turtle«. Die Koordinaten des angeklickten Punktes werden beim Aufruf
an *fun* als Argumente übergeben.

ondrag(*fun*, *btn*)

fun, btn: wie bei onclick()

In der Folge wird *fun* während des »Ziehens« der Turtle (mit gedrückter
Maustaste *btn*) wiederholt aufgerufen. Die Koordinaten der aktuellen
Mausposition werden jeweils an *fun* als Argumente übergeben.



5. Spezielle Turtle-Funktionen

`begin_poly()`

Startet die Aufzeichnung der Eckpunkte eines Polygons mit der aktuellen Turtle-Position.

`end_poly()`

Beendet die Aufzeichnung der Eckpunkte eines Polygons mit der aktuellen Turtle-Position. Dieser letzte Punkt wird mit dem ersten verbunden, wenn das Polygon gezeichnet wird.

`get_poly()`

Gibt das zuletzt aufgezeichnete Polygon zurück.

`clone()`

Erzeugt einen Klon (eine exakte Kopie) der Turtle mit gleicher Position, Orientierung und Eigenschaften und gibt diesen zurück.

`getturtle() | getpen`

Gibt das Turtle-Objekt selbst zurück. Einzige sinnvolle Verwendung: als Funktion, die die »namenlose Turtle« zurückgibt.

`getscreen()`

Gibt das TurtleScreen-Objekt zurück, auf dem die Turtles zeichnen.

`setundobuffer(size)`

`size`: eine positive Zahl oder `None`

Stellt die Größe des undo-Puffers ein. Falls `size` gleich `None` ist, wird der undo-Buffer ausgeschaltet.

`undobufferentries()`

Gibt die Anzahl der Einträge im undo-Puffer zurück.

II. Funktionen für die Kontrolle des Turtle-Grafik-Fensters

6. Fenster-Aktionen

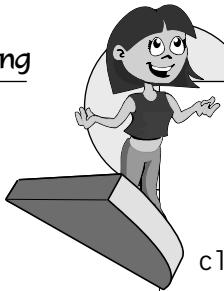
`bgcolor(*args)`

`args`: ein Farbstring oder ein Farbtupel

Setzt die Hintergrundfarbe des Grafik-Fensters oder gibt sie zurück.

`bgpic(picname=None)`

`picname`: String. Name einer *.gif-Datei oder "nopic"



II. Funktionen für die Kontrolle des Turtle-Grafik-Fensters

`setbackground()`
Setzt ein Hintergrundbild für das Grafik-Fenster bzw. entfernt es. Ohne Argument wird der Name der aktuell verwendeten Grafik-Datei zurückgegeben.

`clearscreen()`

Entfernt alle Zeichnungen und alle Turtles vom Grafik-Fenster und setzt das leere Grafik-Fenster auf seinen Anfangszustand.

Als Methode auch mit dem Namen `clear` verfügbar.

`resetscreen()`

Setzt alle Turtles im Grafik-Fenster auf ihren Anfangszustand.

Als Methode auch mit dem Namen `reset` verfügbar.

`screensize(canvwidth=None, canvheight=None,
bg=None)`

canvwidth, *canvheight*: zwei positive Zahlen

bg: Farbstring oder Farbtupel

Stellt die Größe der mittels Scrollbars erreichbaren Zeichenfläche ein, auf der die Turtles zeichnen, oder gibt ein Tupel (*widht*, *height*) zurück. (Die Fenster-Größe wird nicht verändert.)

`setworldcoordinates(llx, lly, urx, ury)`

llx, *lly*, *urx*, *ury*: vier Zahlen

llx, *lly* sind die Benutzerkoordinaten der linken unteren Ecke, *urx*, *ury* die der oberen rechten Ecke.

Stellt ein benutzerdefiniertes Koordinatensystem ein. Falls nötig, stellt es auf den Modus "world" um. Wenn Modus "world" aktiv ist, werden alle vorhandenen Zeichnungen entsprechend dem neuen Koordinatensystem aktualisiert.

7. Kontrolle der Animation

`tracer(flag=None, delay=None)` ! Methode des Grafik-Fensters !

flag: True oder False

Schaltet Turtle-Animation ein (*flag=True*) bzw. aus (*flag=False*).

`update()`

Führt ein Update des Grafik-Fensters aus. Speziell nützlich, wenn `tracer(False)` gesetzt ist.

`delay(delay=None)`

delay: positive ganze Zahl

Setzt – oder gibt zurück – das Zeitintervall zwischen zwei aufeinanderfolgenden Neuzeichnungen des Canvas in Millisekunden. Je größer *delay*, desto langsamer läuft die Animation ab.



8. Ereignisse des Grafik-Fensters

Im Folgenden werden für Maus-Ereignisse die Maustasten mit Zahlen angesprochen: 1 ist die linke, 2 die mittlere und 3 die rechte Maustaste.

`listen(x=None, y=None)`

Setzt den Fokus auf das Grafik-Fenster (siehe: `onkey()`).

`onkeyrelease(fun, key=None) | onkey`

fun: eine Funktion ohne Argumente

key: String-Kennung einer Taste, wie "a", "z", "space", "Escape"

Bindet die Funktion *fun* an das Loslassen der Taste *key*. Das Grafik-Fenster muss den Fokus haben, damit *key*-Ereignisse registriert werden (siehe `listen()`).

`onkeypress(fun, key=None) | onkeyrelease`

fun: eine Funktion ohne Argumente

key: wie bei `onkeyrelease`

Bindet die Funktion *fun* an das Loslassen der Taste *key*. Das Grafik-Fenster muss den Fokus haben (siehe `listen()`).

`onscreenclick(fun, btn=1)`

fun: Funktion mit 2 Argumenten

btn: 1 oder 2 oder 3 für die Maustasten

Bindet den Aufruf von *fun* an das Mausklick-Ereignis ins Grafik-Fenster mit der entsprechenden Maustaste. Die Koordinaten des angeklickten Punktes werden beim Aufruf an *fun* als Argumente übergeben.

Anmerkung: Als Methode auch mit dem Namen `onclick()` verfügbar.

`ontimer(fun, t=0)`

fun: Funktion ohne Argumente

t: positive ganze Zahl

Setzt eine Uhr in Gang, die die Funktion *fun* nach *t* Millisekunden aufruft.

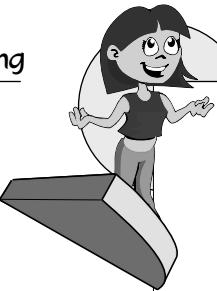
`mainloop()`

Setzt die Ereignis-Schleife in Gang. Muss die letzte Anweisung in einem ereignisorientierten Turtle-Grafik-Programm sein.

9. Einstellungen und spezielle Funktionen

`setup(width=0.5, height=0.75, startx=None, starty=None)`

width, *height*: ganze Zahlen (Breite bzw. Höhe in Pixel), oder Komazahlen ≤ 1.0 , Bruchteil der Bildschirmbreite / -höhe



II. Funktionen für die Kontrolle des Turtle-Grafik-Fensters

startx, starty: Zahlen. Wenn positiv, Startposition in Pixel vom linken/oberen Bildschirmrand. Wenn negativ vom rechten/unteren Bildschirmrand.

startx=None zentriert das Grafik-Fenster horizontal,

starty=None zentriert das Grafik-Fenster vertikal am Bildschirm.

`mode(mode=None)`

mode: die Zeichenkette 'standard', 'logo' oder 'world'

Setzt den Turtle-Modus auf 'standard' oder 'logo' und führt ein `reset()` aus. Modus 'standard' ist kompatibel mit `turtle.py`.

Modus 'logo' ist kompatibel mit den meisten Turtle-Grafik-Implementierungen in Logo. Modus 'world' verwendet benutzerdefinierte Koordinaten. (Siehe `setworldcoordinates()`)

Modus	Anfängliche Orientierung der Turtle	positive Winkel
'standard'	nach rechts (Osten)	im Gegenuhrzeigersinn
'logo'	nach oben (Norden)	im Uhrzeigersinn
'world'	unwesentlich (nach rechts)	im Gegenuhrzeigersinn



In diesem Buch wird ausschließlich der Modus 'logo' verwendet!

`colormode(cmode=None)`

cmode: 1.0 oder 255

Setzt colormode auf 1.0 oder 255. Ohne Argument gibt es den `colormode` zurück.

`getcanvas()`

Gibt das Canvas-Objekt zurück, auf dem die Turtles zeichnen.

`getshapes()`

Gibt eine Liste der Namen der aktuell registrierten Turtle-Shapes zurück.

`register_shape(name, shape=None) | addshape`

name: ein String

shape: ein Polygon oder ein Objekt der Klasse Shape

Füge eine Turtle-Form zum shape-Dictionary des Grafik-Fensters hinzu. Folgende Fälle sind möglich:

(1) *name* ist der Dateiname einer gif-Datei und *shape* ist None: Meldet das entsprechende Bild als Turtle-Gestalt an.

(2) *name* ist eine beliebige Zeichenkette und *shape* ist ein Tupel von Koordinaten-Paaren. Registriert die entsprechende Polygon-Form.

II. Funktionen für die Kontrolle des Turtle-Grafik-Fensters



(3) *name* ist eine beliebige Zeichenkette und *shape* ist ein zusammengefügtes *Shape*-Objekt. Registriert die entsprechend zusammengesetzte Turtle-Form.

turtles()

Gibt eine Liste aller Turtles in dem Grafik-Fenster zurück.

title(*titlestring*)

titlestring: ein String, der im Titelbalken des Grafik-Fensters erscheint

Setzt den *title* des Turtle-Grafik-Fensters auf *titlestring*.

window_height()

Gibt die Höhe des Turtle-Grafik-Fensters zurück.

window_width()

Gibt die Breite des Turtle-Grafik-Fensters zurück.

bye()

Schließt das Turtle-Grafik-Fenster.

exitonclick()

Bindet *bye()* an Mausklicks auf das Grafik-Fenster.

10. Eingabe-Funktionen

textinput(*title*, *prompt*)

title, *prompt*: zwei Strings

Öffnet grafischen Eingabedialog für einen String. *title* ist der Titel des Fensters, *prompt* beschreibt, was einzugeben ist.

numinput(*title*, *prompt*, *default=None*, *minval=None*,

***maxval=None*)**

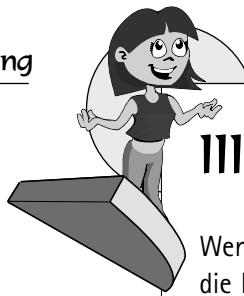
title, *prompt*: zwei Strings wie bei *textinput()*

default: Standard-Vorgabewert

minval, *maxval*: minimaler und maximaler erlaubter Eingabewert

Öffnet grafischen Eingabedialog für eine Zahleneingabe.

Methoden des Grafik-Fensters sind Methoden dieses Objekts.



III. Die Datei »turtle.cfg«

Wenn `turtle.py` importiert oder geladen wird, liest das Modul zunächst die Konfigurations-Datei `turtle.cfg`. Das ist eine Textdatei, in der einige Einstellungen festgelegt werden. Für das Buch »Python für Kids« hat sie folgenden Inhalt:

```
width = 400
height = 300
canvwidth = 360
canvheight = 270
shape = turtle
mode = logo
importvec = False
language = german
```

Du kannst diese Datei editieren und so das Modul `turtle` deinen Bedürfnissen anpassen.

- ❖ `width` und `height` legen die Größe des Grafik-Fensters fest.
- ❖ `canvwidth` und `canvheight` legen die Größe der Zeichenfläche fest.
Du kannst sie größer als das Fenster wählen und bekommst dann Scrollbars an den Fensterkanten, mit denen du die Zeichenfläche verschieben kannst.
- ❖ `shape` legt die Gestalt der Turtle fest. Änderst du den Eintrag beispielsweise auf `shape = arrow`, so startet das Turtle-Grafik-Fenster immer mit einer Turtle in Dreiecksgestalt.

Weitergehende Informationen über den Gebrauch von `turtle.cfg` und das Modul `turtle.py` im Allgemeinen findest du auf der Seite
<http://python4kids.at>.

Stichwortverzeichnis



Symbole

`__init__` 385
`__name__` 218
2-mal-2-Unterstrich-Verfahren 384

A

Anweisung
 `break` 319
 `class` 376, 383
 `def` 107, 108
 `for` 180
 `global` 115
 `if` 118
 `if ... elif ... else` 127
 `if ... else` 123
 `import` 148
 `pass` 269
 `return` 252, 271
 `try ... except` 328
 `while` 237, 241
 `yield` 305
Argument 31, 134
 als Schlüsselwort-Argument 173
`askcolor()` 352
Attribut 389
Aufruf einer Funktion 266
Ausdruck 26
 arithmetischer 28
 boolescher 119
 logischer 119
Ausführbares Objekt 266
auskommentieren 201

B

`backslash` 289
bedingte Schleife 241

Bibliothek, `mytools.py` 147
Boolescher Operator in 293
Bottom-up-Entwurf 163
Friedenslogo 208

C

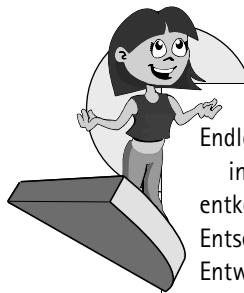
Cäsar-Code 320
case-sensitive 42
codieren 67
`colorchooser.askcolor()` 352

D

Dateien 326
Datei-Methode
 `read()` 327
 `write()` 330
Datum 367
Dictionary 316
Dictionary Comprehension 326
Divisionsoperator
 `%`, Modulo-Operator 258
 `/` in Python 2.x und Python 3 438
 `//` für Ganzzahldivision 226
Doc-String 108
Dummy-Parameter 342

E

Editor-Fenster 35
Effekt einer Funktion 253
eingebaute Funktion
 `len()` 193, 287
 `open()` 327
 `tuple()` 185
 `elif` 127
 `else` 123
 Endlosschleife 239



Endlosschleifen
in der IDLE 240
entkommentieren 201
Entschlüsseln 320
Entwicklungsumgebung,
integrierte 25
Ereignis
Tastatur 343
Timer 358
Ereignisgesteuert 334
event 334

F

Faktorielle 261
False 118
Farben
durch Tupel festlegen 224
Festlegung durch Hexadezimal-
zahlen 353
Fehlermeldung 29
NameError 29
Formatierungsmarke 151
for-Schleife 181, 183
allgemeine 195
als Zählschleife 189
Funktion 31
als Objekt 266
Argumente 134
Aufruf 107
definieren 103
Definition 106
Faktorielle 262
jump() 143
krange() 302
mit Parametern 134, 137, 139
mit Rückgabewert 249, 255
n_eck() 214
nachfolger() 259
quardat() 251, 254
randint() 230
randomwalk() 242
reihe() 259
rosette() 220
strichel() 187
superrosette() 221

zufallsweg() 234
Funktion, eingebaute
input() 92
print() 32
Funktionsaufruf 31
Funktionsdefinition
Kopf 106
Körper 106

G

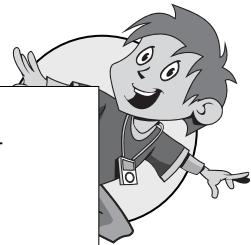
Generator 304
frange() 303
global 115

H

Hexadezimalzahlen 353

I

IDLE 24
Editor-Fenster 35
Shell-Fenster 25
if ... elif ... else - Anweisung 127
if ... else - Anweisung 123
if-Anweisung 118, 120
import 29, 148
this 323
Importieren
Modul 29
in 293, 317
index 321
Index 288
input() 92
Instanz 281
Instanzvariable 389
Initialisierung 390
Integrierte Entwicklungsumgebung
25
IPI - Turtle-Grafik 50



K

Klasse 376
Agent 393
Bote 388
FreundlicherBote 391
Instanz 281
Konstruktor 280
MyTurtle 377
Namenskonvention 281
SchlauerBote 388
SehrSchlauerBote 389
Klassenbibliothek 282
Klassendefinition 276, 376, 383
Klassenmethode 392
Kommazahl 94
Kommentar 38
Konstruktor 280
 __init__ 385
Kopfkommentar 38

L

Langzahlarithmetik 264
Laufzeitmessung 267
lazy evaluation 184
List Comprehension 309
Liste, Methoden 297
Listen-Methode
 append () 297
 pop() 300
Logischer Operator
 and 411
 in 293
 or 411
mytools.py 188
jump() 175

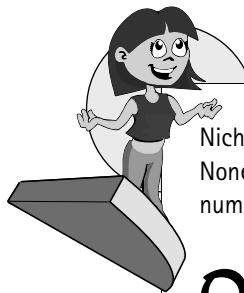
M

Maschinensprache 24
Mehrfachverzweigung 127
Methode 279
 für Listen, append 297
index 321

Methoden von Sequenzen 294
Methodenaufruf
 Syntax 279
Modul
 datetime 366
 importieren 29
 math 29
 polygon 217
 random 230, 270
 this 323
 time 267
 winsound 421
Moorhuhn-Spiel 398
Muster 43
 Allgemeine for-Schleife 195
 Bedingte Anweisung 120
 Bedingte Schleife 241
 Einfaches Python-Script 44
 Einlesen von Dateien 328
 Fehlerbehandlung mit
 try...except... 329
 for-Schleife als Zählschleife 189
Funktion
 mit Rückgabewert 255
Funktion mit Parametern 141
Funktionsaufruf mit Positions-
 und Schlüsselwort-
 Argumenten 175
Funktionsdefinition 111
Funktionsdefinition mit Stan-
 dardwerten 174
Mehrfach-Verzweigung 129, 130
Methodenaufruf 279
Programm-Verzweigung 123
Schreiben von Dateien 330
Turtle-Shapes definieren 361
Wertzuweisung 87

N

Name 77, 79, 83
erklärender 152
global 136
lokaler 135



Nichts 270
None 270
numinput() 96

O

Objekt 83
Namenskonvention 281
Operator
* für Sequenzen 290
+ für Sequenzen 290
in 293, 317

P

Parameter 134, 139
Polygone 214
print() 32
Programm 34
abspeichern 38
ereignisgesteuert 334
Friedensfahne 208
laufzeit.py 269
Mini Quiz 200
guardat.py 253
randomwalk.py 232
rechteck.py 256
selbständig ausführbar 423
Programmausführung 37
Programm-Entwicklung, schrittweise 169
Programm-Entwurf
bottom-up 163
Top-down 158
Programmiersprache 24
Programmverzweigung 122
Prompt 25
python4kids.net 426
Python-Anweisung global 115
Python-Funktion float() 94
Python-Interpreter 24
Python-Programm
dialog.py 95
dreieck() mit for-Schleife 190
dreieck.py 138

dreieck.py 104
miniquiz.py 117, 122, 201
seifenoper.py 149
yinyang.py 158, 163
Python-Special
Generatoren 304
lange Zeilen 171
Schlüsselwort-Argumente für
print() 181
Sequenzen, scheibchenweise 291
Standardwert für Parameter 172
Tupel entpacken 197

Q

Quadratwurzel 30

R

random walk 229
range 184
Reserviertes Wort 43
and 411
break 319
class 376
def 108
elif 127
else 123
except 328
False 118
for 181
from 43
global 115
if 118
import 29
in 181
is 298
None 271
not 294
or 411
pass 270
return 252
True 118
try 328
while 237



yield 304
Rosette 220
Rückgabewert 250

S

Schleife
 bedingte 229, 237
 while 237
Schleifenkopf 183
Schleifenkörper 183
Schleifenvariable 239
Schlüssel-Wert-Paare 316
Schlüsselwort-Argumente 173
 print() 181

screen-Methode
 clear() 338
 getshapes() 360
 listen() 344
 onclick() 334
 onkeypress() 343
 ontimer() 358
 register_shape() 360
Scribble 337
Script 35
self 381
Sequenzen 285

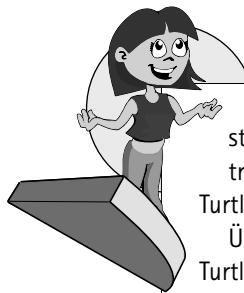
 * Operator 290
 + Operator 290
 entpacken, im Schleifenkopf 199
Funktion len() 287
in Operator 293
Index 288
Methoden 294
Scheiben (Slices) 291
veränderbare / nicht veränderba-
 re 300

Shell 26
sitecustomize.py 147
spezieller Name __name__ 218
strichel() 187
String 32
 Leerstring 40
 mehrzeilig 97
 Methoden 295, 300
 Verkettung 150

String-Methode
 format() 151
 lower() 295
 split() 296
 startswith() 296
 upper() 295
Suchen und Ersetzen 81
Suchpfad, sys.path 148
Syntax 26
Syntax-Colouring 43
Syntaxfehler 26
sys.path 148

T

this 323
tkinter, colorchooser() 352
Top-down-Entwurf 158
True 118
try ... except 328
Tupel 183, 184
 entpacken 197
Turtle-Grafik 49, 50
 Farben 224
Turtle-Grafik-Anwendungen,
 scribble.py 337
Turtle-Grafik-Funktion
 back() 69
 begin_fill() 59
 circle() 70
 dot() 168
 end_fill() 60
 fillcolor() 59
 forward() 52
 left() 53
 onclick() 334
 pencolor() 55
 pendown 68
 pensize() 55
 penup 68
 reset() 55
 right() 53
 setheading() 212
 setup() 180
 shape() 72
 speed() 90



stamp() 236
tracer() 213
Turtle-Grafik-Funktionen, einfache
Übersicht 73, 227
Turtle-Grafik-Programm
 dreieck.py 65, 78
 quadrat.py 55, 88
Turtle-Methode
 begin_poly() 360
 end_poly() 360
 filling() 342
 get_poly() 360
 goto() 336
 ondrag() 340
 shapesize() 338
 write() 369
Turtle-Modul numpinput() 96
Turtle-Shape
 benutzerdefiniert 361
 eigenes erzeugen 359
Typ, fauler 184

U

Uhrzeit 367
Unicode-Zeichen smiley 182
UnterkLASSE 376

V

Variable 87
 globale 103, 114
 lokale 103, 113
Variablenname
 erklärender 152
Vererbung 383, 389
Vergleich
 zweier Objekte 118
Verschlüsseln 320
Verzweigung 123
 mehrfach 127

W

Wertetabelle 265
Wertevorrat, dynamischer 184
Wertzuweisung 79, 84, 87
while-Schleife 237
Wochentag 367
Wörterbuch 316
Wurzel berechnen 30

Z

Zählschleife 181
Zeichenkette 32
Zeitmessung 267
Zufallsgenerator 230
Zuweisung 84
 an Listenelemente 299
Zuweisungsoperator 85