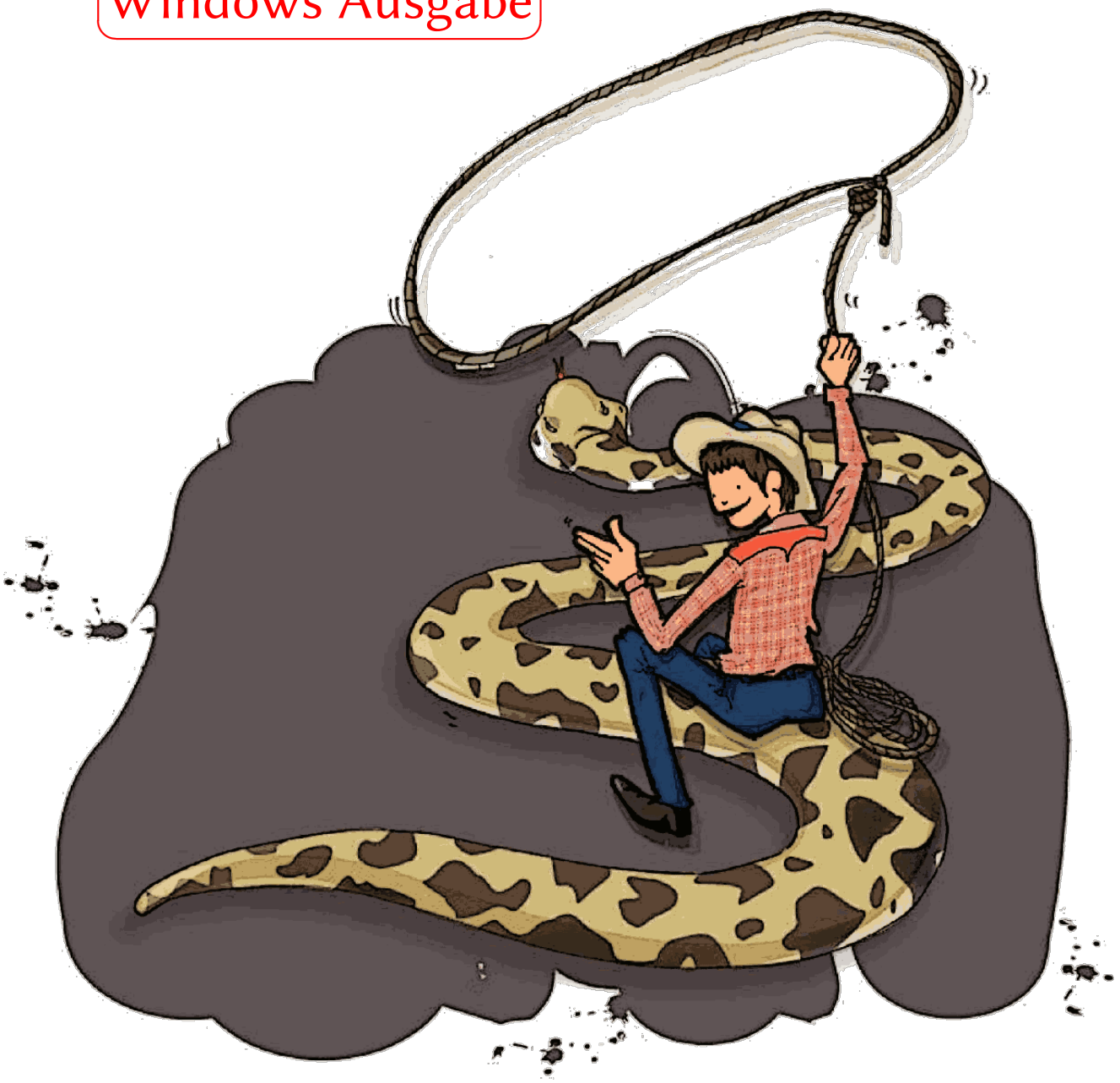
Schlangengerangel für Kinder

Programmieren lernen mit Python

Windows Ausgabe



Geschrieben von Jason R. Briggs
Übersetzt von Joe Ehrensberger

Snake Wrangling for Kids, Learning to Program with Python
Written by Jason R. Briggs

Übersetzung ins Deutsche: Joe Ehrensberger
Version 2012-02

Webseite der Übersetzung: http://code.google.com/p/swfk-de/

Herausgegeben von... eigentlich Niemanden.

Umschlaggestaltung und Illustrationen von Nuthapitol C.

Webseite:
http://www.briggs.net.nz/log/writing/snake-wrangling-for-kids

Danksagung:
An Guido van Rossum (für seine wohlwollende Diktatur über Python), die Mitglieder von der Edu-Sig Mailing Liste (für den hilfreichen Rat und die guten Kommentare), dem Author David Brin (dem Impulsgeber dieses Buches), Michel Weinachter (für das Bereitstellen von Illustrationen höherer Qualität), und unzählige andere Leute für Rückmeldungen und Korrekturvorschläge, wie: Paulo J. S. Silva, Tom Pohl, Janet Lathan, Martin Schimmels, und Mike Cariaso (unter anderen). Falls jemand auf dieser Liste vergessen worden ist, der nicht vergessen werden hätte sollen, so ist dies vollständig der vorzeitigen Senilität des Autors zu verdanken.

Inhaltsverzeichnis

Vorwort

Ein Hinweis für Eltern...

Liebe Eltern,

damit ihr Kind programmieren anfangen kann, müssen Sie Python auf ihrem Computer installieren. Dieses Buch ist vor kurzem an Python 3.1 angepasst worden - diese neue Version von Python ist nicht kompatibel mit früheren Versionen. Wenn Sie also eine ältere Version von Python installiert haben, sollten Sie auf eine ältere Version des Buchs zurückgreifen.

Die Installation von Python ist keine Hexerei, aber es gibt einige Besonderheiten zu beachten, abhängig vom verwendeten Betriebssystem. Wenn Sie gerade einen neuen Computer gekauft haben und keine Ahnung haben, wie Sie die Installation anstellen sollen, ist es wahrscheinlich besser jemanden zu finden, der das für Sie macht. Abhängig vom PC und der Geschwindigkeit der Internetverbindung kann die Installation zwischen 15 Minuten und einigen Stunden dauern.
Gehen Sie zuerst zu www.python.org und laden die neueste Version von Python 3 herunter. Zur Zeit des Schreibens ist dies:

http://www.python.org/ftp/python/3.1.1/python-3.1.1.msi

Zum Installieren genügt ein Doppelklick auf die eben heruntergeladene Datei (Sie erinnern sich noch wohin Sie es gespeichert haben, oder?), und folgen den Anweisungen um es im Standardordner zu installieren (das ist wahrscheinlich *c:\Python31* oder etwas sehr ähnliches).
Nach der Installation. . .
. . .Die ersten Kapitel über sollten Sie Ihr Kind noch am Computer begleiten. Dann wird es hoffentlich die Kontrolle über die Tastatur übernehmen wollen und selber weiter experimentieren. Vorher sollten sie aber wenigstens wissen, wie man einen Texteditor verwendet (nein, kein Textverarbeitungsprogramm wie Microsoft World oder Open Office—ein einfacher altmodischer Texteditor)—und sie sollten zumindest Dateien öffenen und schließen, sowie die Änderungen in den Textdateien speichern können. Von da an wird dieses Buch versuchen die Grundlagen beizubringen.

Danke für deine Zeit und liebe Grüße,
DAS BUCH

1

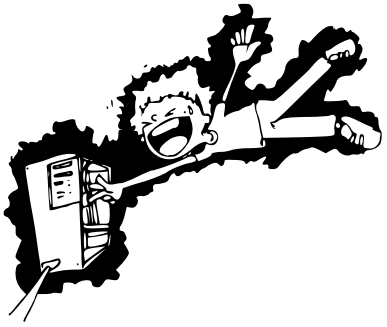
Nicht alle Schlangen beißen

Vielleicht hast du dieses Buch zu deinem Geburtstag bekommen. Oder möglicherweise zu Weihnachten. Tante Helga wollte dir schon wieder ein Paar Socken schenken, die dir viel zu groß sind (und du nebenbei gar nicht anziehen wollen würdest, wenn du reinwächst). Stattdessen hat Tante Helga von diesem druckbaren Buch gehört und sich an dieses Ding erinnert, was bei dir daheim steht und sich Computer nennt. Du wolltest ihr den Computer letztes Weihnachten erklären, hast dann aber aufgegeben, als sie mit der Maus zu sprechen anfing. Sei einfach dankbar, dass du nicht die muffigen Socken bekommen hast. Ich hoffe du bist nicht zu enttäuscht, als ich aus der bunten Geschenksverpackung rausgesprungen bin. Ein nicht ganz so gesprächiges (gut, gar nicht sprechendes) Buch, mit diesem verdächtigen Titel über "Lernen. . .". Aber nimm dir einen Moment Zeit und überlege wie ich mich fühle. Wenn du der Zauberer aus dem Geschichtenbuch wärst, hätte ich vielleicht Zähne… oder sogar Augen. Vielleicht hätte ich bewegende Bilder auf den Seiten, oder könnte gespenstische Geräusche von mir geben, wenn du mich öffnest. Stattdessen bin ich auf gewöhnlichen A4 Seiten gedruckt, habe Eselsohren und bin zusammengestapelt oder in einen Ordner einsortiert. Woher sollte ich das wissen—ich habe ja keine Augen.

Was würde ich für ein Paar schöner scharfer Zähne geben. . .

Aber es ist nicht so schlimm wie es klingt. Auch wenn ich nicht reden kann… oder dir in die Finger beißen, wenn du nicht hinschaust… Ich kann dir ein wenig darüber erzählen, wie Computer funktionieren. Nicht die physikalischen Dinge wie Strom, Kabel und Computerchips, bei denen du wahrscheinlich einen Stromschlag bekommt, wenn du sie anfasst (tu es bitte nicht), sondern die Dinge die Computer eigentlich nützlich machen.

Es ist ein wenig wie Gedanken, die in deinem Kopf herumschwirren. Wenn du keine Gedanken hättest, würdest du am Boden des Schlafzimmers hocken und abwesend an die Decke starren und das T-Shirt ansabbern. Ohne *Programme* wären Computer nur als Türstopper verwendbar—und sogar das mehr schlecht als recht, weil man dauernd in der Nacht darüber stolpern würde. Und es gibt nichts schlimmeres als ein angestoßener blauer Zeh in der Nacht.



Ich bin nur ein Buch, aber sogar ich weiß das.

Deine Familie hat vielleicht eine Playstation, Xbox oder Wii im Wohnzimmer stehen—die werden erst interessant wenn man ein Programm (Spiel) hat. Der DVD Player, vielleicht der Kühlschrank und sogar euer Auto haben alle Programme damit sie nützlicher werden. Der DVD Player hat ein Programm, damit die DVD richtig abgespielt werden kann; der Kühlschrank damit er Energie spart und trotzdem die Speisen kühl hält; und das Auto könnte ein Computerprogramm haben, das dem Fahrer beim Einparken hilft.
Wenn du weißt, wie man Computer Programme schreibt, kannst du alle möglichen nützlichen Dinge damit tun. Vielleicht schreibst du dein eigenes Spiel. Baust Webseiten, die sogar echte Dinge tun anstatt nur bunt auszusehen. Programmieren können kann dir vielleicht bei der Hausübung helfen.
Genug geredet, gehen wir zu etwas Interessanterem.

1.1 Ein paar Worte über Sprachen

Genau wie Menschen, mit Sicherheit auch Wale, vielleicht Delphine und vielleicht sogar Eltern (darüber lässt sich streiten), haben Computer eine eigene Sprache. Eigentlich haben sie genau wie Menschen mehrere Sprachen. Da gibt es Sprachen, die das ganze Alphabet abdecken. A, B, C, D und E sind nicht nur Buchstaben, sondern das sind auch Programmiersprachen (was beweist, dass Erwachsene keine Phantasie haben, und sollten ein Wörterbuch oder Synonymwörterbuch lesen müssen, bevor sie irgendwas benennen).

Da gibt es Progammiersprachen die nach Leuten benannt sind, einfache Akronyme (die ersten Buchstaben von einer Wortfolge) oder nach Fernsehserien benannt sind. Und wenn man ein Plus oder Eine Raute (+, #) hinten anhängt, hat man noch einige Sprachen. Was die Sache noch schlimmer macht ist, dass sich einige Sprachen fast nicht unterscheiden und nur ein wenig verschieden sind.

Was habe ich dir gesagt? Keine Phantasie!

Zum Glück werden viele Sprachen kaum mehr verwendet oder sind ganz verschwunden, aber die Liste der verschiedenen Wege mit einem Computer zu ‘reden’ ist immer noch beunruhigend lang. Ich werde dir nur eine vorstellen—ansonsten würden wir gar nicht anfangen.
Da wäre es produktiver im Schlafzimmer zu sitzen und das T-Shirt anzusabbern. . .

1.2 Ungiftige Würgeschlagen. . .

. . .kurzgesagt: Python.

Abgesehen davon, dass Pythons Schlangen sind, gibt es auch die Programmiersprache Python. Die Sprache wurde aber nicht nach dem beinlosen Reptil benannt, sondern nach einer Fernsehsendung, die in den siebziger Jahren populär war. Monty Python war eine britische Komödie (die heute immer noch populär ist), die man erst in einem bestimmten Alter lustig findet. Kinder unter. . . sagen wir zwölf Jahren werden sich über die Begeisterung wundern [1].

Es gibt eine Anzahl von Dingen die Python (die Programmiersprache, nicht die Schlange und auch nicht die Fernsehsendung) sehr nützlich machen, wenn man anfängt zu programmieren. Für uns ist der wichtigste Grund fürs Erste, dass man sehr schnell Ergebnisse erzielen kann.

Jetzt kommt der Teil an dem du darauf hoffst, dass sich deine Mutter, Vater (oder wer auch immer den Computer bedient), das Vorwort diese Buches mit dem Titel “Ein Hinweis für Eltern” gelesen haben.
Das hier ist ein guter Weg das herauszufinden.

Klick auf das Startsymbol unten links auf dem Monitor. Gehe dann auf ‘alle Programme’ (hat ein grünes Dreieck) und in der Liste siehst du hoffentlich ‘Python 3.1’ (oder so was ähnliches). Abbil-

[1] außer dem fish slapping Dance. Der ist lustig, egal wie alt man ist.

Abbildung 1.1: Python im Windows Menu.

dung 1.1 zeigt dir wonach du Ausschau halten sollst. Klicke auf 'Python (Kommandozeile) und ein Fenster wie in Abbildung 1.2 geht auf.

Falls du entdeckst hast, dass deine Eltern das Vorwort nicht gelesen haben. . .

. . .weil irgendetwas nicht mit der vorherigen Anleitung zusammenstimmt—dann blättere zurück und halte deinen Eltern das Buch unter die Nase während sie gerade die Zeitung lesen und schaue sie erwartungsvoll an. Immer wieder "bitte bitte bitte bitte" sagen kann dich auch weiterbringen, wenn sie nicht von der Couch aufstehen wollen. Was du immer noch machen kannst, ist das Vorwort zu lesen und selber auszuprobieren Python zu installieren.

1.3 Dein erstes Python Programm

Wenn du mit etwas Glück an dem Punkt angelangt bist und die Python Konsole geöffnet hast, kann es schon losgehen. Du öffnest also die Konsole das erste Mal und siehst den sogenannten 'Prompt' (den Prompt siehst du auch nach dem Ausführen eines Kommandos). In der Python Konsole, besteht dieser Prompt aus drei größer-als Zeichen (>), die nach rechts zeigen:

```
>>>
```

Wenn du mehrere Python Befehle zusammenbastelst erhältst du ein Programm, welches du auch ausserhalb der Konsole verwenden kannst. . . aber fürs Erste werden wir die Dinge einfach halten und direkt mit der Konsole nach dem Prompt (>>>) arbeiten. Probier es aus und tippe folgendes:

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

Pass auf, dass du auch die Anführungszeichen verwendest (das sind diese " ") und drücke danach die Eingabetaste. Hoffentlich siehst du etwas wie dieses:

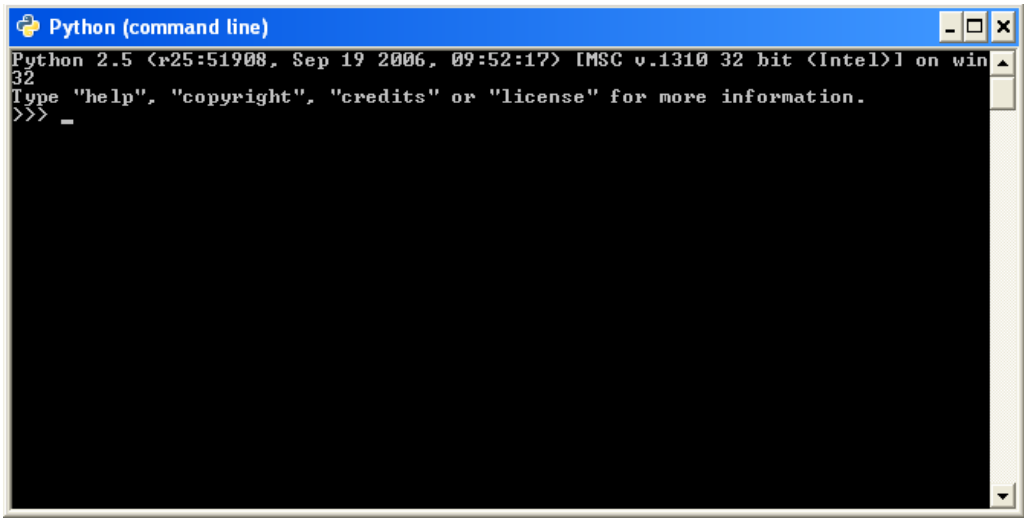


Abbildung 1.2: Die Python Konsole auf Windows.

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

Der Prompt erscheint danach wieder um dir zu sagen, dass die Python Konsole wieder bereit für neue Befehle ist.

Gratulation! Du hast gerade dein erstes Python Programm geschrieben. print ist eine Funktion, die alles innerhalb der Klammern auf die Konsole ausgibt–wir werden das noch öfter verwenden.

1.4 Dein zweites Python Programm. . .schon wieder das gleiche?

Python Programme wären nicht besonders nützlich, wenn du jedes Mal, wenn du etwas tun willst—oder ein Programm für jemanden schreiben willst, alles neu eintippen müsstest.

Das Textverarbeitungsprogramm mit dem du vielleicht deine Hausaufgaben schreibst ist vielleicht zwischen 10 und 100 Millionen Zeilen lang. Abhängig davon wieviele Zeilen du auf ein Blatt druckst (und ob du beidseitig druckst oder nicht) wären das ungefähr 400.000 gedruckte Seiten. . . oder ein 40 Meter hoher Stapel. Stell dir vor, dass du die Software so vom Laden nach Hause bringen würdest. Da müsstest du einige Male hin und her laufen um das ganze Papier nach Hause zu bringen.

. . .und es wäre besser wenn überhaupt kein Wind weht, während du das Papier nach Hause schleppst. Zum Glück gibt es eine Alternative—sonst würde niemand etwas sinnvolles erreichen.

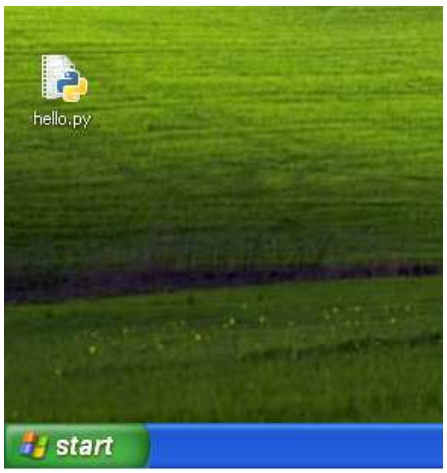


Abbildung 1.3: hallo.py Symbol auf dem Windows Desktop.

Öffene Notpad (Klicke auf Start, Alle Programme und unter Zubehör sollte Notepad sein), und gib dann das gleiche in die Textdatei ein wie vorher in der Konsole:

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

Danach einfach im Datei Menu (in Notepad) auf Speichern gehen und wenn nach dem Dateinamen gefragt wird, *hallo.py* benennen und auf dem Desktop speichern. Wenn du nun auf dem Desktop Symbol hallo.py doppelklickst (siehe Abbildung 1.3) wird für einen kurzen Moment die Konsole erscheinen und wieder verschwinden. Es wird zu schnell verschwinden, als die Wörter zu erkennen, aber die Wörter Hallo Welt wird auf dem Monitor für den Bruchteil einer Sekunde erschienen sein—wir kommen nochmals auf das zurück und beweisen, dass es wirklich so ist.

Wie du siehst, sparst du dir damit die Arbeit die gleichen Dinge immer wieder und wieder einzutippen. So war es in den 1980'er Jahren. Ich mache keine Witze—früher war es so. Geh und frag

deinen Vater ob er jemals einen ZX81 verwendet hat, als er jünger war.

Wenn ja, kannst du mit dem Finger auf ihn zeigen und lachen.

Bei dem musst du mir vertrauen. Du wirst es nicht verstehen. Aber dein Vater wird. [2]
Sei aber vorbereitet davonzulaufen.

Das Ende des Anfangs

Willkommen in der wunderbaren Welt der Programmierung. Wir haben ganz einfach angefangen mit einem "Hallo Welt" Programm—jeder fängt mit diesem an wenn man programmieren lernt. Im nächsten Kapitel werden wir anfangen einige etwas sinnvollere Dinge mit der Python Konsole zu machen und lernen wie man Programme aufbaut.

[2] Der Sinclair ZX81, der in den 1980'er herauskam, war einer der ersten Computer, die man sich leisten konnte. Viele junge Jungen und Mädchen haben sich halb totgeärgert, als sie den Code aus dem populären ZX81 Magazin abtippten nur um später festzustellen, dass das verdammte Ding überhaupt nie richtig funktionierte.

2 8 multipliziert mit 3,57 ergibt. . .

Was ergibt 8 mal 3,57? Du müsstest einen Taschenrechner verwenden, oder? Oder du bist besonders gut im Kopfrechnen und kannst Multiplikationen mit Kommastellen im Kopf rechnen—aber das ist nicht der Punkt. Du kannst auch die Python Konsole dafür verwenden. Starte wieder die Konsole (siehe Kapitel 1 für genauere Infos, falls du es aus seltsamen Gründen übersprungen hast), und wenn du den Prompt siehst, tippst du $8 * 3.57$ ein und drückst die Enter-Taste:

```
Python 3.1.1+ (r311:74480, Nov  2 2009, 14:49:22)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 8 * 3.57
28.56
```

Die Sterntaste (*) wird anstelle des normalen Multiplikationszeichen X verwendet, welches du in der Schule verwendest. Die Sterntaste mußt du verwenden, damit der Computer weiß, ob du den Buchstaben *x* oder das Multiplikationszeichen X meinst. Wie wäre es nun mit einer Rechnung, die etwas interessanter ist?

Nehmen wir an, du machst Hausarbeiten und bekommst 5 € Taschengeld dafür, trägst die Zeitung fünf mal die Woche aus und bekommst dafür 30 €—wie viel Geld würdest du in einem Jahr verdienen?

Python ist kaputt!?!?

Wenn du gerade mit einem Taschenrecher 8 x 3.57 ausgerechnet hast, wird folgendes auf der Anzeige erscheinen:

28.56

Warum gibt Python manchmal ein anderes Ergebnis aus?

Eigentlich nicht. Der Grund warum manchmal die Kommastellen bei Python nicht bis aufs letzte Detail mit dem Ergebnis des Taschenrechners zusammenpassen, liegt in der Art wie Computer mit Fließkommazahlen (Zahlen mit Kommastellen) umgehen. Es ist ein wenig kompliziert und für Anfänger verwirrend. Am besten man merkt sich, wenn man mit Kommazahlen arbeitet, das *manchmal* das Ergebnis nicht exakt dem entspricht, was man erwarten würde. Das stimmt für Multiplikation, Division, Addition und auch Subtraktion.

Auf Papier würden wir folgendes schreiben:

```
(5 + 30) x 52
```

Das ist 5 € + 30 € multipliziert mit 52 Wochen im Jahr. Natürlich sind wir so schlau und wissen, dass 5 + 30 eigentlich 35 ergibt, und so ergibt sich die Formel:

```
35 x 52
```

Was einfach genug ist, es auf Papier oder im Taschenrechner zu rechnen. Aber wir können alles auch mit der Python Konsole erledigen.

```
>>> (5 + 30) * 52
1820
>>> 35 * 52
1820
```

Was passiert also, wenn du jede Woche 10 € ausgibst? Wie viel hast du dann am Ende des Jahres übrig? Wir könnten die Formel auf Papier aufschreiben, aber probieren wir es in der Konsole.

```
>>> (5 + 30 - 10) * 52
1300
```

Das sind 5 € plus 30 € minus 10 € multipliziert mit 52 Wochen im Jahr. Am Ende des Jahres wären 1300 € übrig. Gut, das ist jetzt immer noch nicht so besonders spannend. Das hätten wir auch alles mit einem Taschenrechner machen können. Aber wir kommen da nochmals zurück und zeigen, wie das noch nützlicher und spannender wird.

Du kannst in der Python Konsole Multipizieren, Addieren, Substrahieren und auch Dividieren, aber auch andere mathematische Funktionen verwenden, auf die wir später eingehen werden. Fürs Erste sind die wichtigsten mathematischen Symbole in Python (eigentlich heißen sie Operatoren):

+	Addition
-	Subtraktion
*	Multiplikation
/	Division

Der Grund warum auf Computern der Schrägstrich (/) für die Division verwendet wird, ist der, dass es das Divisionssymbol ÷ nicht auf die Tastatur geschafft hat. Wenn du als Beispiel 100 Eier und 20 Schachteln hättest, dann möchtest du vielleicht wissen wie viele Eier du pro Schachtel einpacken kannst. Du würdest folgendermaßen die 100 durch 20 dividieren:

$$\frac{100}{20}$$

Oder du könntest es folgendermaßen schreiben:

$$100 \div 20$$

In Python würdest du die Division aber so notieren "100 / 20".
Das ist auch viel einfacher denke ich. Aber was weiß ich schon. Ich bin nur ein Buch.

2.1 Verwendung von Klammern und die "Reihenfolge der Operationen"

Wir verwenden Klammern in Programmiersprachen um die "Reihenfolge der Operationen" zu bestimmen. Bei einer Operation verwendet man Operatoren (einen aus der Liste von vorhin). Es gibt mehr Operatoren als in der kurzen Liste stehen, aber fürs Erste genügt zu wissen, dass Multiplikation und Division einen höheren Rang haben als Addition und Subtraktion. Das bedeutet, dass du in einer Rechnung zuerst den Teil mit der Multiplikation und Division berechnest und dann erst die Addition und Subtraktion. In der folgenden Formel kommen nur Additionen vor (+) und die Zahlen werden der Reihenfolge nach addiert:

```
>>> print(5 + 30 + 20)
55
```

In der nächsten Formel gibt es nur Addition und Subtraktion. Deswegen betrachtet Python die Zahlen einfach der Reihe nach wie sie dastehen:

```
>>> print(5 + 30 - 20)
15
```

Aber in der folgenden Formel gibt es zusätzlich noch eine Multiplikation. Deswegen werden die Zahlen 30 und 20 zuerst behandelt. Anstatt die Formel aufzuschreiben, könnte man sagen: "multipliziere 30 mit 20 und addiere 5 zum Ergebnis" (die Multiplikation kommt zuerst, weil sie einen höheren Rang hat als die Addition):

```
>>> print(5 + 30 * 20)
605
```

Und wenn wir Klammern hinzufügen? Dann schaut es so aus:

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

Warum kommt jetzt ein anderes Ergebnis raus? Weil Klammern die Reihenfolge der Berechnung bestimmen. Wenn es Klammern gibt, rechnet Python den Inhalt der Klammern zuerst, dann erst den Rest außerhalb der Klammer. Somit könnte man anstatt der Formel auch sagen: "Addiere 5 zu 30 und das Ergebnis wird mit 20 multipliziert". Die Verwendung von Klammern kann auch komplizierter werden. Es können Klammern innerhalb von Klammern stehen.

```
>>> print(((5 + 30) * 20) / 10)
70
```

In diesem Fall wird Python zuerst die **innerste** Klammer ausrechnen, dann die äußere Klammern und dann den übrigen Operator behandeln. Hier könnte man sagen: "Addiere 5 und 30, multipliziere das Ergebnis mit 20 und das Ergebnis wird dann durch 10 dividiert". Das Ergebnis ohne Klammern wäre verschieden.

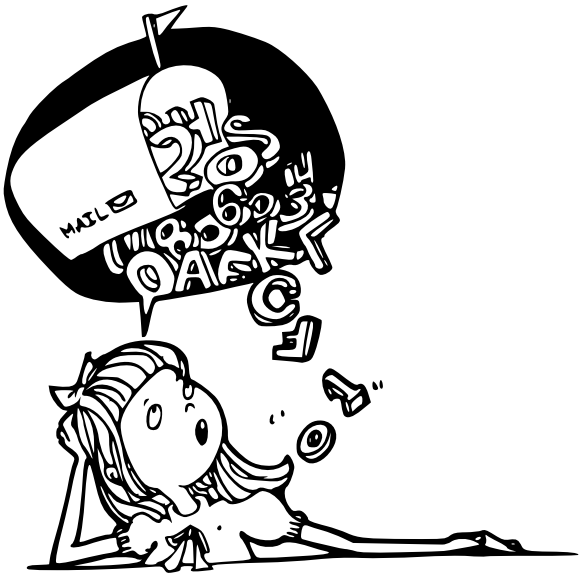
```
>>> 5 + 30 * 20 / 10
65
```

Ohne Klammern wird zuerst 30 mit 20 multipliziert, das Ergebnis durch 10 dividiert und am Ende kommt 5 dazu.

Merke dir einfach, dass Multiplikation und Division immer vor Addition und Subtraktion kommen, außer es werden Klammern verwendet, die als erste behandelt werden. (Merksatz: Punkt vor Strichrechnung.

2.2 Es gibt nichts unbeständigeres als eine Variable

Eine 'Variable' ist ein Platz zum Speichern von Dingen. Diese 'Dinge' können Nummern, Text, Listen von Zahlen oder auch eine Vielzahl von anderen Dingen sein. Eine Variable ist so etwas wie ein Briefkasten.



Du kannst verschiedene Dinge in den Briefkasten werfen wie einen Brief oder ein Paket, so wie man Dinge (Nummern, Text, Listen von Zahlen und so weiter, und so weiter) in einer Variable speichert. So wie dieser Briefkasten funktionieren viele Programmiersprachen. Aber nicht alle.

In Python sind Variablen etwas verschieden. Eine Variable ist weniger ein Briefkasten sondern viel mehr die Beschriftung auf dem Briefkasten drauf. Du kannst die Beschriftung wegnehmen und woanders draufkleben oder sogar (mit der Hilfe von einem Faden) mit mehreren Dingen verbinden. Wir erzeugen eine Variable indem wir ihr einen Namen geben und das Gleichheitszeichen (=) verwenden. Damit sagen wir Python wohin der Name zeigen soll. Zum Beispiel:

```
>>> Fritz = 100
```

Wir haben eben die Variable 'Fritz' erzeugt und gesagt, dass sie auf die Zahl 100 zeigen soll. Es ist ein wenig so wie wenn man Python sagt: merk dir diese Nummer, weil ich werde sie später nochmal brauchen. Um herauszufinden wohin die Variable zeigt, kannst du die Funktion 'print' in der Python Konsole eingeben, gefolgt vom Variablenname. Drücke danach Enter (die Eingabetaste). Zum Beispiel:

```
>>> Fritz = 100
>>> print(Fritz)
100
```

Wir können Python auch sagen, dass die Variable Fritz auf etwas anderes zeigen soll:

```
>>> Fritz = 200
>>> print(Fritz)
200
```

Auf der ersten Zeile sagen wir, dass Fritz auf die Nummer 200 zeigen soll. Auf der zweiten Zeile fragen wir Fritz, wohin er zeigt um zu beweisen, dass Fritz jetzt auf 200 zeigt. Es können auch mehrere Namen auf das gleiche Ding zeigen.

```
>>> Fritz = 200
>>> Franz = Fritz
>>> print(Franz)
200
```

Im Code oben geben wir an, dass der Name (oder Beschriftung) Franz auf das gleiche Ding wie Fritz zeigen soll. Natürlich ist 'Fritz' kein besonderers guter Name für eine Variable. Es sagt uns nichts darüber aus, wofür die Variable verwendet wird. Bei einem Postkasten ist es einfach. Der wird für die Post verwendet. Aber eine Variable kann auf ganz verschiedene Dinge zeigen. Deswegen werden wir normalerweise sinnvollere Namen verwenden.

Nehmen wir an, dass du eben die Python Konsole gestartet hast und 'Fritz = 200' eingegeben hast. Dann gehst du für 10 Jahre auf Weltreise. Du wanderst durch die Sahara, bezwingst den Mount Everest und segelst über den Atlantik—danach kommst du wieder an deinen Computer zurück. Würdest du wissen wofür die Zahl 200 stand und wofür du sie verwendet hast?
Das glaube ich nicht.
Ich habe mir das alles gerade ausgedacht und weiß selber nicht was 'Fritz = 200' bedeutet (außer einem *Namen* der auf die Nummer *200* zeigt). Nach 10 Jahren hast du überhaupt keine Chance dich an die Bedeutung zu erinnern.

Aha! Aber was wäre wenn wir die Variable *anzahl_von_studenten* getauft hätten?

```
>>> anzahl_von_studenten = 200
```

Das können wir deshalb machen, weil Variablennamen aus normalen Buchstaben, Zahlen und Unterstrichen bestehen können. Das erste Zeichen darf aber keine Zahl sein. Wenn du nun nach 10 Jahren die Variable 'anzahl_von_studenten' siehst, wird es immer noch Sinn machen. Du kannst dann folgendes eintippen:

```
>>> print(anzahl_von_studenten)
200
```

Und du würdest sofort wissen, dass es sich um 200 Studenten handelt. Es ist nicht immer so wichtig sich sinnvolle Namen für Variablen auszudenken. Du kannst auch einzelne Buchstaben oder ganze Sätze verwenden; und manchmal, wenn man schnell etwas schreibt, sind kurze Namen genauso praktisch. Es hängt davon ab, ob du später nochmals in den Code reinschauen wirst und rausfinden willst, was du dir damals dabei gedacht hast.

```
das_ist_auch_eine_gueltige_variable_aber_vielleicht_nicht_so_praktisch
```

2.3 Variablen verwenden

Jetzt wissen wir wie man Variablen erzeugt. Aber wie wenden wir sie an? Erinnerst du dich an die Rechnung von früher? Die Berechnung wieviel Geld du am Ende des Jahres gespart haben würdest, wenn du 5 € für die Hausarbeit bekommst und 30 € fürs Zeitung austragen. 10 € pro Woche gibst du aus. Bis jetzt schaut das so aus:

```
>>> print((5 + 30 - 10) * 52)
1300
```

Was, wenn wir die ersten drei Nummern in Variablen verwandeln? Probiere folgendes aus:

```
>>> Hausarbeit = 5
>>> Zeitung_austragen = 30
>>> Ausgaben = 10
```

Somit haben wir eben die Variablen 'Hausarbeit', 'Zeitung_austragen' und 'Ausgaben' angelegt. Nun können wir die Rechnung so schreiben:

```
>>> print((Hausarbeit + Zeitung_austragen - Ausgaben) * 52)
1300
```

Es kommt genau das Gleiche raus. Wenn du nun 2 € mehr für die Hausarbeit bekommst, kannst du einfach die 'Hausarbeit' Variable auf 7 setzen. Danach drückst du die 'Pfeil nach oben Taste' (↑) und wenn die Rechnung wieder erscheint, drückst du Enter.

```
>>> Hausarbeit = 7
>>> print((Hausarbeit + Zeitung_austragen - Ausgaben) * 52)
1404
```

Das bedeutet schon eine Menge weniger Tippen um herauszufinden dass am Ende des Jahres 1404 € übrig sind. Du kannst die anderen Variablen auch verändern und schauen wie sich das aufs Ergebnis der Rechnung auswirkt.

```
Wenn du doppelt so viel in der Woche ausgibst:
>>> Ausgaben = 20
>>> print((Hausarbeit + Zeitung_austragen - Ausgaben) * 52)
884
```

So bleiben dir also 884 € am Ende des Jahres übrig. Fürs arbeiten mit Variablen ist es fürs erste ausreichend, wenn man versteht, dass Variablen zum Speichern von Dingen verwendet werden.
Denk an einen Postkasten der eine Beschriftung hat!

2.4 Ein String?

Wenn du ein wenig aufpasst und die Seiten nicht nur flüchtig überfliegst, dann erinnerst du dich vielleicht, dass sich Variablen für alle möglichen Dinge eignen—nicht nur Zahlen. Beim Programmieren wird ein Text 'String' benannt. Das klingt ein wenig ungewohnt; aber String ist das englische Wort für Schnur und so Wörter kann man sich als aufgefädelte Buchstaben vorstellen. Dann macht die Bezeichung 'String' für Text vielleicht etwas mehr Sinn.
Aber vielleicht auch nicht.

In diesem Fall solltest du dir einfach merken, dass ein String eine Ansammlung von Buchstaben, Ziffern und anderen Zeichen ist, die irgendwie sinnvoll zusammenhängen. Alle diese Buchstaben, Ziffern und Symbole in diesem Buch könnten einen String ergeben. Dein Name könnte ein String sein. Oder auch deine Adresse. Das erste Python Programm, welches wir in Kapitel 1 geschrieben haben, verwendet den String 'Hallo Welt'.

In Python erzeugen wir einen String indem wir Anführungszeichen um den Text angeben. Wir könnten unsere nutzlose Fritz Variable nehmen und mit einem String verbinden.

```
>>> Fritz = "Ja, das ist ein String"
```

Und wir können nachschauen, was in der Variable Fritz gespeichert ist, indem wir print(Fritz) eintippen.

```
>>> print(Fritz)
Ja, das ist ein String
```

Wir können auch einfache Anführungszeichen verwenden, um einen String zu erstellen.

```
>>> Fritz = 'Das ist jetzt ein anderer String'
>>> print(Fritz)
Das ist jetzt ein anderer String
```

Wenn du aber mehr als eine Zeile Text der Variable zuweisen willst und Einfache (') oder Doppelte (") Anführungstriche verwendest, wirst du folgenden Fehler in der Konsole sehen:

```
>>> Fritz = "Dieser String hat jetzt zwei
  File "<stdin>", line 1
    Fritz = "Das sind jetzt zwei
                              ^
SyntaxError: EOL while scanning string literal
>>>
```

Wir werden die Fehler später noch genauer betrachten, aber wenn du fürs Erste Strings mit mehreren Zeilen haben willst, kannst du 3 einfache Anführungsstriche verwenden:

```
>>> Fritz = '''Das sind jetzt zwei
... Zeilen Text in einem einzigen String'''
```

Gib den Inhalt der Variable aus, um zu sehen, ob es funktioniert hat:

```
>>> print (Fritz)
Das sind jetzt zwei
Zeilen Text in einem einzigen String
```

Ganz nebenbei: du wirst diese 3 Punkte (...) noch ein paar mal sehen, wenn du etwas eintippst, was sich über mehrere Zeilen zieht (wie ein mehrzeiliger String).

2.5 Tricks mit Strings

Hier ist eine interessante Frage: was ergibt 10 * 5 (10 mal 5)? Die Antwort ist natürlich 50.
Gut, das war jetzt überhaupt nicht interessant.

Aber was ist 10 * 'a' (10 mal der Buchstabe a)? Das ist eine komische Frage, aber hier kommt die Antwort aus der Python Welt.

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

Das geht auch mit Strings, die aus mehreren Zeichen bestehen:

```
>>> print(20 * 'abcd')
abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcd
```

Ein anderer Trick mit Strings ist das Einbauen von Platzhaltern. Du kannst das machen indem du %s verwendest, was genau einen Platz bereithält. Da könnte auch ein String darin Platz finden. Machen wir ein Beispiel.

```
>>> mein_text = 'Ich bin %s Jahre alt'
>>> print(mein_text % 12)
Ich bin 12 Jahre alt
```

In der ersten Zeile erzeugen wir die Variable mein_text die einen String und einen Platzhalter (%s) enthält. Das kleine Symbol %s steht da und sagt zur Python Konsole: “ersetze mich durch irgend etwas”. Also rufen wir in der nächsten Zeile den print(mein_text) Befehl auf. Darin verwenden wir das % Symbol und sagen Python das es den Platzhalter mit der Nummer 12 ersetzen soll. Den String können wir auch mit anderen Zahlen wiederverwenden.

```
>>> mein_text = 'Hallo %s, wie geht es dir heute?'
>>> name1 = 'Romeo'
>>> name2 = 'Julia'
>>> print(mein_text % name1)
Hallo Romeo, wie geht es dir heute?
>>> print(mein_text % name2)
Hallo Julia, wie geht es dir heute?
```

Im vorigen Beispiel haben wir 3 Variablen erzeugt (mein_text, name1, name2). Die erste Variable enthält den Text mit dem Platzhalter. Danach geben wir die Variable aus und verwenden den Platzhalter % um die Variablen ‘name1’ und ‘name2’ mitzugeben. Du kannst auch mehr als einen Platzhalter verwenden.

```
>>> mein_text = 'Hallo %s und %s, wie geht es euch heute?'
>>> print(mein_text % (name1, name2))
Hallo Romeo und Julia, wie geht es euch heute?
```

Wenn du mehr als einen Platzhalter verwendest, musst du die mitgegebenen Variablen mit Klammern einpacken—also (name1, name2) ist der richtige Weg um 2 Variablen mitzugeben. Man sagt zu so einem Set von Variablen in runden Klammern auch *Tupel*, und es verhält sich ein wenig wie eine Liste. Darüber wollen wir jetzt reden.

2.6 Fast schon eine Einkaufsliste

Eier, Milch, Käse, Sellerie, Honig und Backpulver. Das ist zwar noch nicht die gesamte Einkaufsliste, aber gut genug für unsere Zwecke. Wenn du das alles in einer Variable speichern willst, dann könnte es so ausschauen:

```
>>> Einkaufsliste = 'Eier, Milch, Käse, Sellerie, Honig, Backpulver'
>>> print(Einkaufsliste)
Eier, Milch, Käse, Sellerie, Honig, Backpulver
```

Die andere Möglichkeit ist es eine ‘Liste’ zu erzeugen, die es in Python auch gibt.

```
>>> Einkaufsliste = [ 'Eier', 'Milch', 'Käse', 'Sellerie',
...'Honig', 'Backpulver' ]
>>> print(Einkaufsliste)
['Eier', 'Milch', 'Käse', 'Sellerie', 'Honig', 'Backpulver']
```

Da musst du schon mehr tippen. Aber dafür kann man mit dieser Liste auch mehr anstellen. Da könnte man gezielt das dritte Ding herauspicken (die dritte Position) und ausgeben. Mit Python funktioniert das mit den eckigen Klammern []:

```
>>> print(Einkaufsliste[2])
Käse
```

Listen beginnen mit der Position 0—also wird der erste Eintrag mit 0 angesprochen, der Zweite mit 1, der Dritte mit 2 und so weiter. Das macht für die meisten Leute keinen Sinn, aber es macht für Programmierer Sinn. Bald fängst du auch mit 0 zählen an, wenn du die Treppen einer Stiege zählst. Das wird deinen kleinen Bruder ziemlich verwirren.

Wir können die Inhalte der Liste von der dritten bis zur fünften Position folgendermaßen ausgeben:

```
>>> print(Einkaufsliste[2:5])
['Käse', 'Sellerie', 'Honig']
```

Mit [2:5] sagen wir Python, dass uns die Position 2 bis 5 (aber 5 ist nicht dabei) interessiert. Und da wir ja mit 0 anfangen zu zählen, wird der dritte Eintrag mit 2 angesprochen und die fünfte Position ist in Python die Nummer 4. In Listen kann man alles mögliche speichern. Darin kann man Nummern speichern:

```
>>> meine_liste = [ 1, 2, 5, 10, 20 ]
>>> print(meine_liste)
```

Und auch Strings:

```
>>> meine_liste = [ 'a', 'bbb', 'ccccccc', 'ddddddddd' ]
>>> print(meine_liste)
```

Oder auch eine Mischung von Zahlen und Strings:

```
>>> meine_liste = [1, 2, 'a', 'bbb']
>>> print(meine_liste)
[1, 2, 'a', 'bbb']
```

Und sogar eine Liste von Listen:

```
>>> liste1 = [ 'a', 'b', 'c' ]
>>> liste2 = [ 1, 2, 3 ]
>>> meine_liste = [ liste1, liste2 ]
>>> print(meine_liste)
[['a', 'b', 'c'], [1, 2, 3]]
```

Im obigen Beispiel erzeugen wir eine Variable mit dem Namen ‘liste1’ und speichern 3 Buchstaben darin. In ‘liste2’ speichern wir 3 Zahlen. Die ‘meine_liste’ wird aus den Inhalten der Variablen ‘liste1’ und ‘liste2’ befüllt. Wenn man Listen von Listen erzeugt, werden die Dinge schnell etwas verwirrend. Zum Glück braucht man das nicht oft. Praktisch zu wissen ist jedoch, dass man alle möglichen Inhalte in Python Listen speichern kann.
Und nicht nur den nächsten Einkauf.

Tausche Dinge aus

Wir können einen Eintrag der Liste austauschen, indem wir ihm einen Wert zuweisen, wie wir das mit normalen Variablen tun. Zum Beispiel könnten wir Sellerie an Position 3 durch Bananen ersetzen.

```
>>> Einkaufsliste[3] = 'Bananen'
>>> print(Einkaufsliste)
['Eier', 'Milch', 'Käse', 'Bananen', 'Honig', 'Backpulver']
```

Dinge hinzufügen...

Wir können aber auch Dinge an die Einkaufsliste dranhängen indem wir die Methode ‘append’ benutzen. Eine Methode ist ein Kommando damit Python weiß was wir tun wollen. Wir kommen etwas später im Buch zu Methoden, aber zuerst schreiben wir noch ein Ding zur Einkaufliste dazu. Das geht so:

```
>>> Einkaufsliste.append('Schokoriegel')
>>> print (Einkaufsliste)
['Eier', 'Milch', 'Käse', 'Bananen', 'Honig', 'Backpulver', 'Schokoriegel']
```

Was die Liste schon um einiges verbessert hat.

. . .und Dinge entfernen

Wir können auch Einträge von der Liste streichen in dem wir das Kommando ‘del’ (was eine Kurzform vom englischen delete ist und so viel heißt wie entfernen) verwenden.

```
>>> del Einkaufsliste[5]
>>> print (Einkaufsliste)
['Eier', 'Milch', 'Käse', 'Bananen', 'Honig', 'Schokoriegel']
```

Da die Positionen mit 0 anfängt, bedeutet Einkaufsliste[5] eigentlich den sechsten Eintrag.

Zwei Listen sind besser als Eine

Wir können auch Listen verbinden, wie man zwei Nummern addiert:

```
>>> liste1 = [ 1, 2, 3 ]
>>> liste2 = [ 4, 5, 6 ]
>>> print(liste1 + liste2)
[1, 2, 3, 4, 5, 6]
```

Wir können auch zwei Liste zusammenfassen und das Ergebnis einer anderen Variable zuweisen:

```
>>> liste1 = [ 1, 2, 3 ]
>>> liste2 = [ 4, 5, 6 ]
>>> liste3 = liste1 + liste2
>>> print(liste3)
[1, 2, 3, 4, 5, 6]
```

Und du kannst auch eine Liste multiplizieren, wie damals einen String:

```
>>> liste1 = [ 1, 2 ]
>>> print(liste1 * 5)
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

Im vorigen Beispiel wird die liste1 mit fünf multipliziert. Das ist ein anderer Weg um Python zu sagen "Wiederhole Liste1 fünf mal". Division (/) und Subtraktion (-) machen aber mit Listen keinen Sinn und du würdest einen Fehler wie diesen bekommen:

```
>>> liste1 / 20
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'list' and 'int'
```

oder

```
>>> liste1 - 20
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'type' and 'int'
```

Du bekommst auf jeden Fall eine garstige Fehlermeldung.

2.7 Tupel und Listen

Wir haben eine Tupel schon früher erwähnt. So ein Tupel ist ein wenig wie eine Liste, aber anstatt der eckigen Klammern werden runde Klammern verwendet—d.h. '(' und ')'. Du kannst Tupel ähnlich verwenden wie eine Liste:

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

Der Hauptunterschied zu Listen ist der, dass sich Tupel nach dem Erstellen nicht mehr ändern können. Wenn du also einen Wert austauschen willst, so wie das bei Listen funktioniert, wirst du bei Tupel eine Fehlermeldung bekommen:

```
>>> t[0] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Das heißt aber nicht, dass du die Variable die das Tupel enthält nicht ändern kannst. Folgendes funktioniert zum Beispiel ganz normal:

```
>>> meine_variable = (1, 2, 3)
>>> meine_variable = [ 'Eine', 'Liste', 'von', 'Strings' ]
```

Zuerst erzeugen wir die Variable meine_variable die auf das Tupel mit den 3 Nummern zeigt. Danach lassen wir meine_variable auf eine Liste von Strings zeigen. Das ist vielleicht etwas verwirrend. Aber denke zum Beispiel an Schließfächer mit Namen drauf. Du gibst etwas in ein Fach, sperrst es zu und dann wirfst du den Schlüssel weg. Dann ziehst du den Namen vom Fach ab und klebst den Aufkleber mit dem Namen auf ein leeres Fach. Da gibst du andere Sachen rein, sperrst zu und diesmal behältst du den Schlüssel. Ein Tupel ist wie ein verschlossenes Schließfach. Den Inhalt kann man nicht ändern. Aber du kannst die Beschriftung runternehmen und auf ein leeres Fach kleben.

2.8 Probiere es aus

In diesem Kapitel haben wir gesehen wie man einfache mathematische Gleichungen mit Hilfe der Python Konsole berechnet. Du hast auch gelernt wie die Verwendung von Klammern das Resultat von Rechnungen beeinflusst in dem die Reihenfolge der Berechnung sich ändert. Du weißt jetzt wie man in Python Werte für späteren Zugriff speichern kann—mit der Verwendung von Variablen—und was 'Strings', Listen und Tupel sind.

Übung 1

Mache eine Liste von deinen Lieblingsspielsachen und nenne die Liste spielsachen. Mache eine Liste von deinen Lieblingsspeisen und nenne diese Liste speisen. Verbinde diese zwei Listen und nenne das Ergebnis favoriten. Verwende print um die Liste der Favoriten auszugeben.

Übung 2

Du hast 3 Schachteln und in jeder sind 25 Schokoladen enthalten. Du hast auch 10 Tüten mit jeweils 32 Süßigkeiten. Wie viele Schokoladen und Süßigkeiten hast du insgesamt? (Hinweis: du kannst diese Rechnung mit der Python Konsole durchführen)

Übung 3

Mache eine Variable für deinen Vornamen und eine Variable für deinen Nachnamen. Erzeuge nun einen String und verwende Platzhalter um deinen Namen auszugeben.

3

Schildkröten und andere langsame Lebewesen

Es gibt bestimmte Gemeinsamkeiten zwischen einer lebendigen Schildkröte und einer Python Schildkröte. Im echten Leben sind Schildkröten (manchmal) grüne Reptilien, die sich langsam bewegen und ihr Haus auf dem Rücken tragen. In der Python Welt sind Schildkröten schwarze Pfeile die sich sehr langsam am Bildschirm bewegen. Ohne Haus.

Wenn man bedenkt, dass die Python Schildkröte eine Spur am Bildschirm hinterlässt, was für echte Schildkröten ungewöhnlich ist, müsste das Python Modul eher 'Nacktschnecke' heißen. Das klingt aber wenig attraktiv und so bleiben wir beim Namen Schildkröte (auf Englisch: turtle). Stell dir einfach vor, dass die Schildkröte ein paar Textmarker mitgenommen hat um ihren Weg zu markieren.

In der fernen Vergangenheit gab es einmal eine einfache Programmiersprache die Logo genannt wurde. Logo wurde verwendet um eine kleine Roboterschildkröte (mit dem Namen Irving) zu steuern. Die Zeit verging und aus dem Roboter, der sich am Boden bewegte wurde ein kleiner Pfleil am Monitor der sich bewegte.

Da kann man sehen, dass die Dinge mit der Zeit nicht immer besser werden. Eine kleine Roboterschildkröte wäre viel lustiger.

Das Python Schildkrötenmodul (wir kommen ein wenig später zu Modulen, aber fürs Erste reicht es zu wissen, dass man Module innerhalb von Programmen verwenden kann) ist der Programmiersprache Logo ähnlich. Aber während Logo recht eingeschränkt ist, hat Python sehr viele Möglichkeiten. Mit dem Schildkrötenmodul kann man aber zeigen, wie Computer Bilder auf dem Bildschirm zeichnen.

Fangen wir an. Zuerst sagen wir Python, dass wir dieses Schildkrötenmodul verwenden wollen und importieren es:

```
>>> import turtle
```

Dann brauchen wir eine Leinwand um darauf zu malen. So eine Leinwand wie sie Künstler verwenden. In unserem Fall ist es eine leere Fläche auf dem Bildschirm:

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

Mit diesem Code rufen wir die Funktion (Pen) vom turtle Modul auf. Es erzeugt automatisch die Leinwand zum drauf malen. Eine Funktion ist eine Stück Code, das immer wieder verwendet werden kann (dazu kommen wir später) und nützliche Dinge tut. In diesem Fall gibt die Pen Funktion ein Objekt zurück, das die Schildkröte darstellt. Wir verbinden unsere Variable 'schildkroete' mit diesem

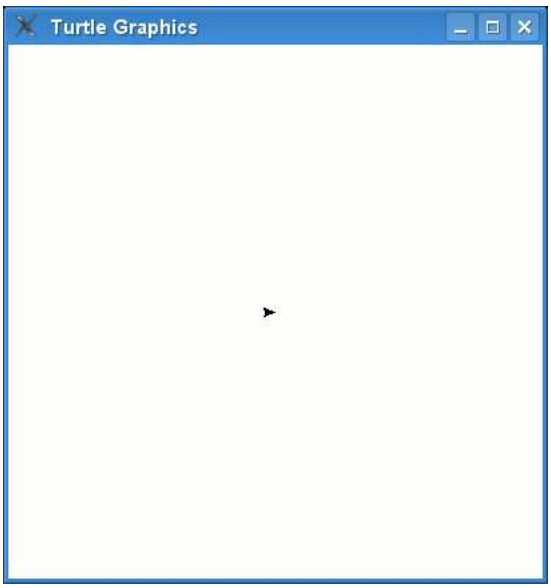


Abbildung 3.1: Ein Pfeil, der die Schildkröte darstellt.

Objekt (genau genommen benennen wir die Leinwand mit diesem Namen). Nachdem du diesen Code in die Python Konsole eingetippt hast, erscheint eine leere Box (die Leinwand) und schaut ungefähr aus wie Abbildung 3.1.

Ja, dieser kleine Pfeil in der Mitte des Bildschirmes ist wirklich die Schildkröte. Und ja, ich weiß, sie schaut nicht besonders sehr nach einer Schildkröte aus.

Jetzt kannst du Befehle an die Schildkröte schicken, indem zu Funktionen auf das Objekt anwendest, das beim turtle.Pen Aufruf zurückkam. Nachdem wir das Objekt 'schildkroete' genannt haben, können wir unsere Befehle an schildkroete schicken. Eine Funktion ist forward. Forward bedeutet vorwärts und sagt der Schildkröte sich nach vorne (wohin ihre Nase zeigt) zu bewegen. Lass uns der Schildkröte sagen, dass sie sich 50 Pixel nach vorne bewegen soll (Pixel werden gleich erklärt).

```
>>> schildkroete.forward(50)
```

Es sollte etwas wie in Abbildung 3.2 erscheinen.

Aus ihrer Sicht hat sich die Schildkröte 50 Schritte bewegt. Aus unserer Sicht 50 Pixel

Was ist also ein Pixel?

Ein Pixel ist ein Punkt auf dem Bildschirm. Wenn du ganz nah zum Monitor hingehst, erkennst du, dass alles aus winzigen rechteckigen Punkten besteht. Die Programme die du verwendest und Spiele die du auf Playstation, Xbox oder Wii spielst, bestehen auch aus einer Unzahl von bunten Punkten, die am Bildschirm angeordnet sind. Mit der Lupe wirst du vielleicht die einzelnen Punkte noch besser erkennen. Wenn wir die Python Leinwand mit der Schildkröte näher unter die Lupe nehmen, erkennen wir die einzelnen Punkte so wie in Abbildung 3.3.

In einem späteren Kapitel werden wir mehr über diese Punkte und Pixel erfahren.

Nun können wir der Schildkröte sagen dass sie links oder rechts abbiegen soll:

```
>>> schildkroete.left(90)
```

Damit sagst du der Schildkröte, dass sie links abbiegen soll (90 Grad). Du hast in der Schule vielleicht schon etwas über Winkelgrad gehört. Wenn nicht, könntest du dir das ungefähr wie das Ziffernblatt von einer Uhr vorstellen. Schau mal Abbildung 3.4 an.

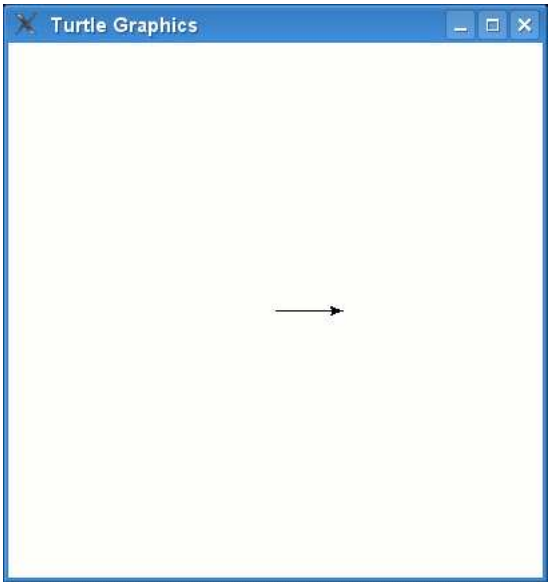


Abbildung 3.2: Die Schildkröte zeichnet eine Linie.

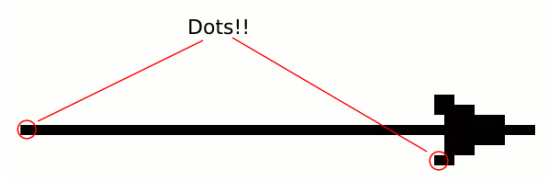


Abbildung 3.3: Die Linie und das Dreieck näher betrachtet.

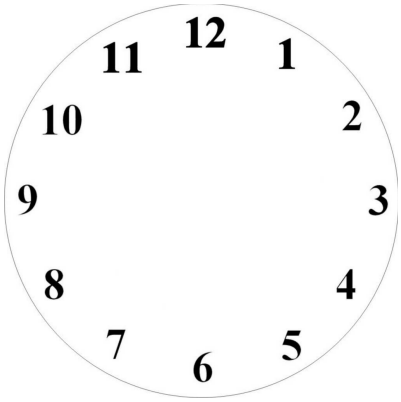


Abbildung 3.4: Die ‘Unterteilung’ der Uhr.

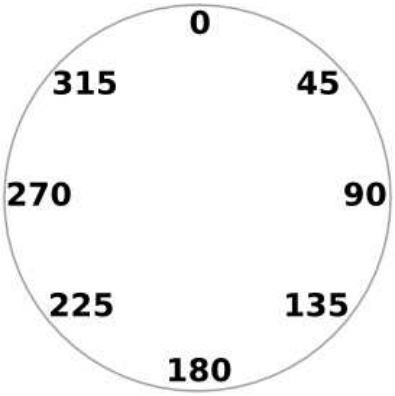


Abbildung 3.5: Unterteilung des Kreises in Grad.

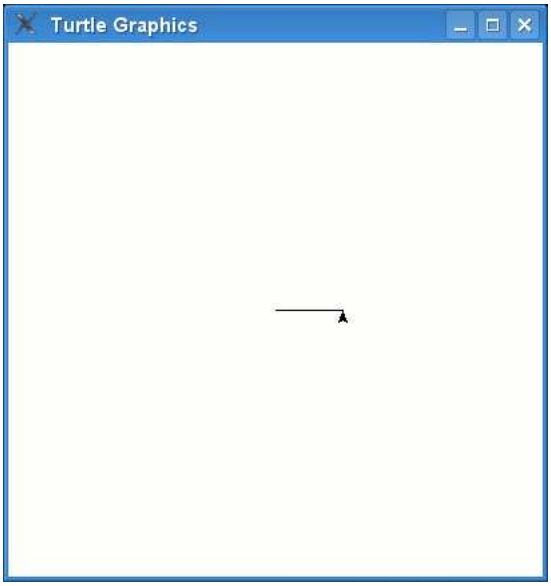


Abbildung 3.6: Die Schildkröte nachdem sie links abgebogen ist.

Im Unterschied zu einer Uhr, die 12 Unterteilungen hat (oder 60 wenn du die Minuten meinst), gibt es auch die Unterteilung in Grad. Dabei hat eine Umdrehung 360 Grad. Wo bei der Uhr die Unterteilung für 3 Uhr ist, würden es 90 Grad sein. Und bei 6 Uhr sind 180 Grad. In Abbildung 3.5 siehst du die Unterteilung in Grad.

Was heißt das also genau, wenn wir left(90) schreiben?

Wenn du aufrecht stehst und deinen Arm seitlich ausstreckst, dann sind das genau 90 Grad. Wenn du den linken Arm ausstreckst, sind das 90 Grad links und wenn du den rechten Arm ausstreckst, sind das 90 Grad rechts. Wenn die Python Schildkröte links abbiegt, dann dreht sie sich um 90 Grad links. Das ist das gleiche, wie wenn du deinen Körper so herumdrehst, dass du danach in die Richtung deiner ausgestreckten Hand schaust. Nachdem du also schildkroete.links(90) eingegeben hast, zeigt die Schildkröte nach oben wie du es in Abbildung 3.6 sehen kannst.

Probieren wir das noch einmal:

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

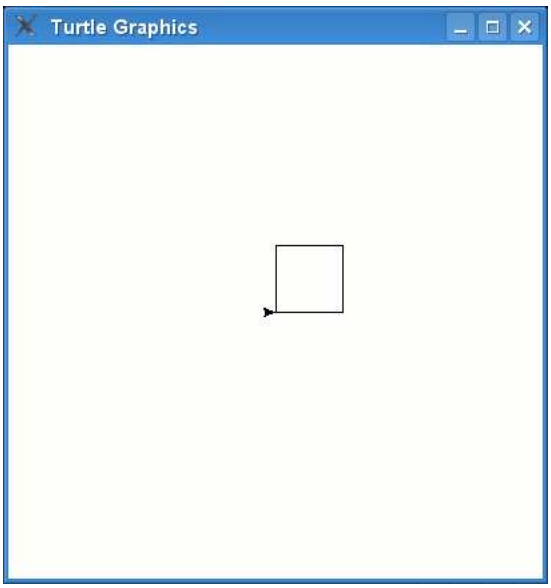


Abbildung 3.7: Ein Quadrat zeichnen.

```
>>> schildkroete.left(90)
```

Unsere Schildkröte hat ein Quadrat gezeichnet und schaut wieder in die selbe Richtung, wie sie gestartet ist (schau dir Abbildung 3.7 an).

Wir können die Leinwand auch löschen, indem wir den Befehl clear aufrufen:

```
>>> schildkroete.clear()
```

Einige andere Befehle, die die Schildkröte versteht sind: reset was auch die Leinwand löscht, aber die Schildkröte auch wieder am Start absetzt; mit backward geht die Schildkröte rückwärts; mit right biegt die Schildkröte rechts ab; und wenn sie keine Spur mehr zeichnen soll, dann gibt man up ein. Wenn die Schildkröte wieder zeichnen anfangen soll, dann gib down ein. Du kannst diese Befehle wie gewohnt verwenden:

```
>>> schildkroete.reset()
>>> schildkroete.backward(100)
>>> schildkroete.right(90)
>>> schildkroete.up()
>>> schildkroete.down()
```

Wir kommen später wieder auf das Schildkröten Modul zurück.

3.1 Probiere es aus

In diesem Kapitel haben wir gesehen, wie mit dem Turtle Modul einfache Linien gezeichnet werden. Wir haben auch Gradangaben verwendet, wenn die Schildkröte in eine andere Richtung zeigen soll.

Übung 1

Erzeuge eine Leinwand indem du die turtle Funktion Pen verwendest und zeichne ein Rechteck.

Übung 2

Erzeuge wieder eine Leinwand indem du die Funktion Pen aufrufst und zeichne ein Dreieck.

4 Stelle eine Frage

Beim Programmieren stellt man Fragen, um je nach Antwort das Eine oder Andere zu tun. Dabei spricht man von einer **if-Bedingung**. Zum Beispiel:

> Wie alt bist du? Wenn du älter als 20 bist, bist zu zu alt!

Das könnte man in Python folgendermaßen schreiben:

```
if alter > 20:
    print('du bist zu alt!')
```

Eine if-Bedingung besteht aus einem 'if' gefolgt von der sogenannten 'Bedingung', gefolgt von einem Doppelpunkt (:). Die folgenden Zeilen müssen in einem Block sein—und wenn die Antwort auf die Frage 'ja' (heißt in der Programmiersprache true oder wahr) ist, wird der Block ausgeführt.

Eine Bedingung ist eine Konstruktion, die entweder 'ja' (true, wahr) oder 'no' (false, falsch) zurückgibt. Es gibt bestimmte Symbole (oder auch Operatoren) die für solche Bedingungen verwendet werden. Zum Beispiel:

==	ist gleich
!=	ungleich
>	größer als
<	kleiner als
>=	größer als oder gleich
<=	kleiner als oder gleich

Wenn du zum Beispiel 10 Jahre alt bist, dann würde die Bedingung dein_alter == 10 zum Beispiel wahr (true) zurückgeben, wenn du aber nicht 10 bist, dann würde falsch (false) zurückkommen. Pass auf dass du die **zwei** Gleichheitszeichen (==) nicht mit dem normalen einfachen Gleichheitszeichen (=) verwechselt. Wenn du nur ein = Zeichen in einer Bedingung verwendest, dann bekommst du einen Fehler.

Nehmen wir an, dass du die Variable alter auf dein Alter setzt und du 12 Jahre bist. Dann würde die Bedingung . . .

```
alter > 10
```

. . . Wahr (true) zurückgeben. Wenn du 8 Jahre wärst, dann würde Falsch (false) zurückkommen. Wenn du genau 10 Jahre wärest, würde auch Falsch rauskommen—weil die Bedingung ja fragt ob du älter als (>) 10 und nicht ob du älter oder auch gleichalt (>=) als 10 bist.

Probieren wir es aus:

```
>>> alter = 10
>>> if alter > 10:
...     print('dann kommst du hier an')
```

Was passiert, wenn du da obige in die Konsole eintippst?
Nichts.
Weil der Wert der Variable alter nicht größer als 10 ist, wird der Block mit dem print Befehl nie ausgeführt werden. Probiers mal so:

```
>>> alter = 10
>>> if alter >= 10:
...     print('dann kommst du hier an')
...
```

Wenn du das Beispiel von oben nun ausprobierst, sollte die Nachricht ‘dann kommmst du hier an’ auf der Konsole erscheinen. (Nach der Zeile mit dem print Befehl zwei mal die Eingabetaste drücken kann helfen). Das gleiche wird auch beim nächsten Beispiel passieren:

```
>>> age = 10
>>> if age == 10:
...     print('dann kommst du hier an')
...
dann kommst du hier an
```

4.1 Tu dies. . . oder das!!!

Wir können die if-Bedingung auch erweitern, so dass auch etwas passiert, wenn die Bedingung falsch ist. Zum Beispiel das Wort ‘Hallo’ ausgeben, wenn du 12 Jahre bist und ‘Auf Wiedersehen’ wenn nicht. Das geht mit einer if-then-else-Bedingung (das ist nur eine andere Art um folgendes auszudrücken *“wenn etwas wahr ist tue **das Eine**, wenn es falsch ist tue **das Andere**”*):

```
>>> alter = 12
>>> if alter == 12:
...     print('Hallo')
... else:
...     print('Auf Wiedersehen')
...
Hallo
```

Nachdem du das Beispiel eingetippt hast, sollte die Ausgabe ‘Hallo’ auf der Konsole erscheinen. Ändere den Wert der Variable alter auf eine andere Zahl und es wird ‘Auf Wiedersehen’ ausgegeben.

```
>>> alter = 8
>>> if alter == 12:
...     print('Hallo')
... else:
...     print('Auf Wiedersehen')
...
Auf Wiedersehen
```

4.2 Tu das. . . oder dies. . . oder jenes. . . oder!!!

Wir können die if-Bedingung sogar weiter aufspalten, indem wir elif (Kurzform für else-if) verwenden. Wir könnten testen, ob dein Alter 10 ist, oder 11, oder 12 und so weiter:

```
 1. >>> alter = 12
 2. >>> if alter == 10:
 3. ...     print('Du bist 10')
 4. ... elif alter == 11:
 5. ...     print('Du bist 11')
 6. ... elif alter == 12:
 7. ...     print('Du bist 12')
 8. ... elif alter == 13:
 9. ...     print('Du bist 13')
10. ... else:
11. ...     print('Hä?')
12. ...
13. Du bist 12
```

Im obigen Code überprüft die zweite Zeile, ob der Wert der Variable 10 entspricht. Wenn nicht springt Python zu Zeile 4 um zu überprüfen, ob der Wert der alter Variable 11 entspricht. Wenn es wieder nicht so ist, springt Python zu Zeile 6. Dort wird geschaut, ob die Variable den Wert 12 hat. In diesem Fall ist es so und deswegen kommt danach die Zeile 7 dran und der print Befehl wird ausgeführt. Wenn du den Code anschaust, erkennst du 5 Gruppen—Zeile 3, 5, 7, 9 und 11.

4.3 Bedingungen kombinieren

Du kannst auch Bedingungen kombinieren in dem du die Schlüsselworte ‘and’ und ‘or’ verwendest. Das obige Beispiel könnte mit der Verwendung von ‘or’ kürzer geschrieben werden.

```
1. >>> if alter == 10 or alter == 11 or alter == 12 or alter == 13:
2. ...     print('Du bist %s' % alter)
3. ... else:
4. ...     print('Hä?')
```

Wenn eine Bedingung der ersten Zeile wahr ist (z.B.: wenn die Variable alter 10 **oder** 11 **oder** 12 **oder** 13 ist, dann springt Python zu Zeile 2, ansonsten zu Zeile 4. Noch kompakter geschrieben geht es mit der Hilfe der ‘and’, >= und <= Symbole:

```
1. >>> if alter >= 10 and alter <= 13:
2. ...     print('Du bist %s' % alter)
3. ... else:
4. ...     print('Häh?')
```

Hoffentlich erkennst du, dass wenn **beide** Bedingungen auf der ersten Zeile erfüllt sind, der Block auf der zweiten Zeile ausgeführt wird (wenn die Variable alter größer gleich 10 oder kleiner gleich 13 ist). Wenn der Wert der alter Variable also 12 ist, würde 'Du bist 12' ausgegeben werden: weil 12 größer als 10 und auch kleiner als 13 ist.

4.4 Nichts

Da gibt es noch einen Datentyp, den man Variablen zuordnen kann. Nämlich der Typ **nichts**.

Variablen können Nummern, Strings und Listen sein. Aber man kann auch 'nichts' einer Variable zuweisen. In Python bezeichnet man das als None (in anderen Programmiersprachen spricht man oft von null). Das Zuweisen funktioniert wie bei anderen Typen:

```
>>> meine_variable = None
>>> print(meine_variable)
None
```

Die Verwendung von None ist eine Möglichkeit eine Variable zurückzusetzen oder eine Variable zu erzeugen ohne ihr gleich einen Wert zu geben.

Machen wir ein kleines Beispiel. Dein Fußballteam sammelt Geld für neue Uniformen. Damit du weißt, wieviel Geld zusammengekommen ist, musst du warten, bis jeder einzelne zurückgekommen ist, damit du die Summen addieren kannst. Wenn du das programmieren würdest, könntest du für jedes Teammitglied eine Variable verwenden und die Variable am Anfang auf None setzen.

```
>>> spieler1 = None
>>> spieler2 = None
>>> spieler3 = None
```

Mit einer if-Bedingung könnten wir diese Variablen überprüfen um zu sehen, ob die Teammitglieder denn schon alle vom Geld sammeln zurückgekommen sind.

```
>>> if spieler1 is None or spieler2 is None or spieler3 is None:
...     print('Warte bitte bis alle Spieler zurückgekommen sind')
... else:
...     print('Ihr habt %s gesammelt' % (spieler1 + spieler2 + spieler3))
```

Die if-Bedingung überprüft ob eine der Variablen noch den Wert None hat und spuckt die erste Nachricht aus, wenn ja. Wenn jede Variable einen Wert hat, dann wird die zweite Nachricht ausgegeben und die gesammelte Geldsumme ausgegeben. Wenn du alle Variablen auf None setzt, wirst du die erste Nachricht sehen (erzeuge die Variablen aber vorher, ansonsten gibt es eine Fehlermeldung):

```
>>> if spieler1 is None or spieler2 is None or spieler3 is None:
...     print('Warte bitte bis alle Spieler zurückgekommen sind')
... else:
```

```
...     print('Ihr habt %s gesammelt' % (spieler1 + spieler2 + spieler3))
...
Warte bitte bis alle Spieler zurückgekommen sind
```

Auch wenn du 2 Variablen auf bestimmte Werte setzt, wirst du folgende Nachricht bekommen:

```
>>> spieler1 = 100
>>> spieler3 = 300
>>> if spieler1 is None or spieler2 is None or spieler3 is None:
...     print('Warte bitte bis alle Spieler zurückgekommen sind')
... else:
...     print('Ihr habt %s gesammelt' % (spieler1 + spieler2 + spieler3))
...
Warte bitte bis alle Spieler zurückgekommen sind
```

Erst wenn alle Variablen Werte zugewiesen bekommen haben, kommt die Nachricht vom zweiten Block:

```
>>> spieler1 = 100
>>> spieler2 = 300
>>> spieler3 = 500
>>> if spieler1 is None or spieler2 is None or spieler3 is None:
...     print('Warte bitte bis alle Spieler zurückgekommen sind')
... else:
...     print('Ihr habt %s gesammelt' % (spieler1 + spieler2 + spieler3))
...
Ihr habt 900 gesammelt
```

4.5 Was ist der Unterschied. . .?

Was ist der Unterschied zwischen 10 und ’10’?

Abgesehen von den Anführungsstrichen nicht viel könntest du denken. Aber nachdem du die vorigen Kapitel gelesen hast, weißt du das 10 eine Zahl und ’10’ ein String ist. Das macht sie verschiedener als du vielleicht denkst. Früher haben wir die Werte von einer Variable (alter) bei einer if-Bedingung mit einer Zahl verglichen.

```
>>> if alter == 10:
...     print('du bist 10')
```

Wenn du die Variable auf 10 setzt, dann wird die print Funktion ausgeführt:

```
>>> alter = 10
>>> if alter == 10:
...     print('du bist 10')
...
du bist 10
```

Aber wenn die Variable alter den Wert ’10’ (achte auf die Anführungsstriche), dann wird die print Funktion nicht ausgeführt:

```
>>> alter = '10'
>>> if alter == 10:
...     print('du bist 10')
...
```

Warum wird der Code im Block (der print Block) nicht ausgeführt? Weil ein String sich von einer Zahl unterscheidet, auch wenn sie gleich aussehen:

```
>>> alter1 = 10
>>> alter2 = '10'
>>> print(alter1)
10
>>> print(alter2)
10
```

Siehst du! Sie schauen genau gleich aus. Aber weil die eine Variable ein String ist und die andere Variable eine Nummer, sind es verschiedene Werte. Deswegen wird alter == 10 (alter ist gleich 10) bei einem String nie wahr sein.

Der wahrscheinlich beste Weg das zu verstehen ist an 10 Bücher und 10 Ziegelsteine zu denken. Die Anzahl ist die gleiche, aber könntest du sagen, dass 10 Bücher das gleiche wie 10 Ziegelsteine sind? Zum Glück hat Python einige magische Funktionen, die Strings in Nummern und Nummern in Strings verwandeln können (auch wenn dadurch Bücher nicht zu Ziegelsteinen werden). Wenn du zum Beispiel den string '10' in eine Nummer wandeln willst, dann könntest du die Funktion int verwenden:

```
>>> alter = '10'
>>> verwandeltes_alter = int(alter)
```

Die Variable verwandeltes_alter hat jetzt die Nummer 10 und keinen String. Um eine Nummer in einen String zu wandeln, nutze die Funktion str:

```
>>> alter = 10
>>> verwandeltes_alter = str(alter)
```

verwandeltes_alter hat jetzt den String 10 und keine Nummer. Nun zurück zur if-Bedingung, die nichts ausgegeben hat:

```
>>> alter = '10'
>>> if alter == 10:
...     print('du bist 10')
...
```

Wenn wir nun die Variable zu einer Zahl machen, *bevor* wir die Bedingung überprüfen, bekommen wir ein anderes Resultat:

```
>>> alter = '10'
>>> verwandeltes_alter = int(alter)
>>> if verwandeltes_alter == 10:
...     print('du bist 10')
...
du bist 10
```

5 Immer wieder

Es gibt nichts Schlimmeres als immer wieder die gleiche Arbeit tun zu müssen. Das ist der Grund, warum dir deine Eltern sagen, du solltest Schafe zählen, wenn du einschlafen willst. Es hat nichts mit einer geheimnisvollen schläfrigmachenden Eigenschaft dieser wolligen Säugetiere zu tun, sondern das dein Gehirn das endlose Wiederholen langweilig findet und deswegen leichter einschläft.

Programmierer mögen sich auch nicht gerne wiederholen. Davon werden sie auch sehr müde. Darum haben alle Programmiersprachen eine sogenannte **for-Schleife**. Wenn du zum Beispiel fünf mal 'hallo' ausgeben willst, *könntest* du folgendes schreiben:

```
>>> print("hallo")
hallo
>>> print("hallo")
hallo
>>> print("hallo")
hallo
>>> print("hallo")
hallo
>>> print("hallo")
hallo
```

Und das ist... ziemlich mühsam.

Oder du verwendest die for-Schleife (beachte die 4 Leerzeichen auf der zweiten Zeile—anstatt der Leerzeichen schreibe ich dir das @-Symbol auf, damit du siehst, wo die Leerzeichen sind. Beim Abtippen lässt du wieder die @ weg):

```
>>> for x in range(0, 5):
... @@@@print('hallo')
...
hallo
hallo
hallo
hallo
hallo
```

Mit der range Funktion kannst du schnell und einfach Zahlenlisten erzeugen, die von einer Startnummer anfangen und bei einer Endnummer aufhören. Zum Beispiel:

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

Bei unserem Beispiel mit der for-Schleife 'for x in range(0, 5)' erzeugt Python eigentlich eine Liste von Zahlen (0, 1, 2, 3, 4) und speichert die Werte in der Variable x. Wir können die Variable x auch in unserer print Funktion verwenden:

```
>>> for x in range(0, 5):
...     print('hallo %s' % x)
hallo 0
hallo 1
hallo 2
hallo 3
hallo 4
```

Ohne die for-Schleife würde es ungefähr so aussehen:

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

Die Schleife hat uns also das Tippen von 8 Zeilen erspart. Das ist extrem nützlich, weil der durchschnittliche Programmierer fauler als ein Nilpferd an einem heißen Sommertag ist, wenns ums tippen geht. Gute Programmierer hassen Dinge mehr als einmal zu tippen, also ist die for-Schleife eine sehr nützliche Konstruktion.

ACHTUNG!!!

Wenn du das Code Beispiel gerade selber ausprobiert hast, hast du vielleicht eine komische Fehlermeldung bekommen. Vielleicht etwas wie:

IndentationError: expected an indented block

Wenn du diesen Fehler bekommst, hast du die Leerzeichen auf der zweiten Zeile vergessen. Leerzeichen (normale Leerzeichen oder ein Tabulator) sind in Python extrem wichtig. Das behandeln wir etwas später genauer.

Wir müssen auch nicht unbedingt range verwenden, sondern könnnten auch bereits erstellte Listen aus dem zweiten Kapitel nutzen:

```
>>> Einkaufsliste = ['Eier', 'Milch', 'Käse', 'Sellerie', 'Honig']
>>> for i in Einkaufsliste:
```

```
...     print(i)
...
Eier
Milch
Käse
Sellerie
Honig
```

Mit diesem Code-Beispiel sagen wir Python "speichere jeden Listeneintrag in der Variable 'i' und gib den Inhalt der Variable danach aus". Ohne die for-Schleife müssten wir es so schreiben:

```
>>> Einkaufsliste = ['Eier', 'Milch', 'Käse', 'Sellerie', 'Honig', 'Backpulver']
>>> print(Einkaufsliste[0])
Eier
>>> print(Einkaufsliste[1])
Milch
>>> print(Einkaufsliste[2])
Käse
>>> print(Einkaufsliste[3])
Sellerie
>>> print(Einkaufsliste[4])
Honig
```

Die for-Schleife hat uns wieder eine Menge Tipparbeit erspart.

5.1 Wann ist ein Block nicht quadratisch?

Wenn es ein Codeblock ist.
Was ist ein 'Codeblock' dann genau?

Ein Codeblock ist eine Gruppierung von Anweisungen. Bei einem for-loop könntest du mehr als nur den Listeneintrag ausgeben wollen. Vielleicht willst du den Eintrag kaufen und dann ausgeben, was das genau war. Nehmen wir an, dass es die Funktion 'kaufen' gibt, dann könnten wir das so schreiben:

```
>>> for i in Einkaufsliste:
...     kaufen(i)
...     print(i)
```

Du brauchst das Beispiel jetzt nicht in der Python Konsole ausprobieren—weil wir die kaufen Funktion noch nicht haben und du einen Fehler zurückbekommen würdest—aber es zeigt dir, wie ein Python Block mit 2 Anweisungen ausschaut:

```
kaufen(i)
print(i)
```

Leerzeichen sind in Python sehr wichtig. Leerzeichen kannst du einfügen, indem zu die Leertaste oder die Tabulator-Taste drückst. Code, der an der gleichen Position gruppiert wurde, ergibt einen Codeblock.

```
    das wäre Block 1
    das wäre Block 1
    das wäre Block 1
        das wäre Block 2
        das wäre Block 2
        das wäre Block 2
    das wäre immer noch Block 1
    das wäre immer noch Block 1
        das wäre Block 3
        das wäre Block 3
            das wäre Block 4
            das wäre Block 4
            das wäre Block 4
```

Du musst aber die Leerzeichen einheitlich verwenden. Zum Beispiel:

```
>>> for i in Einkaufsliste:
...     kaufen(i)
...       print(i)
```

Die zweite Zeile (Funktion kaufen(i)) fängt mit **4** Leerzeichen an. Die dritte Zeile (print(i)) fängt mit **6** Leerzeichen an. Schauen wir uns das ganze nochmals an, indem die Leerzeichen durch sichtbare @-Zeichen ersetzt werden.

```
>>> for i in Einkaufsliste:
... @@@@kaufen(i)
... @@@@@@print(i)
```

Das würde einen Fehler verursachen. Wenn du einmal mit 4 Leerzeichen angefangen hast, dann musst du auch mit 4 Leerzeichen weitermachen. Und wenn du einen Block in einen anderer Block *verschachteln* willst, dann brauchst du 8 Leerzeichen (2 x 4) am Anfang des inneren Blocks.

Der erste Block hat also 4 Leerzeichen (ich markiere sie wieder, damit du sie sehen kannst):

```
@@@@das ist mein erster Block
@@@@das ist mein erster Block
```

Der zweite Block (der 'innerhalb' des Ersten ist) braucht 8 Leerzeichen:

```
@@@@das ist mein erster Block
@@@@das ist mein erster Block
@@@@@@@@das ist mein zweiter Block
@@@@@@@@das ist mein zweiter Block
```

Warum werden Blöcke 'ineinander' verschachtelt? Normalerweise dann, wenn der zweite Block vom ersten Block abhängt. Wenn die for-Schleife der erste Block ist, dann sind die Anweisungen, die immer wieder ausgeführt werden, der zweite Block—somit braucht der zweite Block den ersten Block um richtig zu funktionieren.

Wenn du in der Python Konsole Code in einem Block schreibst, bleibt Python solange in diesem Block, bis du die Enter Taste zwei mal drückst (solange du im Block bist, siehst du 3 Punkte am Anfang der Zeile).

Lass uns ein echtes Beispiel probieren. Öffne die Konsole und tippe folgendes ein (und drücke am Anfang der print Zeile 4 mal die Leertaste):

```
>>> meine_liste = [ 'a', 'b', 'c' ]
>>> for i in meine_liste:
...     print(i)
...     print(i)
...
a
a
b
b
c
c
```

Drücke nach dem zweiten print Befehl zwei mal die Enter Taste—damit sagst du der Python Konsole, dass du den Block beendest. Danach wird jedes Element der Liste zwei mal ausgegeben. Das folgende Beispiel wird nicht funktionieren und ein Fehler wird angezeigt werden:

```
>>> meine_liste = [ 'a', 'b', 'c' ]
>>> for i in meine_liste:
...     print(i)
...       print(i)
...
  File "<stdin>", line 3
    print(i)
    ^
IndentationError: unexpected indent
```

Die zweite print Zeile hat 6 Leerzeichen und nicht 4, was Python nicht mag, weil die Abstände immer gleich bleiben müssen.

MERKE DIR

Wenn du deine Blöcke mit 4 Leerzeichen beginnst, musst du mit 4 Leerzeichen weitermachen. Wenn du mit 2 Leerzeichen beginnst, mache mit 2 Leerzeichen weiter. Am besten aber du hältst dich an 4 Leerzeichen, weil das die meisten Leute so machen.

Hier ist ein etwas komplizierteres Beispiel mit 2 Codeblöcken:

```
>>> meine_liste = [ 'a', 'b', 'c' ]
>>> for i in meine_liste:
...     print(i)
...     for j in meine_liste:
...         print(j)
...
```

Wo sind in diesem Code die Blöcke und was werden sie machen...?
Hier gibt es **zwei** Blöcke—der erste Block gehört zur ersten for-Schleife:

```
>>> meine_liste = [ 'a', 'b', 'c' ]
>>> for i in meine_liste:
...     print(i)                #
...     for j in meine_liste    #-- diese Zeilen sind der ERSTE Block
...         print(j)            #
...
```

Der zweite Block ist die print Zeile der zweiten for-Schleife:

```
>>> meine_liste = [ 'a', 'b', 'c' ]
>>> for i in meine_liste:
...     print(i)
...     for j in meine_liste
...         print(j)            # diese Zeile ist der ZWEITE Block
...
```

Kannst du vorhersagen, was dieses kleine Codebeispiel machen wird?

Es werden die drei Buchstaben von 'meine_liste' ausgeben. Aber wie oft? Wenn du dir die Zeilen genau anschaust, kannst du es vielleicht sehen. Die for-Schleife wird durch alle Elemente der Liste gehen und dann die Kommandos aus Block 1 ausführen. Nachdem also der erste Buchstabe ausgegeben wird, kommt die nächste for-Schleife dran. In dieser Schleife geht Python wieder alle Elemente durch und führt den Befehl im zweiten Block aus. Also sollten wir den Buchstaben 'a' bekommen, gefolgt von 'a', 'b', 'c' dann 'b' gefolgt von 'a', 'b', 'c' und so weiter. Gib den Code in die Python Konsole ein und überzeuge dich selbst:

```
>>> meine_liste = [ 'a', 'b', 'c' ]
>>> for i in meine_liste:
...     print(i)
...     for j in meine_liste:
...         print(j)
...
a
a
b
c
b
a
b
c
c
a
b
c
```

Wie wäre es mit etwas Anderem als nur Buchstaben auszugeben? Erinnerst du dich noch an die Rechnung am Anfang des Buches als wir berechnet haben, wieviel du in einem Jahr verdienen

kannst, wenn du 5 € für die Hausarbeit, 30 € fürs Zeitung austragen bekommst und 10 € pro Woche ausgibst?
Das hat so ausgesehen:

```
>>> (5 + 30 - 10) * 52
```

(Das sind 5 € + 30 € - 10 € multipliziert mit 52 Wochen im Jahr).

Es könnte praktisch sein zu sehen, wie dein Erspartes über das Jahr hinaus ansteigt. Das machen wir mit einer for-Schleife. Zuerst aber laden wir die wichtigen Zahlen in Variablen.

```
>>> hausarbeit = 5
>>> zeitung = 30
>>> ausgaben = 10
```

Die Rechnung können wir nun auch mit den Variablen durchführen:

```
>>> (hausarbeit + zeitung - ausgaben) * 52
1300
```

Oder wir lassen uns für jede Woche im Jahr den zusammengesparten Betrag ausrechnen, indem wir die neue Variable ‘erspartes’ in einer for-Schleife verwenden:

```
1. >>> erspartes = 0
2. >>> for woche in range(1, 53):
3. ...     erspartes = erspartes + hausarbeit + zeitung - ausgaben
4. ...     print('%s Woche = %s' % (woche, erspartes))
5. ...
```

In der ersten Zeile setzen wir die Variable ‘erspartes’ auf 0 (am Anfang des Jahres haben wir noch nichts gespart).
Auf Linie 2 beginnen wir die for-Schleife, die den Block auf Zeile 3 und 4 immer wieder ausführen wird. Bei jedem Durchlauf des Blocks hat die Variable woche eine höhere Zahl.
Zeile 3 musst du dir vielleich ein wenig länger anschauen. Denke dir dabei vielleicht die Variable ‘erspartes’ wie eine Sparbüchse. Jede Woche gibtst du das Geld von der Hausarbeit und dem Zeitung austragen hinein. Gleichzeitig nimmst du aber auch 10 € heraus. Der Computer rechnet den Teil rechts vom Gleichheitszeichen (=) zuerst aus und speichert das Ergebnis zur Variable auf der linken Seite.
In Zeile 4 ist ein print Befehl, der gleichzeitig die Woche sowie den gesammelten Geldbetrag ausgibt. Schau nochmals auf Seite 14 nach, wenn es dir nicht gleich klar wird. Nachdem du alle Zeilen eingegeben hast, wird folgendes ausgegeben. . .

```
1 Woche = 25
2 Woche = 50
3 Woche = 75
4 Woche = 100
5 Woche = 125
6 Woche = 150
7 Woche = 175
8 Woche = 200
```

```
9 Woche = 225
10 Woche = 250
11 Woche = 275
12 Woche = 300
13 Woche = 325
14 Woche = 350
15 Woche = 375
```

. . .bis zur 52. Woche.

5.2 Wenn wir schon von Schleifen reden. . .

Die for-Schleife ist nicht die einzige Art von Schleifen, die du in Python verwenden kannst. Da gibt es auch die while-Schleife. Bei der for-Schleife, weißt du schon im vornhinein, wann die Schleife zu Ende geht. Bei einer while-Schleife weißt du das nicht. Stell dir eine Stiege mit 20 Treppen vor. Du kannst 20 Stufen einfach hochsteigen. Das ist eine for-Schleife.

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

Stelle dir jetzt eine Treppe vor, die auf einen Berg hinaufführt. Du könntest beim hochsteigen müde werden. Oder es kommt ein Wetterumbruch und du drehst um. Das wäre dann eine while-Schleife.

```
>>> stufe = 0
>>> while stufe < 10000:
...     print(stufe)
...     if muede == True:
...         break
...     elif schlechtes_wetter == True:
...         break
...     else:
...         step = step + 1
```

Den obigen Code kannst du noch nicht ausprobieren solange die Variablen muede oder schlechtes_wetter nicht existieren. Aber es zeigt dir die Grundlage einer while-Schleife. Solange die Variable stufe kleiner als 10000 ist, wird der Inhalt des Blocks ausgeführt. Im Block selber wird der Wert der Variable stufe ausgegeben, danach wird überprüft, ob die Variable muede oder schlechtes_wetter wahr ist. Wenn muede wahr ist, dann verlässt Python den Block und macht außerhalb weiter. Wenn schlechtes_wetter wahr ist genauso. In entgegengesetzten Fall wird **1** zur Variable stufe dazugezählt und die Bedingung am Anfang der Schleife wird wieder überprüft.
Die Schritte einer while-Schleife sind zusammengefasst:

- ▷ überprüfe die Bedingung,
- ▷ führe den Block aus,
- ▷ wiederhole von vorne

Sehr häufig wird eine while-Schleife mit mehreren Bedingungen verwendet:

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

In dieser Schleife erzeugen wir die Variable x mit dem Wert 45 und die Variable y mit dem Wert 80. Es sind zwei Bedingungen, die in der Schleife überprüft werden: ob x kleiner als 50 und ob y kleiner als 100 ist. Wenn beide Bedingungen wahr sind, dann wird der Codeblock ausgeführt, der die Zahl 1 zu den Variablen hinzufügt und die Variablen ausgibt. Die Ausgabe des Codes wird folgende sein:

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

Vielleicht kannst du herausfinden, warum diese Nummern ausgegeben werden?[1]

Gerne wird die while-Schleife auch für halb-unendliche Schleifen verwendet. Das ist eine Schleife, die eigentlich unendlich läuft, bis irgendwas im Code passiert. Zum Beispiel:

```
>>> while True:
...     hier steht eine Menge Code
...     hier steht eine Menge Code
...     hier steht eine Menge Code
...     if eine_bedingung == True:
...         break
```

Die Bedingung für die while-Schleife ist 'True' (also Wahr). Deswegen wird der nachfolgende Block auch immer ausgeführt (dies erzeugt eine sogenannte Endlosschleife). Erst wenn die Bedingung eine_bedingung True (Wahr) wird, verlässt Python die Schleife. Ein schönes Beispiel kannst du im Anhang C sehen (im Abschnitt über das random Modul), aber vielleicht liest du noch das nächste Kapitel, bevor zu im Anhang nachblätterst.

5.3 Probiere es aus

In diesem Kapitel haben wir gesehen wie man Schleifen für sich wiederholende Aufgaben verwendet. Wir haben Code Blöcke innerhalb von Schleifen verwendet um die Anweisungen zu Gruppieren, die wiederholt werden sollen.

[1] Wir beginnen mit x bei 45 zu zählen und y fängt bei 80 an. Bei jedem Schleifendurchlauf werden die Zahlen um 1 erhöht. Die Bedingung am Anfang schaut darauf, dass x kleiner als 50 und y kleiner als 100 ist. Nach 5 Durchläufen erreicht die Variable x den Wert 50. Nun ist die erste Bedingung ($x < 50$) nicht mehr wahr und Python verlässt die Schleife.

Übung 1

Was wird der folgende Code machen?

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

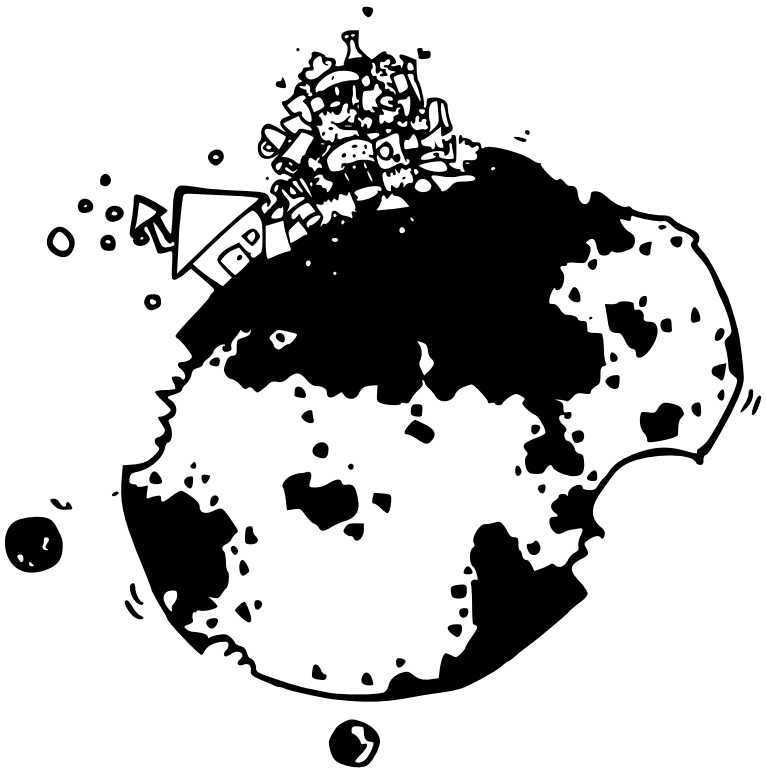
Übung 2

Wenn du Geld in die Bank bringst, bekommst du Zinsen. 'Zinsen' ist der Geldbetrag, den du von der Bank bekommst, damit die Bank dein Geld benutzen kann—jedes Jahr bekommst du so einen kleinen Betrag auf dein Sparbuch zum anderen Geld. Vielleicht fragst du deine Eltern wie das alles funktioniert. Jedenfalls berechnet man die Zinsen mit Prozent. Wenn du nicht weißt, was Prozent sind, macht das nichts. Du musst dir nur merken, wenn du 1% (1 Prozent) Zinsen bekommst, kannst du die Geldsumme mit 0.01 multiplizieren um die Zinsen zu berechnen (bei 1000 € bekommst du also 1000 x 0.01 = 10 €). Bei zwei Prozent Zinsen multiplizierst du mit 0.02 und so weiter. Wenn du nun 100 € gespart hast, und du bekommst 3% Zinsen pro Jahr, wie viel Geld wirst du am Ende der nächsten 10 Jahre haben? Du kannst das mit einer for-Schleife berechnen (Tipp: zähle die Zinsen am Ende des Jahres zur Geldsumme dazu).

6 Wie Recycling. . .

Denke einmal darüber nach, wie viel Müll du jeden Tag erzeugst. Getränkeflaschen, Chipstüten, Sandwichpapier, Fleischtassen, Einkaufstaschen, Zeitungen, Magazine und so weiter.

Stell dir einmal kurz vor wie das wäre, wenn der ganze Müll am Ende deiner Straße auf einen großen Haufen landen würde.



Vielleicht trennst du jetzt schon den Müll und hilfst so bei der Wiederverwertung. Das ist auch ein Glück, denn niemand will gerne auf dem Weg in die Schule über einen Müllberg klettern. Mit dem Recycling werden aus alten Glasflaschen wieder neue. Aus Plastik neuer Kunststoff und aus Altpapier werden wieder neue Zeitungen.

Recycling und Wiederverwerten sind in der Welt der Programmierung genauso interessant. Nicht weil dein Programm unter einem Müllberg verschwinden würde—sondern damit du deine Fingerkuppen schonst, und nicht alles immer wieder eintippen musst.

Es gibt verschiedene Möglichkeiten in Python (und anderen Programmiersprachen), wie man Code wieder verwendet, und eine Art haben wir schon in Kapitel 3 mit der range-Funktion kennen-

gelernt. Funktionen sind also eine Art der Wiederverwendung—du schreibst den Code einmal und verwendest ihn in deinem Programmen immer wieder. Probier ein einfaches Beispiel:

```
>>> def meine_funktion(mein_name):
...     print('Hallo %s' % mein_name)
...
```

Diese Funktion hat den Namen 'meine_funktion' und den Parameter 'mein_name'. Ein *Parameter* ist eine Variable, die nur innerhalb eines Blockes eine Bedeutung hat (das ist der Block, der sofort nach der def-Zeile beginnt). def ist die Kurzform von define, und mit diesem Schlüsselwort definierst du Funktionen. Die Funktion kannst du ausprobieren, indem du sie aufrufst und in runden Klammern einen Parameter mitgibst.

```
>>> meine_funktion('Hannah')
Hallo Hannah
```

Jetzt ändern wir die Funktion, damit 2 Parameter übergeben werden können:

```
>>> def meine_funktion(vorname, nachname):
...     print('Hallo %s %s' % (vorname, nachname))
...
```

Und Aufrufen funktioniert ganz ähnlich:

```
>>> meine_funktion('Hannah', 'Schmidt')
Hallo Hanna Schmidt
```

Oder wir erzeugen ein Paar Variablen und rufen die Funktion mit den Variablen auf:

```
>>> vn = 'Tom'
>>> nn = 'Turbo'
>>> meine_funktion(vn, nn)
Hallo Tom Turbo
```

Und eine Funktion kann auch Werte zurückgeben, indem das return-Schlüsselwort verwendet wird:

```
>>> def ersparnisse(hausarbeit, zeitung, ausgaben):
...     return hausarbeit + zeitung - ausgaben
...
>>> print(ersparnisse(10, 10, 5))
15
```

Diese Funktion nimmt 3 Parameter und addiert die ersten Zwei (hausarbeit und zeitung), bevor es dann den letzten Parameter (ausgaben) abzieht. Das Ergebnis wird zurückgegeben—und kann wie der Wert einer Variable verwendet werden:

```
>>> meine_ersparnisse = ersparnisse(20, 10, 5)
>>> print(meine_ersparnisse)
25
```

Variablen, die innerhalb der Funktion verwendet werden, können aber von außen nicht aufgerufen werden.

```
>>> def variable_test():
...     a = 10
...     b = 20
...     return a * b
...
>>> print(variable_test())
200
>>> print(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Im obigen Beispiel erzeugen wir eine Funktion variable_test, in der zwei Variablen miteinander multipliziert werden (a und b) und das Ergebnis wird zurückgegeben. Wenn wir die Funktion mit der print Funktion aufrufen, erhalten wir 200. Wenn wir aber direkt auf die Variable a oder b zugreifen, erscheint die Fehlermeldung, dass "'a' nicht definiert ist". In der Programmierwelt wird dieses Verhalten mit dem Begriff '*scope*' erklärt.

So eine Funktion ist ein bisschen wie eine Insel mitten im Ozean. Es ist viel zu weit um von der Insel wegschwimmen zu können. Manchmal fliegt ein Flugzeug über die Insel und wirft Papierzettel mit Nachrichten ab (das sind die Parameter, die in die Funktion übernommen werden). Die Bewohner der Insel kommen zusammen um die Nachrichten zu lesen und stecken die Antwort darauf in eine Flasche, die als Flaschenpost in den Ozean geworfen wird (das ist der Rückgabewert). Was die Eingeborenen auf der Insel machen, und wieviele es genau sind, ist für die Person, die die Flaschenpost öffnet, nicht von Bedeutung. Das ist wahrscheinlich die einfachste Art scope zu erklären—bis auf ein kleines wichtiges Detail. Einer der Inselbewohner hat ein großes Fernglas und kann bis aufs Festland schauen. Da sieht er, was die anderen Leute so machen, und das kann die Antwort in der Flaschenpost auch beeinflussen.

```
>>> x = 100
>>> def variable_test2():
...     a = 10
...     b = 20
...     return a * b * x
...
>>> print(variable_test2())
20000
```

Obwohl die Variablen a und b nicht außerhalb der Funktion sichtbar sind, kann die Variable x (die außerhalb der Funktion erzeugt wurde) innerhalb der Funktion verwendet werden. Denke an den Inselbewohner mit dem Fernglas, und vielleicht tust du dir dann leichter.

Um die Schleife aus dem vorigen Kapitel (mit den Ersparnissen über das Jahr) darzustellen, kann praktischerweise eine Funktion verwendet werden.

```
>>> def jahres_ersparnisse(hausarbeit, zeitung, ausgaben):
...     ersparnisse = 0
...     for woche in range(1, 53):
...         ersparnisse = ersparnisse + hausarbeit + zeitung - ausgaben
...         print('Woche %s = %s' % (woche, ersparnisse))
...
```

Gib die Funktion in die Konsole ein und rufe sie mit verschiedenen Werten für hausarbeit, zeitung und ausgaben auf:

Ausgabe

```
>>> jahres_ersparnisse(10, 10, 5)
Woche 1 = 15
Woche 2 = 30
Woche 3 = 45
Woche 4 = 60
Woche 5 = 75
Woche 6 = 90
Woche 7 = 105
Woche 8 = 120
Woche 9 = 135
Woche 10 = 150
Woche 11 = 165

(geht so weiter...)

>>> jahres_ersparnisse(25, 15, 10)
Woche 1 = 30
Woche 2 = 60
Woche 3 = 90
Woche 4 = 120
Woche 5 = 150

(continues on...)
```

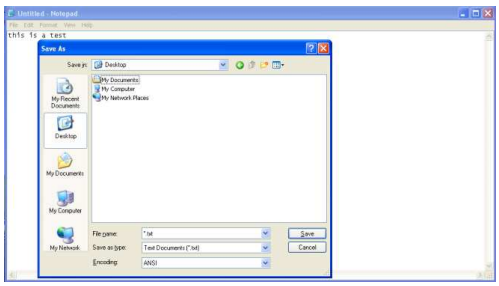


Abbildung 6.1: Der Speichern Dialog im Windows Notepad.

Das ist doch schon schöner, als die ganze for-Schleife immer wieder neu zu tippen, wenn du neue Werte übergeben willst. Mehrere Gruppen können auch in sogenannte 'Module' zusammengefasst werden und da bringt Python auch schon einige fixfertige Modlue mit.
Mehr über Module etwas später.

6.1 Dies und das

Wenn Python an deinem Computer installiert wird, wird auch ein ganzer Haufen von Funktionen und Modulen mitinstalliert. Einige Funktionen sind sofort verfügbar, so wie range. Andere Funktionen, so wie file auch, aber das haben wir noch nicht verwendet.

Um zu erklären, wie du file verwenden kannst, öffne Notepad und tippe ein paar Worte ein und speichere die Datei auf der Festplatte unter C:

1. Klicke ins Datei Menü und gehe auf Speichern,
2. Doppelklicke auf 'Mein Computer' im Datei Dialog,
3. Klicke auf (C:) doppelt,
4. benenne die Datei 'test.txt'

Nun kannst du die Python Konsole wieder öffnen und Folgendes eintippen:

```
>>> f = open('c:\\test.txt')
>>> print(f.read())
```

Der Inhalt der Datei, die du eben erzeugt und gespeichert hast, sollte auf der Konsole ausgegeben werden.

Was macht denn nun dieser kurze Code? Die erste Zeile ruft die Funktion file auf und übergibt als Parameter den Dateinamen. Die Funktion gibt eine spezielle Variable (Objekt genannt) zurück, die die Datei darstellt. Es ist nicht die Datei selber. Es ist mehr wie ein Finger, der auf die Datei zeigt und sagt "DA IST ES!!!" Das File Objekt (Datei Objekt) ist in der Variable f gespeichert.

Die zweite Zeile ruft die spezielle Funktion read auf, um den Inhalt der Datei zu lesen und mit der print Funktion auszugeben. Da die Variable f das Datei Objekt enthält, wird die read Funktion auf die f Variable angewendet indem ein Punkt hinter das Objekt geschrieben wird.

> Anhang B (am Ende diese Buches) listet die wichtigsten Python Funktionen auf.

6.2 Module

Jetzt haben wir schon einige Varianten gesehen, wie man Code wieder verwenden kann. Eine davon ist, eine selber angelegte Funktion zu verwenden oder eine bereits in Python eingebaute Funktion (wie range, file oder str). Eine andere Variante ist spezielle Funktionen auf Objekte anzuwenden—indem wir einen Punkt hinter das Objekt schreiben. Und die nächste Variante wäre die Verwendung von Modulen, wie zum Beispiel das 'time':

```
>>> import time
```

Mit dem Import Kommando sagen wir Python, dass wir ein Modul verwenden wollen. Im obigen Beispiel wird das Modul 'time' eingebunden. Danach können wir Funktionen und Objekte verwenden, die in diesem Modul enthalten sind, indem wir wieder das Punkt Symbol verwenden:

```
>>> print(time.localtime())
time.struct_time(tm_year=2010, tm_mon=1, tm_mday=1, tm_hour=17, tm_min=39,
tm_sec=51, tm_wday=4, tm_yday=1, tm_isdst=0)
```

localtime ist eine Funktion *innerhalb* vom time Modul, welches das aktuelle Datum und die Zeit zurückgibt—aufgeteilt in Jahr, Monat, Tag, Stunde, Minute, Sekunde, Tag der Woche, Tag im Jahr und ob es Sommer- oder Winterzeit ist. Die Werte sind in einem Tupel (schlag nach bei *Tupel und Listen* auf Seite 18). Du kannst eine andere Funktion vom time Modul verwenden um die zurückgegebenen Werte verständlicher darzustellen.

```
>>> t = time.localtime()
>>> print(time.asctime(t))
Fri Jan  1 17:53:14 2010
```

Das alles kannst du auch in eine Zeile verpacken, wenn du magst:

```
>>> print(time.asctime(time.localtime()))
Fri Jan  1 17:53:14 2010
```

Nehmen wir an, du wolltest jemanden um eine Tastatureingabe fragen. Fragen könntest du mit print, und die Antwort könntest du mit dem 'sys' Modul verarbeiten—sys kannst du genau gleich wie das time Modul einbinden:

```
import sys
```

Im sys Modul gibt es ein Objekt mit dem Namen 'stdin' (Englisch für standard input). In stdin gibt es die nützliche Methode (oder Funktion) mit dem Namen readline—verwendet wird es um Eingaben der Tastatur zu verarbeiten (die Eingabe wird durch die Eingabetaste beendet). Teste readline, indem du Folgendes in die Python Konsole eingibst:

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

Danach tippst du irgendwelche Wörter und drückst am Ende die Eingabetaste. Alles, was du bis dahin eingegeben hast, wird nochmals auf der Python Konsole ausgegeben. Denke nochmals kurz an das Beispiel mit der if-Schleife:

```
if alter >= 10 and alter <= 13:
    print('Du bist %s' % alter)
else:
    print('was?')
```

Statt die Variable alter schon vorher zu erzeugen, bitten wir jemanden sein Alter einzugeben. Zuerst machen wir aus dem Code eine Funktion. . .

```
>>> def dein_alter(alter):
...     if alter >= 10 and alter <= 13:
...         print('Du bist %s' % alter)
...     else:
...         print('was?')
...
```

Diese Funktion kannst du aufrufen, indem du eine Zahl als Parameter mitgibst. Testen wir zuerst, ob es richtig funktioniert:

```
>>> dein_alter(9)
was?
>>> dein_alter(10)
Du bist 10
```

Jetzt wissen wir, dass unsere Funktion funktioniert. Nun ändern wir die Funktion, damit sie nach dem Alter der Person fragt:

```
>>> def dein_alter():
...     print('Bitte gib dein Alter ein')
...     alter = int(sys.stdin.readline())
...     if alter >= 10 and alter <= 13:
...         print('Du bist %s' % age)
...     else:
...         print('was?')
...
```

Weil readline() das Eingetippte als Text zurückgibt (oder anders gesagt als string), müssen wird das zuerst mit Hilfe der Funktion int in eine Zahl verwandeln (damit die if-Abfrage richtig funktioniert). Schau unter *Was ist der Unterschied* auf Seite 31 nach für mehr Details dazu. Rufe zum Ausprobieren die Funktion dein_alter ohne Parameter auf und gib etwas ein, wenn ‘Bitte gib dein Alter ein’ da steht:

```
>>> dein_alter()
Bitte gib dein Alter ein
10
Du bist 10
>>> dein_alter()
Bitte gib dein Alter ein
15
was?
```

[Das Wichtige zum Merken hier ist, dass obwohl du eine Zahl eingegeben hast (einmal 10 und einmal 15) readline immer einen String zurückgibt.

> Die Module sys und time sind nur zwei von den vielen Modulen, die bei Python dabei sind. Für mehr Information zu weiteren (aber nicht allen) Python Modulen schlage bitte im Anhang C nach.

6.3 Probiere es aus

In diesem Kapitel haben wir gesehen, wie Wiederverwertung mit Python funktionert: durch Verwenden von Funktionen und Modulen. Wir haben ein wenig den 'scope' (Sichtbarkeit) von Variablen behandelt und wie Variablen ausserhalb von Funktionen 'gesehen' werden können. Unsere eigenen Funktionen haben wir mit def erzeugt.

Übung 1

In Übung 2 aus Kapitel 5 haben wir eine for-Schleife verwendet um die Zinsen zu berechnen, die wir aus 100 € über den Zeitraum von 10 Jahren bekommen würden. Diese for-Schleife lässt sich gut in eine Funktion verpacken. Versuche eine Funktion zu definieren, der man die Geldsumme und den Zinssatz mitgibt. Diese könntest du folgendermaßen aufrufen:

```
berechne_zinsen(100, 0.03)
```

Übung 2

Nimm die Funktion, die du gerade geschrieben hast, und passe sie so an, dass sie verschiedene Zeiträume als Parameter annimmt—wie 5 Jahre oder 15 Jahre. Der Aufruf der Funktion könnte so ausschauen:

```
berechne_zinsen(100, 0.03, 5)
```

Übung 3

Vielleicht wäre es noch besser, wenn wir die Parameter zuerst von einem Nutzer abfragen. Also schreiben wir eine Funktion, das nach den Werten für die Geldsumme, Zinssatz und Zeitdauer fragt. In diesem Fall rufen wir die Funktion ohne Parameter auf:

```
berechne_zinsen()
```

Um dieses kleine Programm zu erstellen, brauchst du die Funktion float. Wir haben sie noch nicht behandelt, aber sie macht das Umwandeln sehr ähnlich wie die int Funktion. Sie konvertiert strings aber nicht zu Ganzzahlen (Integer) sondert zu Fließkommazahlen (floating point number, wie in Kapitel 2 erklärt). Fließkommazahlen sind Nummern mit Kommastellen, wie 20.3 oder 2541.933.

7 Ein kurzes Kapitel über Dateien

Wahrscheinlich weißt du schon, was eine Datei ist.
Wenn deine Eltern ein Büro zu Hause haben, ist sicher irgendwo auch ein Aktenschrank zu finden. Da sind verschiedene wichtige Zettel drin (meist ziemlich langweilige). Die Zettel sind sicher in verschiedenen Mappen einsortiert und die Mappen (Aktenordner) sind vermutlich mit Buchstaben oder Zahlen beschriftet und nach dem Alphabet sortiert. Dateien auf dem Computer sind diesen Aktenordnern recht ähnlich. Die Dateien sind auch beschriftet (entspricht dem Namen der Datei), und sie speichern wichtige Informationen. Die Aktenordner um die Zettel zu sortieren, entsprechen den Verzeichnissen (oder auch Ordner genannt) im Computer.

Im vorigen Kapitel haben wir schon eine Datei mit Python erstellt. Das Beispiel hat so ausgesehen:

```
>>> f = open('c:\\test.txt')
>>> print(f.read())
```

Mit dem Datei Objekt in Python kannst du mehr machen als nur lesen. Aktenschränke wären auch nicht besonders sinnvoll, wenn du nur Dinge rausnehmen, aber nicht mehr zurückbringen kannst. Wir können eine neue leere Datei erstellen, indem wir andere Parameter der Python Datei Funktion mitgeben.

```
>>> f = open('meine_datei.txt', 'w')
```

Mit dem Parameter **w** (kommt vom englischen write) sagen wir Python, dass wir in die Datei auch schreiben wollen. Mit der Funktion write können wir nun Information in die Datei schreiben.

```
>>> f = open('meine_datei.txt', 'w')
>>> f.write('das hier ist ein Test')
```

Wenn wir fertig mit schreiben sind, müssen wir es Python sagen, und Python macht dann die Datei zu—wir verwenden dazu die Funktion close.

alle Befehle zusammengefasst

```
>>> f = open('meine_datei.txt', 'w')
>>> f.write('das hier ist ein Test')
>>> f.close()
```

Wenn du nun die Datei mit dem Editor deiner Wahl öffnest, wirst du den Text: “das hier ist ein Test” sehen. Oder wir verwenden wieder Python um es wieder einzulesen.

```
>>> f = open('meine_datei.txt')
>>> print(f.read())
das hier ist ein Test
```

8 Invasion der Schildkröten

Lass uns wieder das turtle (Schildkröte) Modul anschauen, das wir schon in Kapitel 3 angesehen haben. Um zuerst wieder die Leinwand zu erzeugen, müssen wir das Modul importieren und dann das 'Pen' Objekt erzeugen:

```
>>> import turtle
>>> schildkroete = turtle.Pen()
```

Jetzt können wir wieder mit den bekannten Funktionen die Schildkröte auf der Leinwand herumbewegen und einfache Formen zeichnen. Oder aber wir verwenden ein paar Dinge, die wir in den letzten Kapiteln gelernt haben. Bis jetzt hat unser Code um ein Rechteck zu zeichnen so ausgesehen:

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

Wenn wir das mit einer for-Schleife machen, könnte es so aussehen:

```
>>> schildkroete.reset()
>>> for x in range(1,5):
...     schildkroete.forward(50)
...     schildkroete.left(90)
...
```

Das ist schon viel weniger Tipparbeit, und damit es interessanter aussieht, probier das Folgende:

```
>>> schildkroete.reset()
>>> for x in range(1,9):
...     schildkroete.forward(100)
...     schildkroete.left(225)
...
```

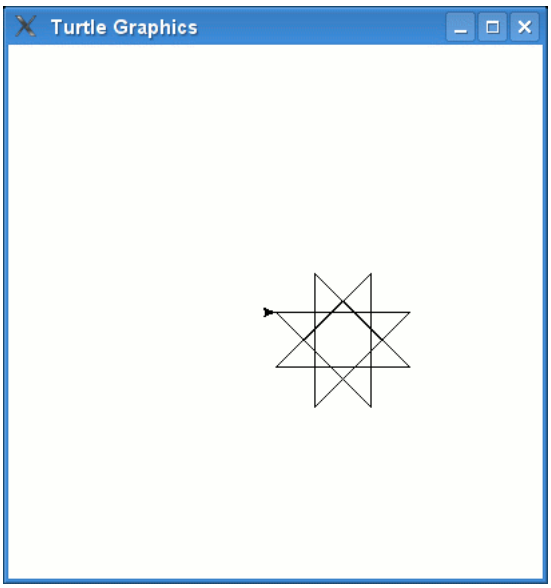


Abbildung 8.1: Die Schildkröte zeichnet einen 8-zackigen Stern.

Dieser Code erzeugt einen 8-zackigen Stern wie in Abbildung 8.1 (die Schildkröte dreht sich immer um 225 Grad, jedes Mal wenn sie 100 Pixel zurückgelegt hat).
Mit einem anderen Winkel (175 Grad) und einer längeren Schleife (37 Mal), können wir einen Stern mit noch mehr Punkten (wie in Abbildung 8.2) zeichnen:

```
>>> schildkroete.reset()
>>> for x in range(1,38):
...     schildkroete.forward(100)
...     schildkroete.left(175)
...
```

Probier mal diesen Code aus, der einen spiralenähnlichen Stern wie in Abbildung 8.3 erzeugt:

```
>>> for x in range(1,20):
...     schildkroete.forward(100)
...     schildkroete.left(95)
...
```

Und hier etwas Komplizierteres:

```
>>> schildkroete.reset()
>>> for x in range(1,19):
...     schildkroete.forward(100)
...     if x % 2 == 0:
...         schildkroete.left(175)
...     else:
...         schildkroete.left(225)
...
```

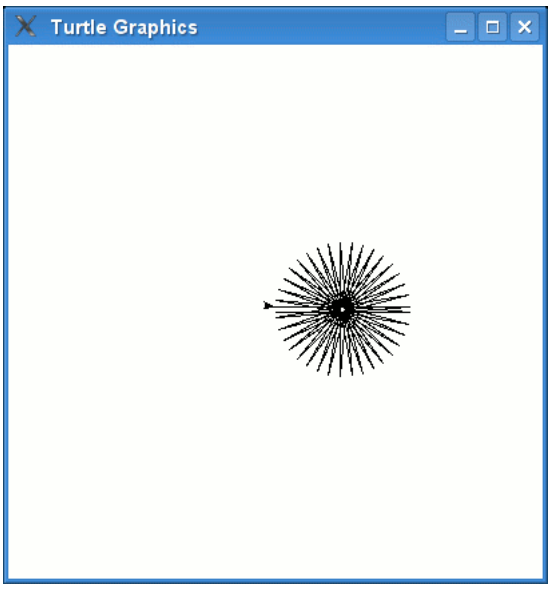


Abbildung 8.2: Ein Stern mit vielen spitzen Zacken.

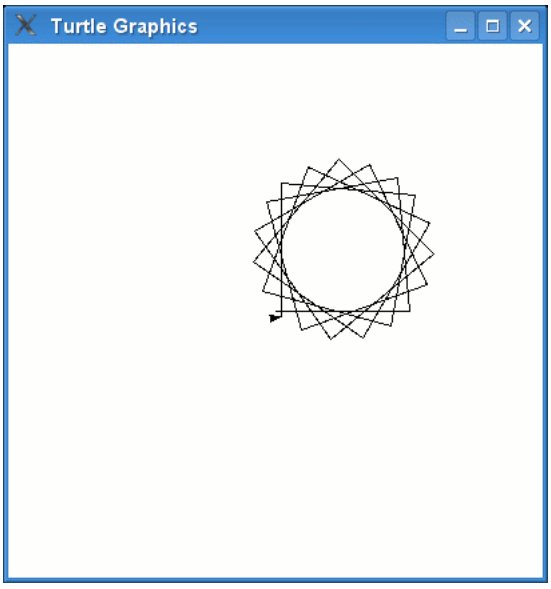


Abbildung 8.3: Ein Stern mit vielen Zacken.

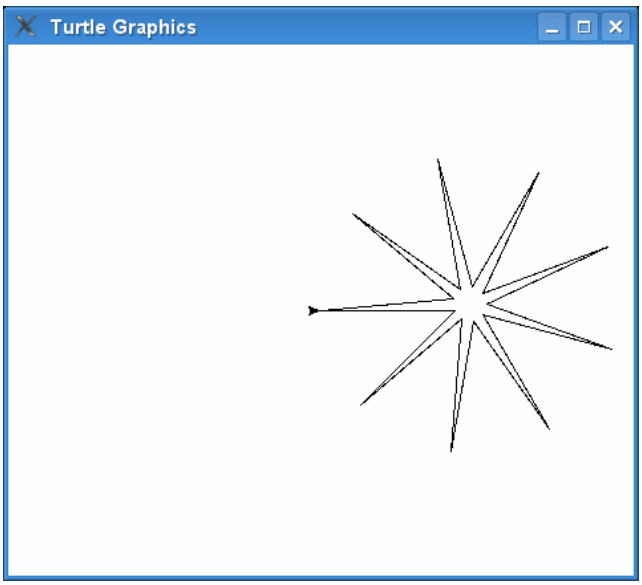


Abbildung 8.4: Ein 9-zackiger Stern

Im obigen Code wird überprüft, ob x eine gerade Zahl enthält. Dafür verwenden wir den sogenannten modulo Operator (%)
indexOperator!Modulo, in der Zeile if x % 2 == 0:.

x % 2 ist dann Null, wenn die Nummer in der Variable x durch 2 dividiert werden kann, ohne dass ein Rest übrigbleibt—wenn das keinen Sinn macht, mach dir keine Sorgen. Merke dir nur, dass du 'x % 2 == 0' verwenden kannst, um herauszufinden, ob eine Nummer gerade ist. Wenn du den Code ausführst, wirst du den 9-zackigen Stern von Abbildung 8.4 sehen.

Du musst nicht immer nur Sterne und einfache Formen zeichnen. Deine Schildkröte kann auch ganz andere Dinge zeichnen. Zum Beispiel:

```
schildkroete.color(1,0,0)
schildkroete.begin_fill()
schildkroete.forward(100)
schildkroete.left(90)
schildkroete.forward(20)
schildkroete.left(90)
schildkroete.forward(20)
schildkroete.right(90)
schildkroete.forward(20)
schildkroete.left(90)
schildkroete.forward(60)
schildkroete.left(90)
schildkroete.forward(20)
schildkroete.right(90)
schildkroete.forward(20)
schildkroete.left(90)
schildkroete.forward(20)
schildkroete.end_fill()
schildkroete.color(0,0,0)
schildkroete.up()
schildkroete.forward(10)
```

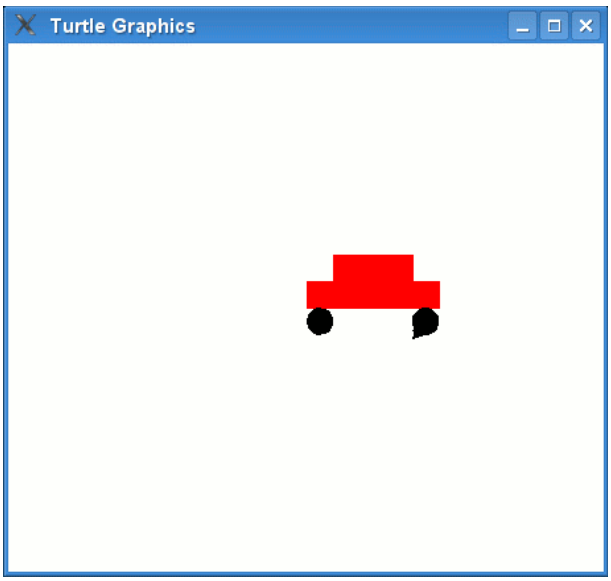


Abbildung 8.5: Die Schildkröte muss noch üben um schöne Autos zu zeichnen!

```
schildkroete.down()
schildkroete.begin_fill()
schildkroete.circle(10)
schildkroete.end_fill()
schildkroete.setheading(0)
schildkroete.up()
schildkroete.forward(90)
schildkroete.right(90)
schildkroete.forward(10)
schildkroete.setheading(0)
schildkroete.begin_fill()
schildkroete.down()
schildkroete.circle(10)
schildkroete.end_fill()
```

Was eine langsame Methode ist, ein ziemlich einfach aussehendes Auto zu zeichnen (siehe Abbildung 8.5. Aber es zeigt einige neue Funktionen, die die Schildkröte beherrscht: color, um die Farbe des Zeichenstifts zu ändern, fill, um einen Bereich mit Farbe zu füllen, und circle, um einen Kreis mit einer bestimmten Größe zu zeichnen.

8.1 Mit Farben füllen

Die color Funktion nimmt 3 Parameter.

Warum rot, grün und blau?

Wenn du schon mal mit verschiedenen Farben des Malkastens herumgespielt hast, dann kennst du die Antwort schon fast. Wenn du zwei verschiedene Farben mischt, bekommst du eine andere Farbe[1]. Wenn du beim Malen Blau und Rot zusammenmischt, bekommst du Lila. . . und wenn du zu

[1]Eigentlich sind die 3 **Grundfarben** beim Malen rot, gelb und blau, und nicht rot, grün und blau (RGB) wie auf dem Computer. Beim Malen gibt es das Prinzip der substraktiven Farbmischung und beim Computer ist es das Prinzip der additiven Farbmischung.

viele Farben mischt, erhältst du für gewöhnlich einen bräunlichen Ton. Auf dem Computer kannst du Farben in einer ähnlichen Weise mischen—bringe rot und grün zusammen, und es kommt gelb raus—beim Computer mischt du aber Lichter und beim Malen Farben.

Denke dir drei große Farbtöpfe mit Farbe. Ein roter, ein grüner und ein blauer Farbtopf. Jeder Topf ist voll, und wir sagen die Farbe hat den Wert 1 (oder 100%). Dann leeren wir den ganzen roten (100%) und grünen (100%) Topf in einen leeren Behälter und mischen. Nach etwas Rühren bekommen wir eine gelbe Farbe. Lass uns nun einen gelben Kreis mit unserer Schildkröte zeichnen:

```
>>> schildkroete.color(1,1,0)
>>> schildkroete.begin_fill()
>>> schildkroete.circle(50)
>>> schildkroete.end_fill()
```

Im obigen Beispiel rufen wir die color-Funktion mit 100% für rot, 100% für grün und 0% für blau auf (oder in anderen Worten 1, 1, 0). Um das Experimentieren mit anderen Farben einfacher zu machen, packen wir das alles in eine Funktion:

```
>>> def mein_kreis(rot, gruen, blau):
...     schildkroete.color(rot, gruen, blau)
...     schildkroete.begin_fill()
...     schildkroete.circle(50)
...     schildkroete.end_fill()
...
```

Jetzt können wir einen hellen grünen Kreis mit der ganzen grünen Farbe (1 oder 100%) zeichnen:

```
>>> mein_kreis(0, 1, 0)
```

Und einen Kreis mit dunklerem Grün können wir zeichnen, indem wir nur das halbe grün (0.5 oder 50%) verwenden.

```
>>> mein_kreis(0, 0.5, 0)
```

Hier macht die Analogie mit dem Farbtopf nicht mehr viel Sinn. In der echten Welt ist es egal, wieviel man von einer Farbe nimmt, sie schaut immer gleich aus. Da am Computer die Farben aus Licht bestehen, macht es einen Unterschied. Weniger Licht (weniger Farbe) erscheint dunkler. Das ist das Gleiche mit einer Taschenlampe in der Nacht. Wenn die Batterien immer schwächer werden, wird das Licht immer dunkler. Probiere es am Computer selber aus, indem du einen roten Kreis (1, 0, 0), einen 'halbroten' (0.5, 0, 0), und einen blauen und 'halbblauen' Kreis zeichnest.

```
>>> mein_kreis(1, 0, 0)
>>> mein_kreis(0.5, 0, 0)

>>> mein_kreis(0, 0, 1)
>>> mein_kreis(0, 0, 0.5)
```

Verschiedenen Kombinationen von rot, grün und blau erzeugen alle möglichen Farben. Eine goldene Farbe ergibt sich, wenn du 100% rot, 85% grün und kein blau verwendest:

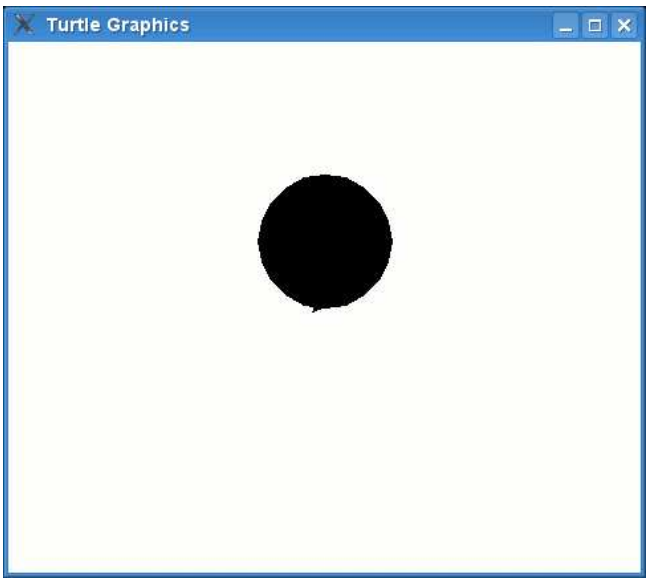


Abbildung 8.6: Ein schwarzes Loch!

```
>>> mein_kreis(1, 0.85, 0)
```

Eine rosarote Farbe erreichst du mit 100% rot, 70% grün und 75% blau:

```
>>> mein_kreis(1, 0.70,0.75)
```

Orange ist eine Mischung aus 100% rot und 65% grün; Braun ist 60% rot, 30% grün und 15% blau:

```
>>> mein_kreis(1, 0.65, 0)
>>> mein_kreis(0.6, 0.3, 0.15)
```

Die Leinwand kannst du mit schildkroete.clear() wieder löschen.

8.2 Die Dunkelheit

Eine kleine Frage für dich: Was passiert, wenn du in der Nacht alle Lampen ausschaltest? Alles wird schwarz.

Das gleiche passiert mit den Farben am Computer. Kein Licht bedeutet keine Farbe. Ein Kreis mit 0 für rot, 0 für grün und 0 für blau schaut so aus:

```
>>> mein_kreis(0, 0, 0)
```

Und das ergibt einen dicken, schwarzen Punkt (wie in Abbildung 8.6).

Und auch das Gegenteil ist wahr: wenn du 100% für rot, 100% für grün und 100% für blau verwendest, erhältst du weiß. Nimm den folgenden Code, und der schwarze Kreis wird wieder ausgelöscht:

```
>>> mein_kreis(1,1,1)
```

8.3 Dinge füllen

Du hast vielleicht herausgefunden, dass durch die Übergabe des Parameters '1' die fill Funktion eingeschalten wird. Mit dem Parameter '0' wir die Funktion wieder ausgeschalten. Beim ausschalten wird die bisher gezeichnete Form ausgemalt. So können wir einfach aufgefüllte Quadrate zeichnen indem wir den Code von früher verwenden. Um ein Quadrat mit der Schildkröte zu zeichnen gibst du folgendes ein:

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

Etwas flexibler wird das ganze, wenn wir eine Funktion erzeugen und die Größe des Quadrats als Parameter mitgeben:

```
>>> def mein_quadrat(groesse):
...     schildkroete.forward(groesse)
...     schildkroete.left(90)
...     schildkroete.forward(groesse)
...     schildkroete.left(90)
...     schildkroete.forward(groesse)
...     schildkroete.left(90)
...     schildkroete.forward(groesse)
...     schildkroete.left(90)
```

Du kannst die Funktion aufrufen, indem du Folgendes eingibst:

```
>>> mein_quadrat(50)
```

Als Anfang schon ganz gut, aber noch nicht perfekt. Wenn du den Code oben anschaust, wirst du ein Muster erkennen. Wir wiederholen: forward(groesse) und left(90) vier Mal. Das ist unnötige Tipparbeit. Wir können eine for-Schleife verwenden, die uns das erspart:

```
>>> def mein_quadrat(groesse):
...     for x in range(0,4):
...         schildkroete.forward(groesse)
...         schildkroete.left(90)
```

Das ist eine große Verbesserung im Vergleich zur vorigen Version. Testen wir die Funktion mit verschiedenen Größen (verschiedenen Parametern):

```
>>> schildkroete.reset()
>>> mein_quadrat(25)
>>> mein_quadrat(50)
>>> mein_quadrat(75)
>>> mein_quadrat(100)
>>> mein_quadrat(125)
```

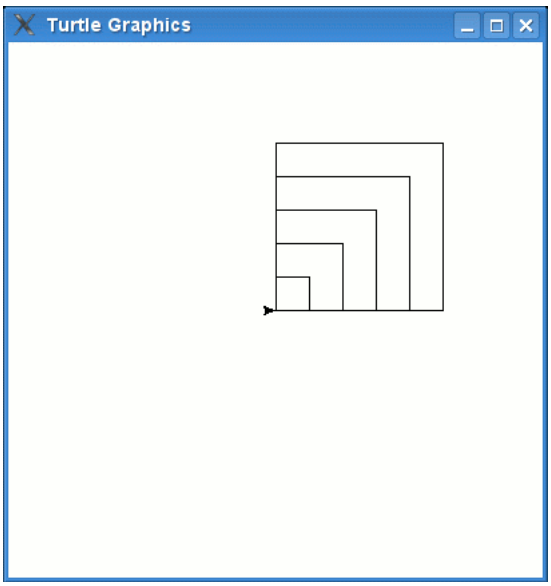


Abbildung 8.7: Viele Quadrate.

Und unsere Schildkröte wird etwas zeichnen das wie in Abbildung 8.7 aussieht.
Jetzt versuchen wir das Quadrat mit Farbe zu füllen. Zuerst löschen wir die Leinwand:

```
>>> schildkroete.reset()
```

Dann schalten wir 'ausfüllen' ein und rufen nochmals die Funktion mein_quadrat auf:

```
>>> schildkroete.begin_fill()
>>> mein_quadrat(50)
```

Du siehst immer noch das leere Quadrat bis du die Funktion 'ausfüllen' wieder ausschaltest.

```
>>> schildkroete.end_fill()
```

Und nun erhältst du etwas wie Bild 8.8.

Sollen wir unsere Funktion ein wenig verändern, damit wir entweder befüllte oder unbefüllte Quadrate zeichen können? Um das zu erreichen, verwenden wir einen zusätzlichen Parameter:

```
>>> def mein_quadrat(groesse, fuellen):
...     if fuellen == True:
...         schildkroete.begin_fill()
...     for x in range(0,4):
...         schildkroete.forward(groesse)
...         schildkroete.left(90)
...     if fuellen == True:
...         schildkroete.end_fill()
...
```

Die ersten zwei Zeilen überprüfen, ob der Parameter 'fuellen' auf True gesetzt wurde. Wenn ja, wird filling eingeschalten. Dann wird die Schleife vier Mal durchlaufen und danach wieder der Paramter 'fuellen' überprüft. Wenn der Wert True ist, dann wird filling wieder ausgeschalten. Ein gefülltes Quadrat kann also so gezeichnet werden:

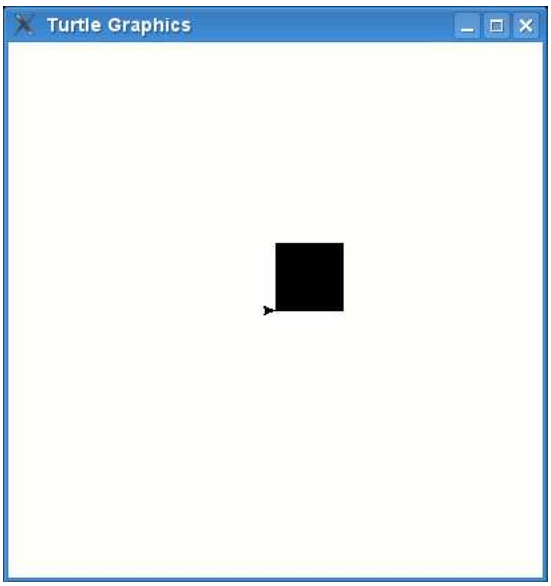


Abbildung 8.8: Ein schwarzes Quadrat.

```
>>> mein_quadrat(50, True)
```

Und ein leeres Quadrat so:

```
>>> mein_quadrat(150, False)
```

Als Ergebnis zeichnet die Schildkröte ein Bild wie in Abbildung 8.9. . . . . und das schaut ein wenig wie ein seltsames, viereckiges Auge aus.

Du kannst alle möglichen Formen zeichnen und sie mit Farbe füllen. Lass uns den Stern von vorhin in eine Funktion verpacken. Der Originalcode hat so ausgesehen:

```
>>> for x in range(1,19):
...     schildkroete.forward(100)
...     if x % 2 == 0:
...         schildkroete.left(175)
...     else:
...         schildkroete.left(225)
...
```

Wir können die gleiche if-Bedingung von der mein_stern Funktion nehmen und als Parameter die Größe übergeben:

```
1.  >>> def mein_stern(groesse, fuellen):
2.  ...     if fuellen:
3.  ...         schildkroete.begin_fill()
4.  ...     for x in range(1,19):
5.  ...         schildkroete.forward(groesse)
6.  ...         if x % 2 == 0:
7.  ...             schildkroete.left(175)
```

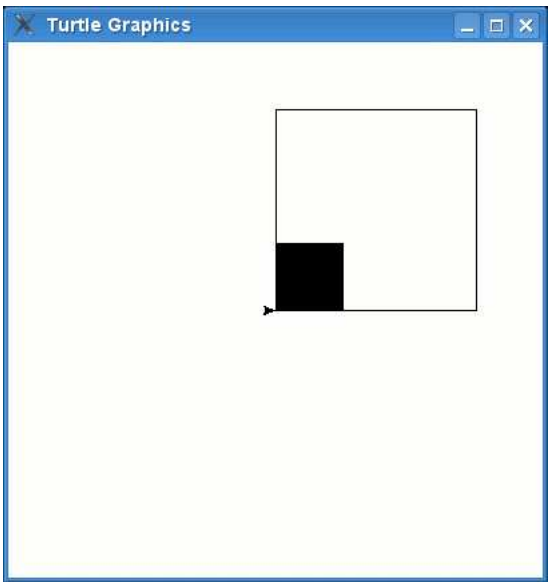


Abbildung 8.9: A square eye.

```
8.  ...          else:
9.  ...              schildkroete.left(225)
10. ...      if fuellen:
11. ...          schildkroete.end_fill()
```

In Zeile 2 und 3 wird abhängig vom Parameter fuellen die Funktion begin_fill() ausgeführt oder nicht (bei 'True' ist es an, bei 'False' ist es aus). Das Gegenteil machen wir bei Zeile 10 und 11. Zusätzlich übergeben wir die Größe als Parameter und verwenden sie in Zeile 5.

Lass uns die Farbe nun auf Gold (besteht aus 100% rot, 85% grün und ohne blau) setzen und die Funktion nochmals aufrufen.

```
>>> schildkroete.color(1, 0.85, 0)
>>> mein_stern(120, True)
```

Die Schildkröte sollte nun den goldenen Stern in figure 8.10 zeichnen. Für die schwarze Umrandung können wir die Farbe nochmals ändern (diesmal auf schwarz) und den Stern nochmals zeichnen ohne ihn mit schwarz auszufüllen.

```
>>> schildkroete.color(0,0,0)
>>> mein_stern(120, False)
```

Und so schaut der Stern nun aus wie Abbildung 8.11.

8.4 Probiere es aus

In diesem Kapitel haben wir mehr über das turtle Modul gelernt und es verwendet um geometrische Formen zu zeichnen. Funktionen wurden verwendet um Code wieder zu verwenden und es einfacher zu machen verschiedene Formen zu zeichnen und zu füllen.

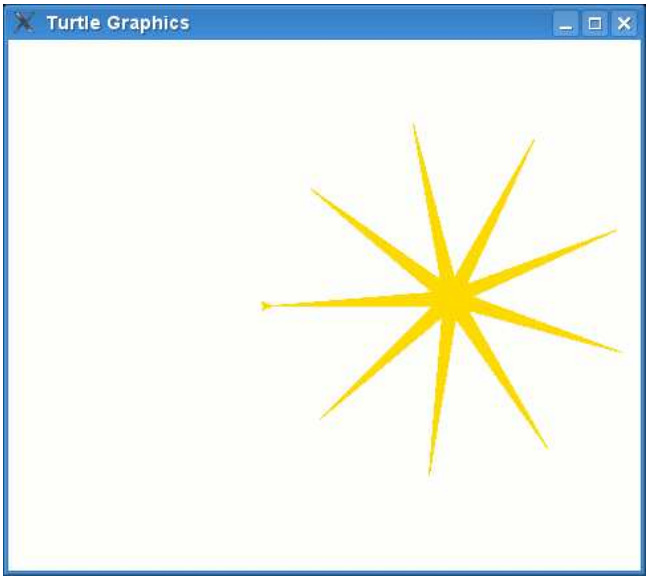


Abbildung 8.10: Ein goldener Stern.

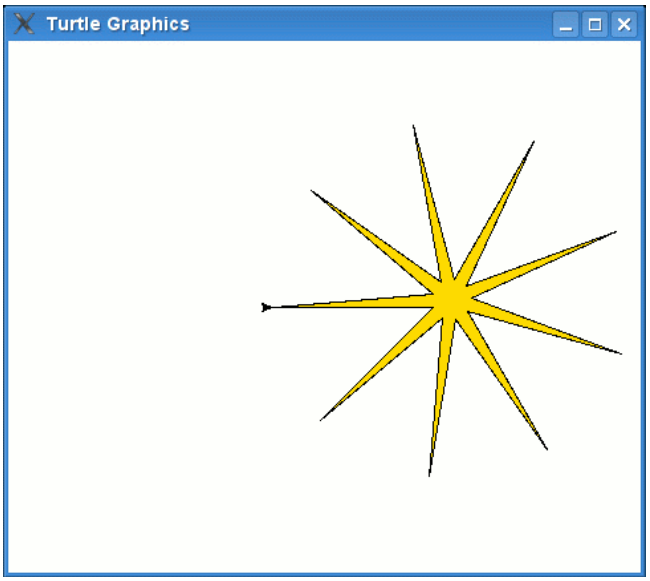


Abbildung 8.11: Ein Stern mit Umriss.

Übung 1

Wir haben Sterne, Quadrate und Rechtecke gezeichnet. Wie schaut es mit einem Achteck aus? Ein Achteck ist zum Beispiel die Stopptafel, die du oft auf Kreuzungen sehen kannst. (Tipp: versuche mit 45 Grad zu zeichnen).

Übung 2

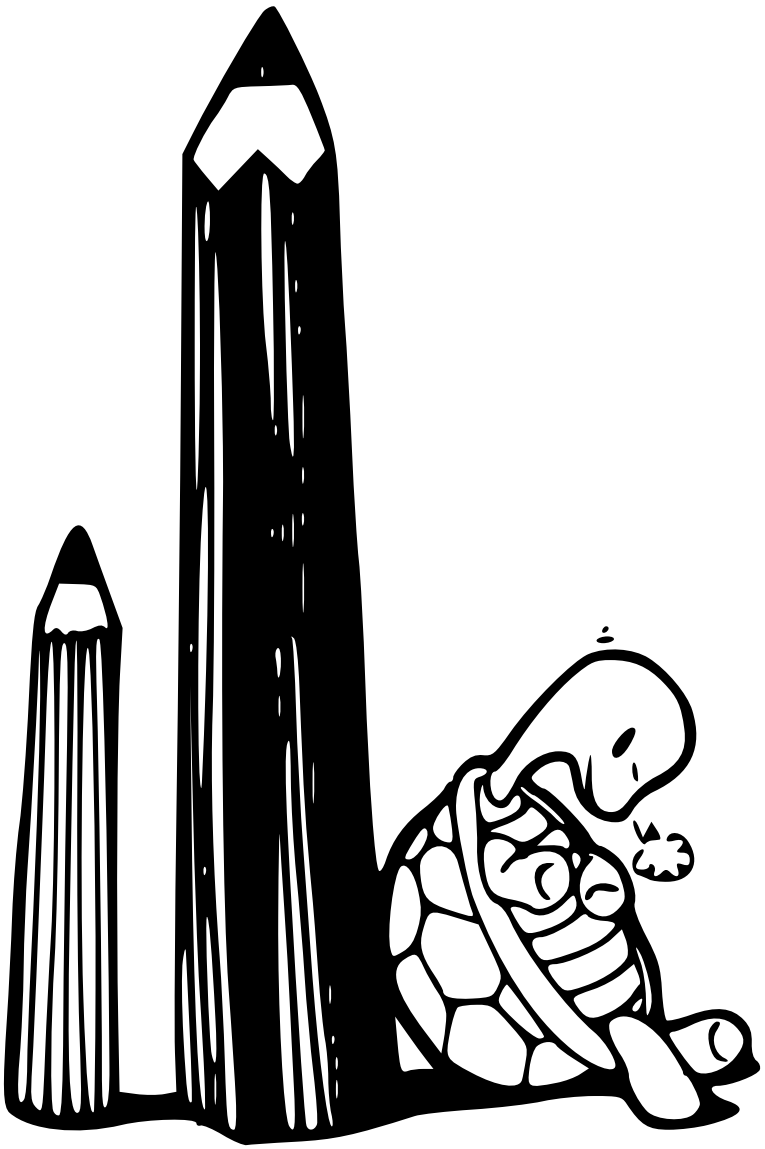
Verpacke den Code in eine Funktion und fülle das Achteck mit einer Farbe.

9 Zeichnen

Das Problem wenn man Schildkröten zeichnen lässt ist,. . . dass. . .. . . Schildkröten. . .. . wirklich. . .. . sehr. . .. . langsam sind.

Auch wenn die Schildkröte Vollgas gibt, ist sie immer noch langsam. Das macht den Schildkröten nichts aus—sie haben genug Zeit—aber wenn wir Computergraphik machen möchten, wird es zu einem Problem. Wenn du einen Nintendo DS hast oder Spiele auf dem Computer spielst, kannst du dir mal die Grafiken und Bilder genauer anschauen. Da gibt es ganz verschiedene. Manche Spiele sind in 2D (zweidimensional). Da sind die Bilder flach und die Spielfiguren bewegen sich normalerweise nur nach rechts, links, oben oder unten. Dann gibt es noch pseudo-3D-Spiele (fast dreidimensional), bei denen die Spielfiguren schon etwas echter ausschauen, und zu guter Letzt gibt es 3D-Spiele bei denen die Bilder die Realität nachahmen.

Aber alle Arten haben etwas gemeinsam—das Zeichnen auf dem Bildschirm muss schnell gehen. Hast du schon mal ein Daumenkino gemacht? Also an den weißen Rand eines Buches ein Bild oder ein Strichmännchen gezeichnet. Auf der nächsten Seite wieder das gleiche Männchen, aber die Hand an einer etwas veränderten Position. Das machst du ein paar Mal und wenn du das Buch schnell genug durchblätterst, wirst du sehen, wie das Männchen sich bewegt. Das ist das Grundprinzip aller Animationen und auch des Fernsehers. Dinge werden ganz schnell hintereinander immer wieder gezeichnet, aber in leicht veränderter Form. So ergibt sich die Illusion der Bewegung. Also muss jedes Bild der Animation sehr schnell gezeichnet werden. Die Schildkröte wäre dazu zu langsam.

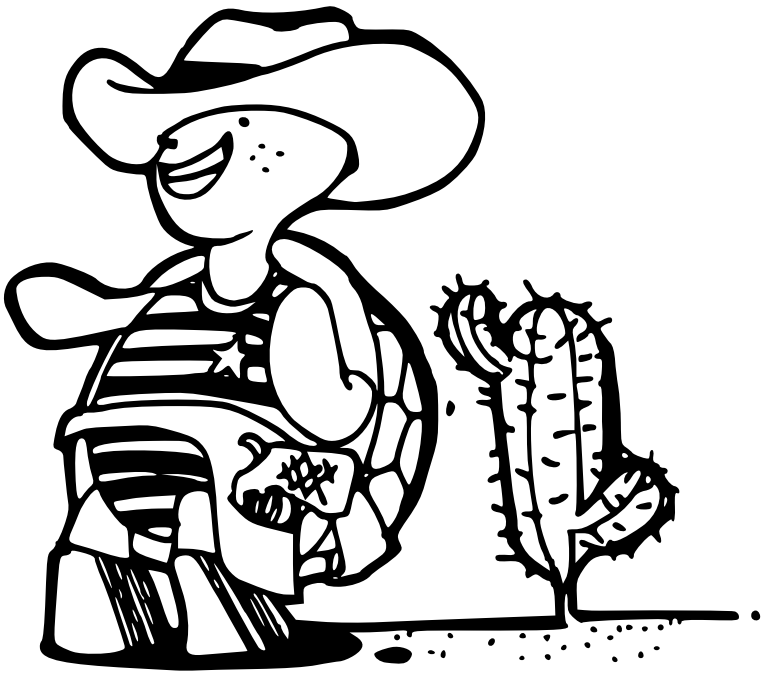


9.1 Quick Draw

In jeder Programmiersprache gibt es verschiedene Arten um auf den Bildschirm zu zeichnen. Manche sind langsam, und manche schnell. Spieleentwickler müssen ihre Programmiersprache gut auswählen.

Auch in Python gibt es verschiedene Arten Bilder zu zeichnen (turtle haben wir dazu schon verwendet), aber die beste Art normalerweise ist schon fertige Module und Bibliotheken zu verwenden, die nicht direkt bei Python dabei sind. Da brauchst du aber zuerst sicher einige Jahre Programmiererfahrung um diese komplexen Bibliotheken zu installieren und zu benutzen.

Zum Glück ist bei Python ein Modul dabei, welches wir für einfache Grafiken verwenden können, und schneller als das Turtle-Modul ist. Vielleicht sogar schnell genug um es die 'schnelle Schildkröte' zu nennen.



Das Modul heißt tkinter (ein komischer Name, der für 'TK Schnittstelle' steht) und kann für ganze Programme verwendet werden. Machen wir ein kleines Programm mit einem kleinem Knopf mit folgendem Code:

```
1. >>> from tkinter import *
2. >>> tk = Tk()
3. >>> knopf = Button(tk, text="Klick mich")
4. >>> knopf.pack()
```

In Zeile 1 importieren wir den Inhalt vom TK Modul, damit wir es verwenden können. Das nützlichste davon ist Tk, das uns ein Fenster erzeugt. Nachdem Zeile 2 bestätigt ist, erscheint das Fenster. In Zeile 3 erzeugen wir einen Knopf, indem wir tk als Parameter übergeben und als benannten Parameter 'Klick mich' mitgeben.

Benannte Parameter

Das ist das erste Mal, dass wir 'benannte Parameter' verwenden. Diese Parameter funktionieren wie normale Parameter, außer daß sie in beliebiger Reihenfolge verwendet werden können und so geben wir Namen mit.
Nehmen wir als Beispiel eine Funktion, die ein Rechteck zeichnet, und dazu die Parameter Breite und Höhe benötigt. Normalerweise könnten wir die Funktion so aufrufen: rechteck(200,100) und es würde ein Rechteck von 200 Pixel Breite und 100 Pixel Höhe gezeichnet werden. Aber was wäre, wenn die Reihenfolge der Parameter unwichtig ist? Woher würden wir wissen, was die Breite und was die Höhe ist?
In diesem Fall ist es besser diese Information mitzugeben. Zum Beispiel: rechteck(hoehe=100, breite=200). Tatsächlich sind die Details zu den benannten Parametern noch etwas kniffliger als das—aber es macht Funktionen flexibler und die Details kannst du immer noch in einem Python Buch für Fortgeschrittene nachlesen.

In der letzten Zeile (4) wird der Knopf dann gezeichnet. Gleichzeitig wird das Fenster kleiner, welches in Zeile 2 gezeichnet wurde, und es bleibt der Knopf mit dem Schriftzug 'Klick mich' übrig. Das schaut dann in etwa so aus:

Der Knopf macht zwar nichts, aber wenigsten kann man ihn anklicken. Damit der Knopf auch etwas macht, müssen wir den Code ein wenig ändern (mache zuerst aber auf jeden Fall das alte Fenster zu). Zuerst machen wir eine Funktion, die einen Text ausgibt:

```
>>> def hallo():
...     print('Hallo Du da!')
```

Dann ändern wir unser Beispiel, damit wir diese Funktion auch verwenden:

```
>>> from tkinter import *
>>> tk = Tk()
>>> knopf = Button(tk, text="Klick mich", command=hallo)
>>> knopf.pack()
```

Der benannte Parameter 'command' sagt, dass wir die hallo-Funktion ausführen wollen, wenn wir den Knopf drücken. Wenn du nun den Knopf drückst, wirst du "Hallo Du da!" auf der Konsole erscheinen sehen—und zwar jedes Mal, wenn du den Knopf drückst.

9.2 Einfaches Zeichnen

Knöpfe allein sind noch nicht so nützlich, wenn du Dinge auf dem Bildschirm zeichnen willst—wir brauchen noch eine andere Komponente dafür: eine Art Leinwand, Canvas. Wenn wir eine Leinwand hinzeichnen, müssen wir im Gegensatz zu einem einfachen Knopf Breite und Höhe mitgeben (in Pixel):

```
>>> from tkinter import *
>>> tk = Tk()
>>> leinwand = Canvas(tk, width=500, height=500)
>>> leinwand.pack()
```

Analog zum vorigen Beispiel wird nach dem Bestätigen der zweiten Zeile ein Fenster erscheinen. Nach dem 'pack' Befehl in Zeile 4 wird das Fenster plötzlich größer werden, damit es 500 Pixel breit und hoch ist. Nun können wir Linien auf dieser Leinwand zeichnen, indem wir die Pixelkoordinaten angeben. Koordinaten geben die Position von Punkten auf der Leinwand an. Auf einer Tk-Leinwand ist der Nullpunkt oben links. Die x-Koordinate gibt an, wie weit der Punkt rechts entlang der x-Achse liegt. Die y-Koordinate gibt an, wie weit dieser Punkt nach unten entlang der y-Achse ist.

Nachdem die Leinwand 500 Pixel breit und 500 Pixel hoch ist, sind die Koordinaten der rechten unteren Ecke genau 500,500. Um also die Linie in Abbildung 9.1 zu zeichnen kannst du 0,0 als Startkoordinate und 500,500 als Zielkoordinate eingeben:

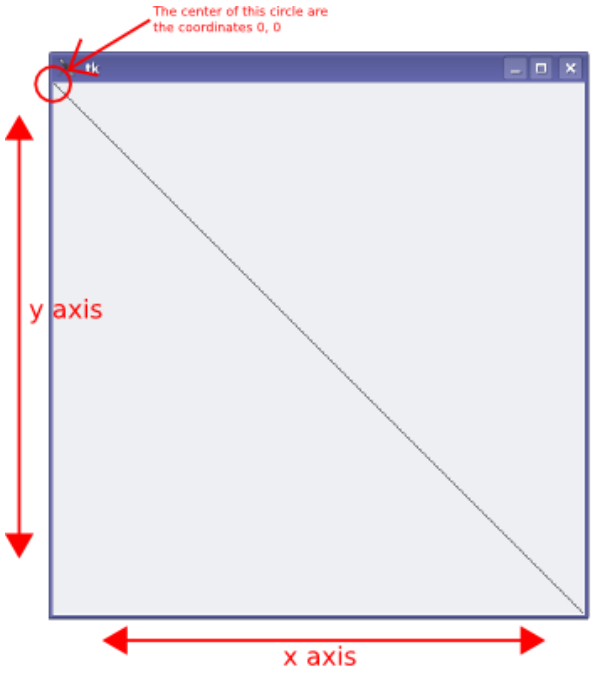


Abbildung 9.1: Leinwand mit x- und y-Achse

```
>>> from tkinter import *
>>> tk = Tk()
>>> leinwand = Canvas(tk, width=500, height=500)
>>> leinwand.pack()
>>> leinwand.create_line(0, 0, 500, 500)
```

Hätte die Schildkröte diese Linie zeichnen sollen, hätte das so ausgeschaut:

```
>>> import turtle
>>> turtle.setup(width=500, height=500)
>>> stift = turtle.Pen()
>>> stift.up()
>>> stift.goto(-250,250)
>>> stift.down()
>>> stift.goto(500,-500)
```

Wie du siehst, ist der tkinter-Code schon eine Verbesserung, da er kürzer und einfacher ist. Und das canvas-Objekt hat auch sonst noch einige nützliche Methoden, die wir mit dem Leinwand-Objekt verwenden können.

9.3 Rechtecke

Mit der Schildkröte haben wir Rechtecke und Quadrate gezeichnet, indem die Schildkröte immer eine Seite entlanggewandert ist, sich nach rechts umgedreht hat und die nächste Seite gezeichnet hat. Mit tkinter ist es viel einfacher ein Rechteck zu zeichnen—du brauchst nur die Koordinaten der Ecken zu kennen.

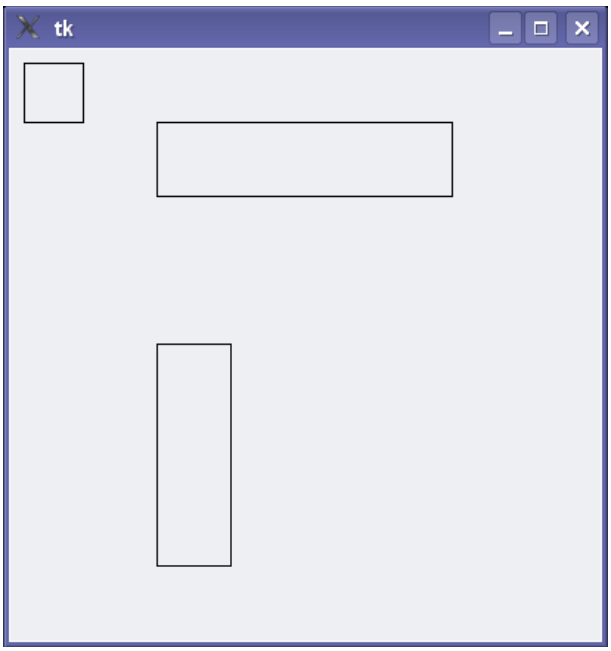


Abbildung 9.2: Rechtecke mit tkinter gezeichnet.

```
>>> from tkinter import *
>>> tk = Tk()
>>> leinwand = Canvas(tk, width=400,height=400)
>>> leinwand.pack()
>>> leinwand.create_rectangle(10, 10, 50, 50)
1
```

In diesem Beispiel erzeugen wir eine Leinwand, die 400 Pixel breit und 400 Pixel hoch ist. Danach zeichnen wir ein Quadrat im oberen linken Bereich. Du fragst dich vielleicht, was diese Nummer ist, die nach dem Zeichnen des Quadrates erschienen ist, als du leinwand.create_rectangle eingegeben hast. Das ist die identifizierende Nummer für das eben gezeichnete Gebilde (egal, ob das eine Linie, ein Quadrat oder Kreis ist). Dazu kommen wir etwas später.

Die Parameter, die du an create_rectangle übergibst, sind folgende: linke obere x-Position und linke obere y-Position. Und danach die rechte untere x- und y-Position. Um uns diese Tipparbeit zu sparen werden wir ab jetzt x1, y1 für oben links und x2, y2 für unten rechts schreiben. Ein größeres Rechteck gibt es also wenn x2 größer wird.

```
>>> leinwand.create_rectangle(100, 100, 300, 50)
```

Oder wenn y2 größer wird:

```
>>> leinwand.create_rectangle(100, 200, 150, 350)
```

Mit diesem Beispiel sagt man Python soviel wie: gehe 100 Pixel nach rechts und 200 Pixel nach unten. Dann zeichne ein Rechteck, das 350 Pixel breit und 150 Pixel hoch ist. Nach allen Beispielen solltest du so etwas wie in Abbildung 9.2 sehen.

Lass uns nun die Leinwand mit Rechtecken füllen, die alle eine eigene Farbe haben. Das können wir mit dem random machen. Zuerst importieren wir das Zufallsmodul (random):

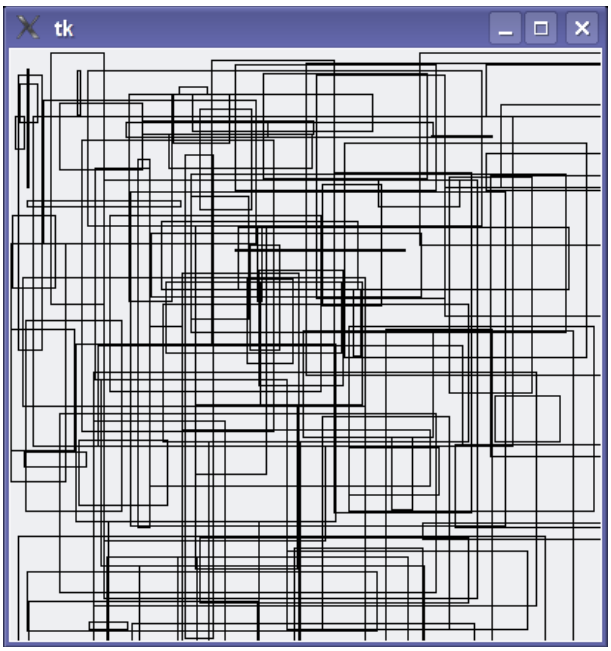


Abbildung 9.3: Rechtecke, wohin man schaut.

```
>>> import random
```

Danach verwenden wir eine Funktion, die Zufallszahlen für die Koordinaten verwendet:

```
>>> def zufaelliges_rechteck(breite, hoehe):
...     x1 = random.randrange(breite)
...     y1 = random.randrange(hoehe)
...     x2 = random.randrange(x1 + random.randrange(breite))
...     y2 = random.randrange(y1 + random.randrange(hoehe))
...     leinwand.create_rectangle(x1, y1, x2, y2)
```

In den ersten zwei Zeilen erzeugen wir Zufallsvariablen für die x- und y-Koordinaten der oberen linken Ecke. Die randrange-Funktion schränkt die Zahlen ein die zurückkommen können. Ein randrang(10) gibt zum Beispiel die Zahlen zwischen 0 und 9 zurück. Die nächsten zwei Zeilen erzeugen Variablen für die Koordinaten der rechten unteren Ecke. Zu guter Letzt wird mit der random_rectangle Funktion das Rechteck auf der Leinwand gezeichnet.

```
>>> zufaelliges_rechteck(400, 400)
```

Oder gleich eine Menge von Rechtecken in einer Schleife erzeugen:

```
>>> for x in range(0, 100):
...     zufaelliges_rechteck(400, 400)
```

Da kommt zwar eine große Unordnung raus (Abbildung 9.3), aber das ist trotzdem interessant.

Im letzten Kapitel haben wir die Schildkröte Linien in verschiedenen Farben zeichnen lassen. Das waren die Grundfarben rot, grün und blau. Mit tkinter kannst du auch Farben verwenden. Es ist aber ein wenig komplizierter. Lass uns zuerst unsere Funktion zum Reckteck Zeichnen anpassen, damit wir auch eine Farbe mitgeben können:

```
>>> def zufaelliges_rechteck(breite, hoehe, farbe):
...     x1 = random.randrange(breite)
...     y1 = random.randrange(hoehe)
...     x2 = random.randrange(x1 + random.randrange(breite))
...     y2 = random.randrange(y1 + random.randrange(hoehe))
...     leinwand.create_rectangle(x1, y1, x2, y2, fill=farbe)
```

Jetzt können wir der create_rectangle Funktion auch den Farbparameter 'fill' mitgeben. Probiere Folgendes aus:

```
>>> zufaelliges_rechteck(400, 400, 'green')
>>> zufaelliges_rechteck(400, 400, 'red')
>>> zufaelliges_rechteck(400, 400, 'blue')
>>> zufaelliges_rechteck(400, 400, 'orange')
>>> zufaelliges_rechteck(400, 400, 'yellow')
>>> zufaelliges_rechteck(400, 400, 'pink')
>>> zufaelliges_rechteck(400, 400, 'purple')
>>> zufaelliges_rechteck(400, 400, 'violet')
>>> zufaelliges_rechteck(400, 400, 'magenta')
>>> zufaelliges_rechteck(400, 400, 'cyan')
```

Die meisten Farben müssten so funktionieren. Aber manche könnten auch eine Fehlermeldung zurückgeben, je nachdem, ob du Windows, Mac OS X oder Linux verwendest. So weit, so gut. Aber was ist mit der goldenen Farbe? Mit der Schildkröte haben wir Gold aus 100% Rot und 85% Grün erzeugt. Mit tkinter könnten wir Gold folgendermaßen erzeugen:

```
>>> zufaelliges_rechteck(400, 400, '#ffd800')
```

Was natürlich eine komische Art ist um Farben zu erzeugen. 'ffd800' ist eine Hexadezimalzahl und eine andere Art um Zahlen auszudrücken. Die Erklärung von Hexadezimalzahlen würde jetzt zu weit führen, aber in der Zwischenzeit kannst du folgende Funktion verwenden um hexadezimale Zahlen zu erzeugen:

```
>>> def hexfarbe(rot, gruen, blau):
...     rot = 255*(rot/100.0)
...     gruen = 255*(gruen/100.0)
...     blau = 255*(blau/100.0)
...     return '#%02x%02x%02x' % (rot, gruen, blau)
```

Die hexfarbe-Funktion mit 100% Rot, 85% grün und 0% Blau aufzurufen, gibt den hexadezimalen Wert für die Farbe Gold zurück:

```
>>> print(hexfarbe(100, 85, 0))
#ffd800
```

Und ein helles Lila aus 98% Rot, 1% Rot und 77% blau ist:

```
>>> print(hexfarbe(98, 1, 77))
#f902c4
```

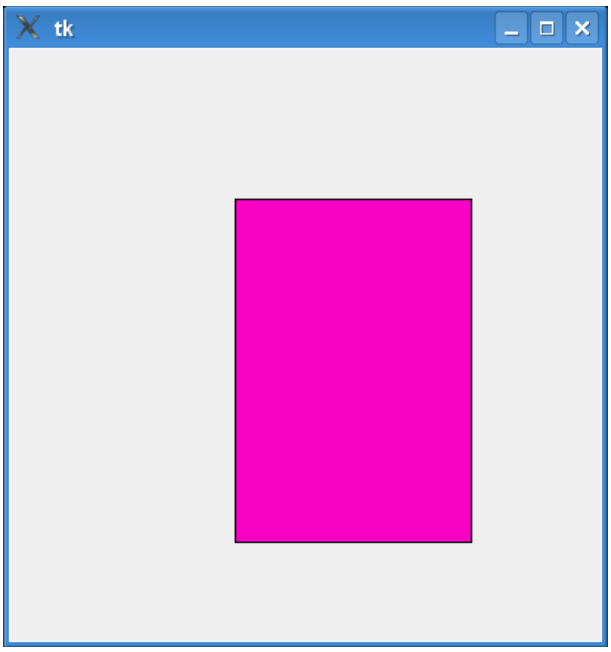


Abbildung 9.4: Ein pinkes Rechteck.

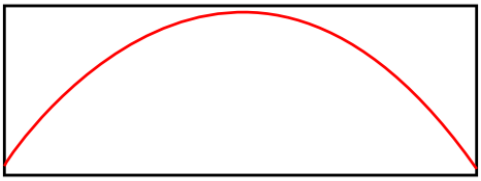


Abbildung 9.5: Ein Bogen innerhalb eines Rechtecks.

Du kannst das auch mit der zufaelliges_rechteck Funktion von vorher kombinieren:

```
>>> zufaelliges_rechteck(400, 400, hexfarbe(98, 1, 77))
```

9.4 Bögen

Um mit tkinter einen Bogen zu zeichnen, fängt man mit einem Rechteck an. Und in dieses Rechteck wird dann der Bogen reingezeichnet (siehe Abbildung 9.5). Der Code dazu könnte so ausschauen:

```
leinwand.create_arc(10, 10, 200, 100, extent=180, style=ARC)
```

Somit ist die obere linke Ecke auf den Koordinaten 10, 10 (also 10 nach rechts und 10 nach unten) und die rechte untere Ecke auf den Koordinaten 200, 100 (200 nach rechts und 100 nach unten). Der nächste Parameter (ein **benannter** Parameter), 'extend', wird benutzt um den Kreis näher in Grad zu beschreiben. Dabei entsprechen 359 Grad einem vollen Kreis und 180 einem halben. Probiere die nächsten Zeilen aus, um zu sehen, wie sich die Parameter auswirken (das Ergebnis siehst du in Abbildung 9.6):

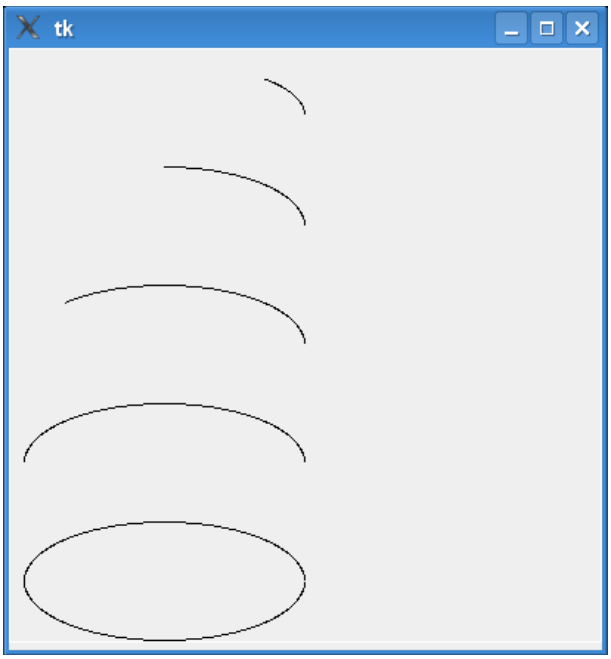


Abbildung 9.6: Verschiedene Varianten von Bögen.

```
>>> leinwand.create_arc(10,  10, 200,  80, extent= 45, style=ARC)
>>> leinwand.create_arc(10,  80, 200, 160, extent= 90, style=ARC)
>>> leinwand.create_arc(10, 160, 200, 240, extent=135, style=ARC)
>>> leinwand.create_arc(10, 240, 200, 320, extent=180, style=ARC)
>>> leinwand.create_arc(10, 320, 200, 400, extent=359, style=ARC)
```

9.5 Ellipsen

Mit dem letzten Befehl des vorigen Abschnitts kommt schon ein ovaler Kreis (eine Ellipse) raus. Aber auch die Funktion create_oval erzeugt eine Ellipse. Ähnlich zu den Bögen, wird der ovale Kreis innerhalb eines Rechtecks gezeichnet. Zum Beispiel mit folgendem Code:

```
>>> tk = Tk()
>>> leinwand = Canvas(tk, width=400,height=400)
>>> leinwand.pack()
>>> leinwand.create_oval(1, 1, 300, 200)
```

Dieser Code zeichnet die Ellipse ins imaginäre Rechteck mit den Eckpunkten 1,1 und 300,200. Zum verdeutlischen kannst du noch ein rotes Rechteck herum zeichnen (siehe Abbildung 9.7):

```
>>> leinwand.create_rectangle(1, 1, 300, 200, outline="#ff0000")
```

Um einen Kreis anstatt einer Ellipse zu zeichnen, muss das imaginäre Rechteck quadratisch sein (siehe Abbildung 9.8):

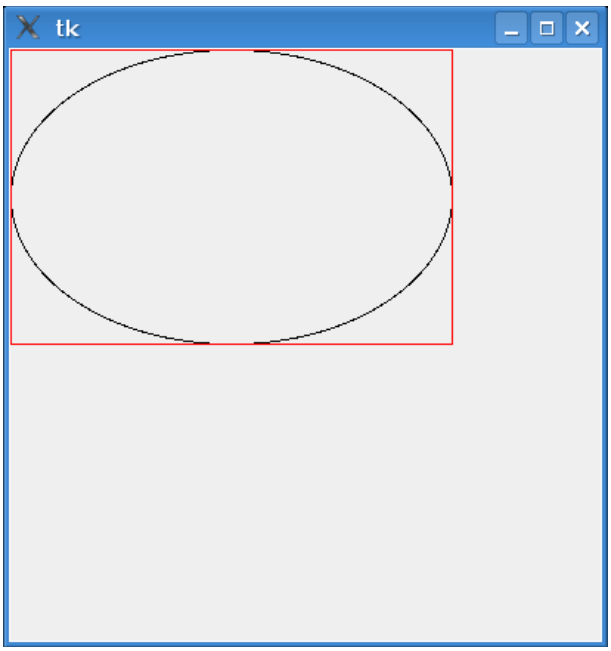


Abbildung 9.7: Die Ellipse innerhalb des roten Rechtecks.

```
>>> tk = Tk()
>>> leinwand = Canvas(tk, width=400,height=400)
>>> leinwand.pack()
>>> leinwand.create_oval(1, 1, 300, 300)
```

9.6 Polygone

Ein Polygon ist eine Form mit mehr als drei Seiten. Dreiecke, Vierecke, Fünfecke, Sechsecke und so weiter sind Beispiele dafür. Und die Formen können auch ganz ungleichmäßig sein. Für ein Dreieck musst du 3 Koordinatenpaare angeben (für jeden Eckpunkt).

```
>>> from tkinter import *
>>> tk = Tk()
>>> leinwand = Canvas(tk, width=400,height=400)
>>> leinwand.pack()
>>> leinwand.create_polygon(10,10,100,10,100,50,fill="",outline="black")
```

Das Ergebnis siehst du in Abbildung 9.9.
Zeichnen wir jetzt da noch ein unregelmäßiges Vieleck mit dem folgenden Code dazu.

```
>>> leinwand.create_polygon(200,10,240,30,120,100,140,120,fill="",outline="black")
```

Dreieck und Vieleck siehst du in Abbildung 9.10.

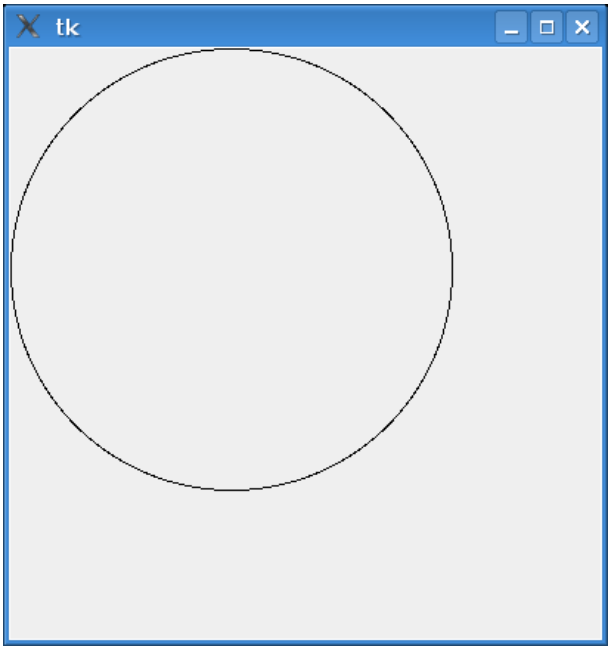


Abbildung 9.8: Ein einfacher Kreis.

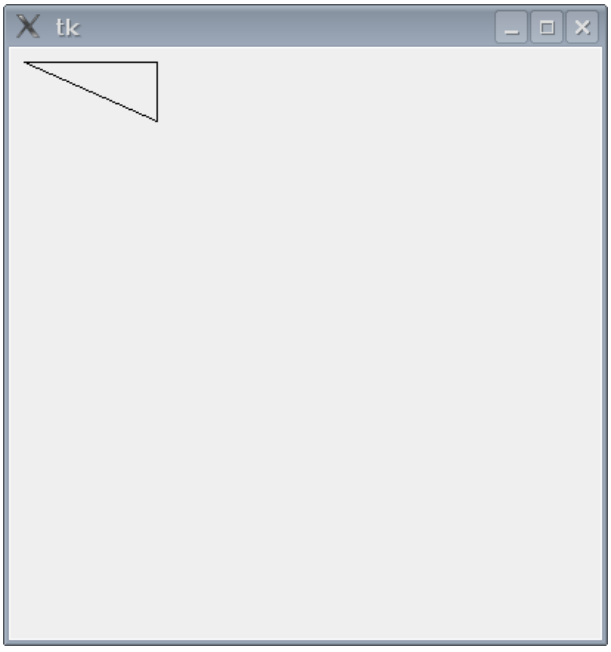


Abbildung 9.9: Ein einfaches Dreieck.

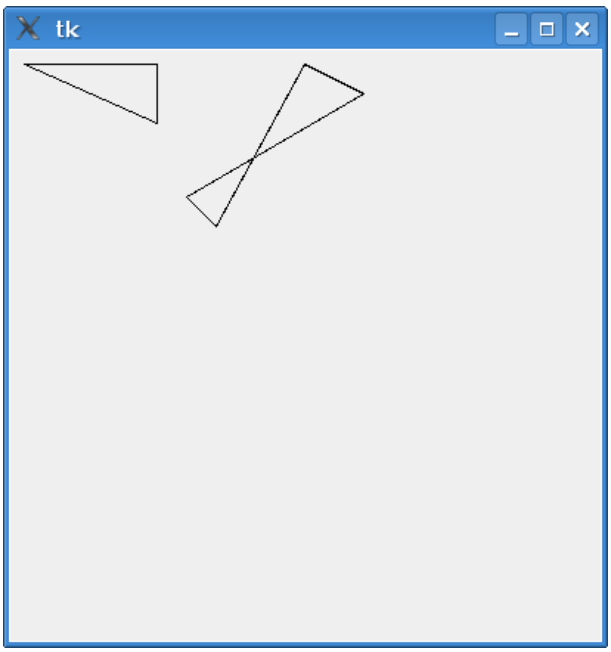


Abbildung 9.10: Das Dreieck und das unregelmäßige Vieleck.

9.7 Zeichne Bilder

Du kannst auch Bilder auf die Leinwand laden. Zuerst lädst du das Bild, dann benützt du die Funktion create_image und es erscheint auf der Leinwand. Probiere es aus:

```
1. >>> from tkinter import *
2. >>> tk = Tk()
3. >>> leinwand = Canvas(tk, width=400, height=400)
4. >>> leinwand.pack()
5. >>> mein_bild = PhotoImage(file='test.gif')
6. >>> canvas.create_image(0, 0, image=mein_bild, anchor=NW)
```

In den ersten 4 Zeilen generieren wir wieder die Leinwand wie in den vorigen Beispielen. In der fünften Zeile laden wir dann das Bild in die Variable mein_bild. Das Bild muss in einem Ordner liegen, auf den Python zugreifen kann. Normalerweise ist dies das Verzeichnis in dem die Python-Konsole gestartet wurde. Wo genau dieses Verzeichnis ist, kannst du leicht herausfinden. Importiere das os-Modul und verwende die getcwd()-Funktion:

```
>>> import os
>>> print(os.getcwd())
```

Unter Windows wirst du vermutlich etwas wie ‘c:
Python30’ bekommen.

Kopiere nun das Bild ins Verzeichnis und lade es mit der PhotoImage-Funktion (siehe Zeile 5). Dann benützt du die create_image-Funktion um das Bild auf der Leinwand zu befestigen. Das Ergebnis könnte wie Abbildung 9.11 aussehen.

Mit PhotoImage kannst du Dateien mit den Endungen .gif, .ppm und .pgm laden. Wenn du andere Bildformate laden willst (zum Beispiel speichern Digitalkameras Bilder meistens als .jpg), musst du

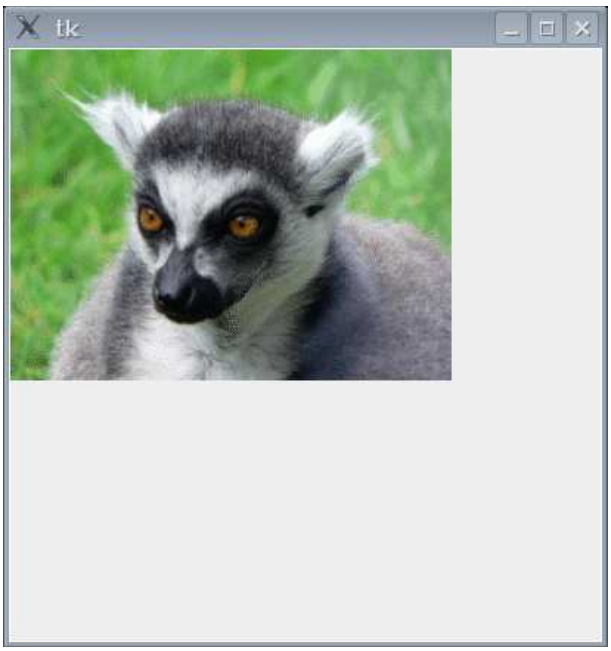


Abbildung 9.11: Ein Photo auf der Leinwand.

zuerst eine Python Erweiterung laden. Die Python-Bilder-Bibliothek [1] (Python Imaging Library auch PIL genannt) wäre ein Beispiel dafür.

9.8 Einfache Animationen

Bis jetzt haben wir immer statische Bilder—das sind Bilder, die sich nicht bewegen, gezeichnet. Was ist mit Animationen? Tk ist zwar nicht direkt darauf ausgelegt, aber für einfache kleine Sachen reicht es schon. Zum Beispiel können wir ein Dreieck füllen und dann über den Bildschirm wandern lassen:

```
1.  >>> import time
2.  >>> from tkinter import *
3.  >>> tk = Tk()
4.  >>> leinwand = Canvas(tk, width=400, height=400)
5.  >>> leinwand.pack()
6.  >>> leinwand.create_polygon(10, 10, 10, 60, 50, 35)
7.  1
8.  >>> for x in range(0, 40):
9.  ...     leinwand.move(1, 5, 0)
10. ...     tk.update()
11. ...     time.sleep(0.05)
```

Wenn du nun nach dem letzten Befehl Enter drückst, wird sich das Dreieck über den Bildschirm bewegen (und dann nach 40 x 5 = 200 Pixeln etwa in der Mitte stehenbleiben wie in Abbildung 9.12).

Wie funktioniert das?

Zeile 1 bis 5 haben wir schon öfters so verwendet—wir stellen die Leinwand in einer bestimmten Größe dar—in Zeile 6 wird das Dreieck gezeichnet (mit der create_polygon Funktion), und in Zeile 7

[1] Du findest die Python Bibliothek für Bilder unter http://www.pythonware.com/products/pil/index.htm

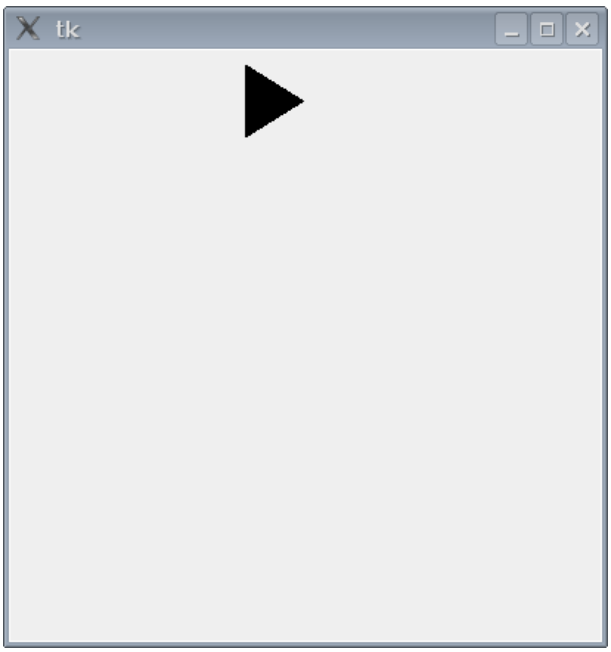


Abbildung 9.12: Das Dreieck wandert über den Bildschirm.

siehst du die eindeutige Nummer des Dreiecks (hier die Zahl 1), unter der das Dreieck aufrufbar ist. In Zeile 8 zählen wir in einer for-Schleife von 0 bis 40.

Im Block der Zeilen 9 bis 11 wird das Dreieck bewegt. Die move-Funktion bewegt ein definiertes Objekt, indem es die x- und y-Koordinaten des Objektes verändert. In Zeile 9 bewegen wir das Objekt mit der Nummer 1 (also das Dreieck) um 5 Pixel nach rechts und 0 Pixel nach unten. Wenn wir es wieder zurück nach links bewegen wollten, müssten wir leinwand.move(1, -5, 0) verwenden.

Die Funktion tk.update zeichnet das Dreieck in jedem Durchlauf der Schleife an die neue Position. Ansonsten würden wir die Bewegung nicht sehen. Und zu guter Letzt sagen wir Python in Zeile 11, dass Python eine zwanzigstel Sekunde (0.05) warten soll. Lass uns den Code so verändern, dass sich das Dreieck schräg nach unten bewegt. Mach zuerst die Leinwand zu, indem du auf die X-Schaltfläche am oberen Eck klickst. Probiere danach diesen Code:

```
>>> import time
>>> tk = Tk()
>>> leinwand = Canvas(tk, width=400, height=400)
>>> leinwand.pack()
>>> leinwand.create_polygon(10, 10, 10, 60, 50, 35)
1
>>> for x in range(0, 40):
...     leinwand.move(1, 5, 5)
...     tk.update()
...     time.sleep(0.05)
...
```

Abbildung 9.13 zeigt dir das Ergebnis der Bewegung. Wenn du das Dreieck wieder diogonal zurückbewegen willst, kannst du -5, -5 eingeben:

```
>>> import time
>>> for x in range(0, 40):
...     leinwand.move(1, -5, -5)
```

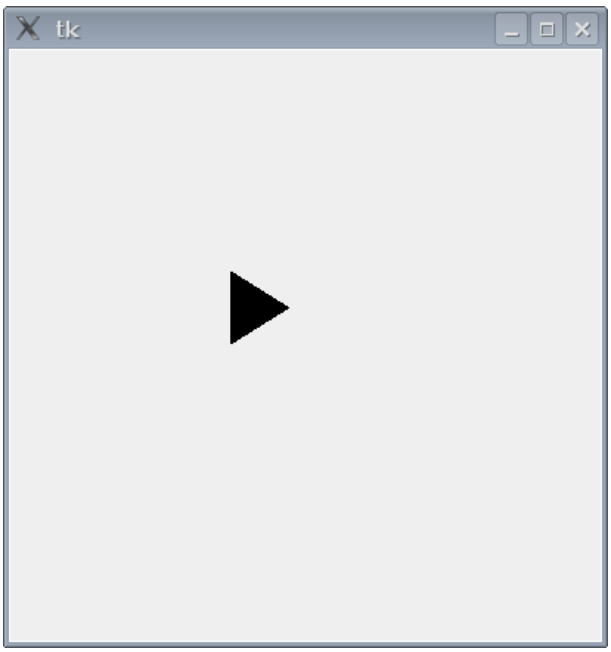


Abbildung 9.13: Das Dreieck bewegt sich schräg nach unten.

```
...     tk.update()
...     time.sleep(0.05)
```

9.9 Reagiere auf Ereignisse. . .

Wir können das Dreieck auch mit der Tastatur steuern, indem wir sogenannte *event bindings* verwenden. Ereignisse (engl. events) sind Dinge, die sich ereignen, während das Programm läuft. Also eine Mausbewegung oder ein Tastaturanschlag. Du kannst Tk so einstellen, dass es auf diese Ereignisse wartet und reagiert. Damit das also funktioniert, machen wir zuerst eine Funktion. Nehmen wir an, das Dreieck soll sich beim Drücken der Enter Taste bewegen. Also definieren wir eine Funktion um das Dreieck zu bewegen:

```
>>> def bewege_dreieck(ereignis):
...     leinwand.move(1, 5, 0)
```

Diese Funktion benötigt einen Parameter (ereignis), über den Tk Information an die Funktion sendet, was genau passiert ist. Zusätzlich sagen wir Tk, dass die Funktion immer bei einem bestimmten Ereignis aufgerufen werden soll, indem wir die bind_all Funktion mit der leinwand verbinden. Der Code schaut dann so aus:

```
>>> from tkinter import *
>>> tk = Tk()
>>> leinwand = Canvas(tk, width=400, height=400)
>>> leinwand.pack()
>>> leinwand.create_polygon(10, 10, 10, 60, 50, 35)
>>> def bewege_dreieck(ereignis):
...     leinwand.move(1, 5, 0)
```

```
...
>>> leinwand.bind_all('<KeyPress-Return>', bewege_dreieck)
```

Der erste Parameter der bind_all Funktion beschreibt die Ereignisse, auf die Tk warten soll. In diesem Fall ist es das Ereignis <KeyPress-Return> (also das Drücken der Enter-Taste). Dann sagen wir Tk, dass nach dem Drücken der Enter Taste die Funktion bewege_dreieck aufgerufen werden soll. Wenn du diesen Code ausführst, klicke in das Bild mit dem Dreieck und drücke die Enter Taste auf der Tastatur.

Wir könnten aber auch die Richtung der Bewegung mit den 4 Richtungstasten steuern. Zuerst ändern wir die bewege_dreieck-Funktion:

```
>>> def bewege_dreieck(ereignis):
...     if ereignis.keysym == 'Up':
...         leinwand.move(1, 0, -3)
...     elif ereignis.keysym == 'Down':
...         leinwand.move(1, 0, 3)
...     elif ereignis.keysym == 'Left':
...         leinwand.move(1, -3, 0)
...     else:
...         leinwand.move(1, 3, 0)
```

Das Objekt, welches von Tk an die Funktion bewege_dreieck weitergegeben wird, enthält eine Anzahl von *Eigenschaften*[2]. Eine von diesen Eigenschaften it keysym, was ein String ist, in dem die aktuell gedrückte Taste verpackt ist. Wenn keysym den String 'Up' enthält, rufen wir bewege_dreieck mit den Parametern (1, 0, -3) auf; wenn es den Wert 'Down' enthält, rufen wir die Funktion mit den Parametern (1, 0, 3) auf. Der erste Parameter ist immer die Nummer, die das Dreieck identifiziert, der zweite Parameter gibt an, wie weit nach rechts sich das Dreieck bewegen soll, und der dritte Parameter gibt an, wie weit nach unten. Zum Schluss müssen wir Tk noch sagen, auf alle 4 Tasten (links, rechts, oben, unten) zu reagieren. Also schaut der Code dann so aus:

```
>>> from tkinter import *
>>> tk = Tk()
>>> leinwand = Canvas(tk, width=400, height=400)
>>> leinwand.pack()
>>> leinwand.create_polygon(10, 10, 10, 60, 50, 35)
1
>>> def bewege_dreieck(ereignis):
...     if ereignis.keysym == 'Up':
...         leinwand.move(1, 0, -3)
...     elif ereignis.keysym == 'Down':
...         leinwand.move(1, 0, 3)
...     elif ereignis.keysym == 'Left':
...         leinwand.move(1, -3, 0)
...     else:
...         leinwand.move(1, 3, 0)
...
>>> leinwand.bind_all('<KeyPress-Up>', bewege_dreieck)
```

[2]Eigenschaften sind Werte, die etwas genauer beschreiben. Eine Eigenschaft von einem Auto ist zum Beispiel, dass es Räder hat. Eine Eigenschaft hat in Python einen Namen und einen Wert.

```
>>> leinwand.bind_all('<KeyPress-Down>', bewege_dreieck)
>>> leinwand.bind_all('<KeyPress-Left>', bewege_dreieck)
>>> leinwand.bind_all('<KeyPress-Right>', bewege_dreieck)
```

Mit diesem Codebeispiel bewegt sich das Dreieck nun in die Richtung der gedrückten Pfeiltaste.

10 Wie geht's jetzt weiter

Gratulation! Du hast es bis hierher geschafft.

Hoffentlich hast du ein paar grundlegende Konzepte gelernt, die dir auch das Lernen von anderen Programmiersprachen leichter machen werden. Während Python eine tolle Programmiersprache ist, ist Python nicht *immer* das beste Werkzeug für jede Aufgabe. Schau dir immer wieder mal andere Sprachen und Möglichkeiten an, deinen Computer zu programmieren.

Wenn dich Spieleprogrammierung ganz besonders interessiert, dann könntest du vielleicht einen Blick auf BlitzBasic (www.blitzbasic.com) werfen. Oder vielleicht auf Flash, mit dem viele Internetseiten Animationen und Spiele machen—wie zum Beispiel die Nickelodeon-Seite www.nick.com.

Falls du weiterhin Python für die Spieleentwicklung verwenden willst, ist die Seite www.pygame.org ein guter Tipp.

Und wenn du einfach mehr über Python erfahren und lernen willst (die fortgeschrittenen Programmierthemen), könntest du das frei verfügbare Python Tutorial auf (http://tutorial.pocoo.org/ durchlesen oder dich auf der deutschsprachigen Python Seite http://wiki.python.de umsehen.

Viel Glück, und habe Spaß beim Programmieren.

A

Python Schlüsselworte

Schlüsselwörter in Python (oder anderen Programmiersprachen) sind wichtige Wörter, die Python selber verwendet. Wenn du diese Schlüsselwörter als Variablen verwenden willst, oder sie falsch benutzt, wirst du komische (manchmal lustige, manchmal verwirrende) Fehlermedlungen zurück bekommen. Alle Python Schlüsselwörter mit einer kurzen Erklärung folgen in diesem Kapitel.

and

Das Schlüsselwort **and** wird verwendet um 2 Ausdrücke zu verbinden. Beide Ausdrücke müssen dann wahr sein. Zum Beispiel:

```
if alter > 10 and alter < 20
```

Das bedeutet, dass die Variable alter größer als 10 und kleiner als 20 sein muss.

as

Mit dem Schlüsselwort **as** kannst du ein importiertes Modul unter einem anderen Namen ansprechen. Wenn du zum Beispiel ein Modul mit diesem Namen hättest:

```
ich_bin_ein_python_modul_mit_langem_namen
```

Das wäre sehr lästig, wenn du jedes Mal wenn du das Modul verwenden willst den ganzen Namen eintippen müsstest:

```
>>> import ich_bin_ein_python_modul_mit_langem_namen
>>>
>>> ich_bin_ein_python_modul_mit_langem_namen.mach_was()
Ich habe was gemacht.
>>> ich_bin_ein_python_modul_mit_langem_namen.mach_was_anderes()
Ich hab was anderes gemacht!
```

Stattdessen kannst du dem Modul beim Importieren einen neuen kurzen Namen geben (wie ein Spitzname):

```
>>> import ich_bin_ein_python_modul_mit_langem_namen as kurzer_name
>>>
>>> kurzer_name.mach_was()
Ich habe was gemacht.
>>> kurzer_name.mach_was_anderes()
Ich hab was anderes gemacht!
```

Meist wird das Schlüsselwort ‘as’ aber selten verwendet.

assert

Assert ist ein fortgeschrittenes Schlüsselwort. Damit geben Programmieren an, dass etwas wahr sein muss. Damit kannst du Fehler und Probleme in Code finden—was normalerweise in fortgeschrittenen Programmen genutzt wird.

break

Mit dem **break** Schlüsselwort stoppst du Code beim Ausführen. Zum Beispiel könntest du den Befehl innerhalb einer for-Schleife verwenden:

```
>>> alter = 25
>>> for x in range(1, 100):
...     print('zähle %s' % x)
...     if x == alter:
...         print('zählen zu Ende')
...         break
```

Wenn die Variable ‘alter’ auf 10 gesetzt ist, würde Folgendes ausgegeben werden:

```
zähle 1
zähle 2
zähle 3
zähle 4
zähle 5
zähle 6
zähle 7
zähle 8
zähle 9
zähle 10
zählen zu Ende
```

In Kapitel 5 kannst du dir nochmals die Details zu for-Schleifen anschauen.

class

Mit dem **class** Schlüsselwort definierst du eine Art von Objekt. Das ist schon für Fortgeschrittene, aber sehr nützlich wenn du größere Programme programmiert. Für dieses Buch ist es zu fortgeschritten.

del

Del ist eine Funktion um etwas zu löschen. Wenn du eine Liste mit Geburtstagswünschen hast und etwas nicht mehr haben willst, dann würdest du es durchstreichen und was Neues hinschreiben: Sagen wir mal, die Liste besteht aus einem ferngesteuerten Auti, einem neuen Fahrrad und einem Computerspiel. Die gleiche Liste in Pyhton wäre:

```
>>> was_ich_will = ['ferngesteuertes Auto', 'neues Fahrrad', 'Computerspiel']
```

Wir könntes das Computerspiel entfernen indem wir del verwenden und etwas Neues mit der Funktion append hinzufügen:

```
>>> del was_ich_will[2]
>>> was_ich_will.append('Spielzeug-Dino')
```

Somit beinhaltet die neue Liste:

```
>>> print(was_ich_will)
['ferngesteuertes Auto', 'neues Fahrrad', 'Spielzeug-Dino']
```

In Kapitel 2 gibts Details zu Listen.

elif

elif wird als Teil von if-Bedingungen verwendet. Schau bei **if** weiter unten nach…

else

else ist auch ein Teil von if-Bedingungen. Schau weiter unten bei **if** nach…

except

Ein anderes Schlüsselwort um mit Fehlern in Code umzugehen. Das ist für kompliziertere Programme und zu fortgeschritten für dieses Buch.

exec

exec ist eine spezielle Funktion um einen String wie Python Code zu behandeln. Zum Beispiel kannst du eine Variable mit diesem Wert erzeugen:

```
>>> meine_variable = 'Hallo, wie gehts?'
```

Und den Inhalt ausgeben:

```
>>> print(meine_variable)
Hallo, wie gehts?
```

Aber du könntest auch Python Code in die String Variable packen:

```
>>> meine_variable = 'print("Hallo, wie gehts?")'
```

Und mit **exec** verpackst du den String in ein mini Python Programm und führst es aus:

```
>>> exec(meine_variable)
Hallo, wie gehts?
```

Das ist vielleicht eine komische Art Code auszuführen, aber wenn du es einmal brauchst, wirst du es zu schätzen wissen. Genau wie assert ist es ein Schlüsselwort für fortgeschrittene Programmierer und größere Programme.

finally

finally heißt auf Deutsch soviel wie 'endlich' oder 'zuletzt'. Mit diesem fortgeschrittenen Schlüsselwort wird nachdem ein Fehler aufgetreten ist, ein definierter Code ausgeführt, der wieder etwas Ordnung schafft.

for

Mit **for** werden for-Schleifen erzeugt. Zum Beispiel:

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

Diese for-Schleife führt den Codeblock (den Print Befehl) 5 mal aus und erzeugt folgende Ausgabe:

```
x ist 0
x ist 1
x ist 2
x ist 3
x ist 4
```

from

Beim Importieren von einem Modul, kannst du auch nur die Teile importieren, die du benötigst. Dazu verwendest du das **from** Schlüsselwort. Das turtle Modul hat zum Beispiel die Funktionen Pen(), mit dem das Pen Objekt erzeugt wird (eine Leinwand auf der die Schildkröte sich bewegen kann)—du kannst nun das gesamte turtle Modul importieren und dann die Pen Funktion verwenden:

```
>>> import turtle
>>> schildkroete = turtle.Pen()
```

Oder du importierst nur die Pen Funktion aus dem Modul und verwendest die Funktion direkt.

```
>>> from turtle import Pen
>>> schildkroete = Pen()
```

Das heißt natürlich, dass du nur diese Funktionen verwenden kannst, die du aus dem Modul importiert hast. Das Modul ‘time’ hat beispielsweise die Funktionen ‘localtime’ und ‘gmtime’. Wenn wir localtime importieren und dann gmtime verwenden wollen, wird es einen Fehler geben:

```
>>> from time import localtime
>>> print(localtime())
time.struct_time(tm_year=2010, tm_mon=1, tm_mday=16, tm_hour=16,
tm_min=45, tm_sec=14, tm_wday=5, tm_yday=16, tm_isdst=0)
```

Das funktioniert gut, aber:

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

Python sagt uns “‘gmtime’ is not defined”, also kann Python die Funktion gmtime. . . noch nicht finden. Wenn du mehrere Funktionen aus einem Modul verwenden willst und du sie nicht mit dem vollen Modul Namen ansprechen willst (das heißt time.localtime und time.gmtime und so weiter), kannst du alles auf einmal importieren, indem du das Stern Symbol (*) verwendest:

```
>>> from time import *
>>> print(localtime())
time.struct_time(tm_year=2010, tm_mon=1, tm_mday=16, tm_hour=17,
tm_min=39, tm_sec=15, tm_wday=5, tm_yday=16, tm_isdst=0)
>>> print(gmtime())
time.struct_time(tm_year=2010, tm_mon=1, tm_mday=16, tm_hour=16,
tm_min=39, tm_sec=37, tm_wday=5, tm_yday=16, tm_isdst=0)
```

Somit haben wir alle Funktionen aus dem time Modul importiert und können die Funktion direkt mit dem Namen der Funktion ansprechen.

global

In Kapitel 6 haben wir über *scope* geredet. Scope bezeichnet die ‘Sichtbarkeit’ von einer Variable. Wenn eine Variable ausserhalb einer Funktion definiert wurde, kann sie normalerweise innerhalb einer Funktion gesehen werden. Variablen, die innerhalb von Funktionen definiert werden, sind aber gewöhnlich **ausserhalb** der Funktion unsichtbar.

Mit dem global Schlüsselwort gibt es jedoch eine Ausnahme. Variablen die als global definiert werden, können überall gesehen werden. Wenn die Python Konsole so etwas wie eine kleine Erdkugel ist, dann heißt global so viel wie weltweit sichtbar:

```
>>> def test():
...     global a
...     a = 1
...     b = 2
```

Was wird passieren, wenn du nach dem Ausführen von der Funktion test(), print(a) und print(b) eintippst? Das Erste wird funktionieren, das Zweite wird einen Fehler erzeugen:

```
>>> test()
>>> print(a)
1
>>> print(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

Die Variable a ist global (sichtbar von überall auf der 'Erde'), aber b ist nur innerhalb der Funktion sichtbar. Wichtig zum merken ist, dass du global verwenden musst, bevor du der globalen Variable einen Wert zuweist.

if

Mit dem if Schlüsselwort wird eine Entscheidung über irgendwas gefällt—was manchmal mit den Schlüsselworten else und elif (else if) kombiniert wird. Eine if-Bedingung ist eine Art zu sagen, "wenn etwas wahr ist, dann mache etwas Bestimmtes". Zum Beispiel:

```
if spielzeug_preis > 1000:
    print('Diese Spielzeug ist zu teuer')
elif spielzeug_preis > 100:
    print('Dieses Spielzeug ist teuer')
else:
    print('Dieses Spielzeug will ich kaufen')
```

Diese if-Bedingung sagt, dass wenn ein Spielzeug über 1000 € kostet, es zu teuer ist; wenn es über 100 € kostet, dann ist es teuer und ansonsten sagt es "Dieses Spielzeug will ich kaufen". In Kapiel 4 gibt es mehr Informationen zu if-Bedingungen.

import

Mit dem **import** Schlüsselwort lädt Python ein Modul damit du es verwenden kannst. Zum Beispiel:

```
>>> import sys
```

Mit diesem Code gibst du Python Bescheid, dass du das Modul sys verwenden willst.

in

in wird in Ausdrücken verwendet, um herauszufinden ob ein Eintrag sich innerhalb einer Liste befindet. Um zu überprüfen, ob die Nummer 1 in einer Liste von Zahlen dabei ist, kannst du Folgendes eingeben:

```
>>> if 1 in [1,2,3,4]:
...     print('Die Nummer 1 ist in der Liste']
...
Die Nummer 1 ist in der Liste
```

Oder um zu überprüfen ob Salat auf der Einkaufsliste steht:

```
>>> einkaufsliste = [ 'Milch', 'Eier', 'Käse']
>>> if 'Salat' in einkaufsliste:
...     print('Salat ist auf der Einkaufsliste')
... else:
...     print('Salat fehlt auf der Einkaufsliste')
...
Salat fehlt auf der Einkaufsliste
```

is

Das Schlüsselwort **is** funktioniert so ähnlich wie der ist gleich-Operator (==) mit dem Dinge verglichen werden (10 == 10 ergibt wahr wogegen 10 == 11 falsch ergibt). Es gibt aber einen fundamentalen Unterschied zwischen **is** und ==. Beim Vergleich von Dingen kommt bei == manchmal wahr zurück und bei is falsch (auch wenn du glaubst die Dinge sind gleich).

Das ist etwas von den extrem fortgeschrittenen Konzepten und am Anfang enorm verwirrend. Lass uns am Anfang also immer == verwenden.

lambda

Dies ist ein anderes Schlüsselwort für Fortgeschrittene. In Wirklichkeit ist es so kompliziert, dass sogar ein Erklärungsversuch diese Buch zerstören würde.

Am besten wir reden gar nicht darüber.

not

Wenn etwas wahr ist, macht das **not** Schlüsselwort es falsch. Wenn wir zum Beispiel die Variable x auf true (wahr) setzen…

```
>>> x = True
```

… und dann den Wert der Variable unter Verwendung von not ausgeben:

```
>>> print(not x)
False
```

Das schaut nicht besonders nützlich aus, bis du anfängst dieses Schlüsselwort in if-Bedingungen zu verwenden. Wenn du 10 Jahre alt wärst und das wichtigste Alter für dich ist 10, dann willst du nicht alle anderen Zahlen folgendermaßen benennen:

> "1 ist kein wichtiges Alter" "2 ist kein wichtiges Alter" "3 ist kein wichtiges Alter" "4 ist kein wichtiges Alter" … … "50 ist kein wichtiges alter"

Und so weiter.

Mit einer if-Bedingung könnten wir das folgendermaßen schreiben. . .

```
if alter == 1:
    print("1 ist kein wichtiges Alter")
elif alter == 2:
    print("2 ist kein wichtiges Alter")
elif alter == 3:
    print("3 ist kein wichtiges Alter")
elif alter == 4:
    print("4 ist kein wichtiges Alter")
```

. . . und so weiter. Einfacher würde es so ausschauen:

```
if alter < 10 or alter > 10:
    print("%s ist kein wichtiges Alter" % alter)
```

Aber der einfachste Weg diese if-Bedingung zu formulieren ist **not** zu verwenden.

```
if not alter == 10:
    print("%s ist kein wichtiges Alter" % alter)
```

Was eine andere Art um "wenn das Alter nicht 10 ist" zu sagen.

or

Mit dem **or** Schlüsselwort werden zwei Ausdrücke miteinander verbunden (wie bei einer if-Bedingung). Damit wird festgestellt, ob einer der Ausdrücke wahr ist. Zum Beispiel:

```
>>> if vorname == 'Ruth' or vorname == 'Robert':
...     print('Die Rapps')
... elif vorname == 'Barbara' or vorname == 'Bert':
...     print('Die Bergers')
```

Wenn also die Variable vorname 'Ruth' oder 'Robert' enthält, dann kommt 'Die Rapps' raus. Wenn die Variable 'Barbara' oder 'Bert' enthält, dann wird 'Die Bergers' ausgegeben.

pass

Manchmal, beim Schreiben von Programmen willst du nur Teile schreiben um Dinge auszuprobieren. Das Problem bei der Sache ist, dass du if-Bedingungen nicht halbfertig lassen kannst. Der Codeblock der bei der if-Bedingung ausgeführt werden soll, muss definiert werden. Bei for-Schleifen ist es genau so. Zum Beispiel:

```
>>> if alter > 10:
...     print('Älter als 10')
```

Der Code von oben wird funktionieren. Aber wenn du Folgendes eintippst:

```
>>> if alter > 10:
...
```

Wirst du folgenden Fehler in der Konsole erhalten:

```
  File "<stdin>", line 2
    ^
IndentationError: expected an indented block
```

Dies ist die Fehlermeldung, die Python ausgibt, wenn nach einer Bedingung ein Code-Block fehlt.

In diesen Fällen kannst du mit **pass** arbeiten. Somit kannst du zum Beispiel eine for-Schleife schreiben, mit einer if-Bedingung drin, ohne aber alles auszuformulieren. Später kannst du dich dann entscheiden ob du ein break, ein print oder was Anderes in der if-Bedingung verwenden willst. Vorerst verwendest du **pass** und der Code wird funktionieren (auch wenn er noch nicht das macht, was er soll). Das Codebeispiel:

```
>>> for x in range(1,7):
...     print('x ist %s' % x)
...     if x == 5:
...         pass
```

wird folgendes ausgeben:

```
x ist 1
x ist 2
x ist 3
x ist 4
x ist 5
x ist 6
```

Später kannst du den Block der if-Bedingung ausbauen (indem du **pass** mit dem richtigen Code austauscht).

print

Mit dem **print** Schlüsselwort kannst du etwas auf der Python Konsole ausgeben. Zum Beispiel einen String, eine Zahl oder eine Variable:

```
print('Hallo, wie gehts')
print(10)
print(x)
```

raise

Das ist wieder ein fortgeschrittenes Schlüsselwort. **raise** wird benutzt um einen Fehler zu erzeugen—was vielleicht unsinnig erscheint, aber in komplexen Programmen sehr nützlich ist.

return

return wird benutzt um einen Wert von einer Funktion zurückzugeben. Zum Beispiel könntest du eine Funktion schreiben, die den Betrag des gesparten Geldes zurückgibt.

```
>>> def mein_geld():
...     return geld_menge
```

Wenn du diese Funktion ausführst, kannst du den zurückgegebenen Wert einer anderen Variable zuweisen:

```
>>> geld = mein_geld()
```

oder ausgeben:

```
>>> print(mein_geld())
```

try

Mit **try** fängt ein Block an, der mit **except** und/oder **finally** aufhört. Zusammen genommen sorgen **try/except/finally** dafür, dass im Fehlerfall nützliche verständliche Fehlermeldungen ausgegeben werden, statt den normalen Python Fehlermeldungen.

while

Ähnlich wie ein for-loop, macht eine **while**-Schleife mehrere Durchgänge. Während aber bei der for-Schleife im vorheinein die Zahl der Durchläufe festgelegt ist, läuft eine while-Schleife solange eine Bedingung wahr ist. Da musst du auch aufpassen. Denn wenn eine Bedingung immer Wahr ist, wird der Loop immer weiterlaufen und nie aufhören (das nennt man auch Endlos-Schleife). Zum Beispiel:

```
>>> x = 1
>>> while x == 1:
...     print('Hallo')
```

Wenn du den Code oben ausführst, wird der Code in der Schleife ewig ausgeführt. Oder bis du die Pyton Konsole mit **STRG+Z** oder **STRG+C** beendest. Der folgende Code hat dieses Problem nicht:

```
>>> x = 1
>>> while x < 10:
...     print('Hallo')
...     x = x + 1
```

Diesmal wird ‘Hallo’ 9 mal ausgegeben (und jedesmal wird zur Variable x auch 1 hinzugezählt), solange x kleiner als 10 ist. Das ist ähnlich einer for-Schleife, passt aber in bestimmten Situationen besser.

with

with ist zu fortgeschritten für dieses Buch.

yield

Auch **yield** wirst du erst später benötigen, wenn du mehr mit Python programmierst.

B

Eingebaute Funktionen

Python hat schon eine Menge von eingebauten Funktionen—Funktionen, die sofort benützt werden können, ohne zuerst den **import** Befehl auszuführen zu müssen. Einige dieser eingebauten Funktionen sind hier aufgelistet.

abs

Die **abs** Fuktion gibt den absoluten Wert einer Zahl zurück. Ein absoluter Wert ist nie negativ. Also ist der absolute Wert von 10 auch 10, aber von -20.5 ist der absolute Wert 20.5. Zum Beispiel:

```
>>> print(abs(10))
10
>>> print(abs(-20.5))
20.5
```

bool

Die Funktion **bool** gibt abhängig von den Parametern, entweder True (wahr) oder False (falsch) zurück. Für Zahlen gibt 0 ein False zurück, während alle anderen Zahlen True zurückgeben:

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

Für alle anderen Parameter erzeugt None ein False und alles andere ein True:

```
>>> print(bool(None))
False
>>> print(bool('a'))
True
```

dir

Die **dir** Funktion gibt zu einem Objekt eine Liste von Informationen zurück. Du kannst die **dir** Funktion mit Strings, Zahlen, Funktionen, Module, Objekte, Klassen—eigentlich fast allem verwenden. Manchmal wird die zurückgegebe Information nützlich sein und manchmal weniger. Wenn du die Funktion dir auf die Nummer 1 anwendest, kommt das zurück:

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

. . .eine ganze Menge von speziellen Funktionen. Wenn du die Funktion **dir** nun auf den String 'a' anwendest, dann wird Folgendes ausgegeben:

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

Und hier kannst du erkennen, dass es Funktionen wie capitalize gibt (den ersten Buchstaben in einen Großbuchstaben wandeln) . . .

```
>>> print('aaaaa'.capitalize())
Aaaaa
```

. . .isalnum wiederum ist eine Funktion, die True (wahr) zurückgibt, wenn der String alphanumerisch ist—also nur Buchstaben und Zahlen enthält. Dir Funktion dir ist nützlich, um schnell herauszufinden was du mit einer Variable so alles anstellen kannst.

eval

Der **eval** Funktion gibst du einen String als Parameter mit und führst die Funktion aus, als wäre es normaler Python Code. Das ist ähnlich zum **exec** Schlüsselwort, aber funktioniert leicht anders. Mit exec kannst du kleine Python Programme in einen String packen, aber eval erlaubt nur einfache Ausdrücke wie:

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

file

Mit der **file** Funktion öffnest du eine Datei und bekommst ein Datei Objekt mit eigenen Funktionen zurück. Du kannst dann den Inhalt der Datei auslesen oder die Größe und so weiter. Genaueres steht in Kapitel 7.

float

Die **float** Funktion verwandelt einen String in eine Fließkommazahl. Also eine Zahl mit einer Kommastelle. Die Zahl 10 ist zum Beispiel ein 'integer' (eine Ganzzahl), aber 10.0, 10.1, 10.253 und so weiter sind 'floats' (Fließkommazahlen). Das Umwandeln eines Strings in eine float geht so:

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

Du kannst auch die Kommastelle dem String mitgeben:

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

Auch eine normale Ganzzahl (integer) kann in eine Fließkommazahl verwandelt werden:

```
>>> float(200)
200.0
```

Und wenn du eine Fließkommazahl umwandeln willst, kommt natürlich die gleiche Fließkommazahl heraus:

```
>>> float(100.123)
100.123
```

Das Aufrufen der float Funktion ohne Argumente gibt 0.0 zurück.

int

Mit **int** kannst du eine Nummer oder einen String in eine Ganzzahl (oder Integer) verwandeln:

```
>>> int(12345.678)
12345
>>> int('12345')
12345
```

Diese Funktion funktioniert etwas anders als die **float** Funktion. Beim Umwandeln von einem String, der eine Fließkommazahl darstellt, gibt es diesen Fehler:

```
>>> int('12345.678')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12345.678'
```

Beim Aufrufen ohne Parameter wird 0 zurückgegeben.

len

Die **len** Funktion gibt die Länge von einem Objekt zurück. Wenn das Objekt ein String ist, dann bekommst du die Anzahl der Zeichen zurück:

```
>>> len('Das ist ein Test String')
23
```

Bei einer Liste oder einem Tupel bekommst du die Anzahl der Dinge zurück:

```
>>> meine_liste = [ 'a', 'b', 'c', 'd' ]
>>> print(len(meine_liste))
4
>>> mein_tupel = (1,2,3,4,5,6)
>>> print(len(mein_tupel))
6
```

Für map's bekommst du auch die Anzahl der Einträge zurück:

```
>>> meine_map = { 'a' : 100, 'b' : 200, 'c' : 300 }
>>> print(len(meine_map))
3
```

Vielleicht wirst du die len Funktion nützlich finden, wenn du sie mit Schleifen kombinierst. Wenn du zum Beispiel die Einträge einer Liste durchgehen willst, könntest du folgendes eingeben:

```
>>> meine_liste = [ 'a', 'b', 'c', 'd' ]
>>> for eintrag in meine_liste:
...     print(eintrag)
```

Damit würden alle Einträge der Liste ausgegeben (a, b, c, d)—aber wenn wir auch die Position des Eintrages ausgeben wollen? In diesem Fall könnten wir die Länge der Liste vorher bestimmen und dann durchzählen:

```
>>> meine_liste = [ 'a', 'b', 'c', 'd' ]
>>> laenge = len(meine_liste)
>>> for x in range(0, laenge):
...     print('Eintrag auf Position %s ist %s' % (x, meine_liste[x]))
...
Eintrag auf Position 0 ist a
Eintrag auf Position 1 ist b
Eintrag auf Position 2 ist c
Eintrag auf Position 3 ist d
```

Somit speichern wir die Länge der Liste in der Variable mit dem Namen ‘laenge’ und verwenden sie in der range Funktion um unsere Schleife zu erzeugen.

max

Die **max** Funktion gibt den größten Eintrag einer Liste, einem Tupel und sogar einem String zurück. Zum Beispiel:

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

Was auch funktioniert ist eine String, bei dem die einzelnen Einträge mit Kommas unterteilt sind:

```
>>> s = 'a,b,d,h,g'
>>> print(max(s))
h
```

Aber du kannst auch direkt die max Funktion mit den Werten aufrufen:

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

min

Die **min** Funktion funktioniert genau gleich wie die **max** Funktion, aber es gibt den kleinsten Eintrag zurück:

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

range

Die **range** Funktion wird hauptsächlich bei for-Schleifen verwendet. In Kapitel 5 haben wir dir range Funktion mit 2 Parametern verwendet. Wir können aber auch 3 Parameter verwenden. Hier nochmals ein Beispiel mit 2 Parametern:

```
>>> for x in range(0, 5):
...     print(x)
...
0
1
2
3
4
```

Was du sicher noch nicht gemerkt hast, ist, dass die range Funktion eigentlich ein Objekt zurückgibt (einen Iterator) mit dem die For-Schleife danach arbeitet. Diesen Iterator kannst du mit der list Funktion in eine Liste umwandeln um zu sehen, was da genau drin ist:

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

Du erhältst also eine Zahlenliste, die du auch woanders in deinem Programm verwenden kannst.

```
>>> meine_zahlenliste = list(range(0, 30))
>>> print(meine_zahlenliste)
[0, 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]
```

Der range Funktion kannst du auch ein drittes Argument mitgeben. Eine sogenannte Schrittweite (oder step auf englisch). Wenn du da nichts mitgibst, dann wird die Schrittweite 1 verwendet. Lass uns die Schrittweite 2 mitgeben und schauen was passiert:

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

Jeder Eintrag der Liste ist um 2 größer als der letzte. Wir können auch größere Schritte verwenden.

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

Somit bekommen wir eine Liste von 0 bis 500 (500 ist natürlich nicht dabei) und jede Zahl ist 50 größer als die Vorhergehende.

sum

Die **sum** Funkton summiert alle Einträge einer Liste und gibt das Ergebnis zurück.

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

>>> print(sum(meine_zahlenliste))
2250
```

C

Ein Paar Python Module

Für Python gibt es eine große Menge von verfügbaren Modulen, die viele nützliche Dinge tun können. Wenn du Details dazu nachlesen willst, bist du auf http://docs.python.org/3.1/modindex.html richtig. Diese Liste ist sehr lang und einige der Module sind auch kompliziert. Aber ein Paar der gebräuchlichsten werden gleich hier erklärt.

Das 'random' Modul

Wenn du jemals das Zahlen-Rate-Spiel gespielt hast, bei dem der Mitspieler eine Zahl zwischen 1 und 100 eraten muss, dann wirst du das 'random' Modul gut verwenden können. In diesem Modul sind viele Funktionen verpackt, die zufällige Zahlen zurückgeben. Ein wenig so, wie wenn du den Computer nach einer Zahl fragen würdest. Die interessantesten Funktionen im 'random' Modul sind randint, choice und shuffle. Die erste Funktion, random pickt eine Zahl zwischen der Start- und Endnummer heraus (zwischen 1 und 100, 100 und 1000, oder 1000 und 5000, und so weiter). Zum Beispiel:

```
>>> import random
>>> print(random.randint(1, 100))
58
>>> print(random.randint(100, 1000))
861
>>> print(random.randint(1000, 5000))
3795
```

Damit könnten wir schon ein einfaches und lästiges Ratespiel machen, indem wir eine Schleife verwenden:

```
import random
import sys
num = random.randint(1, 100)
print('Such dir eine Zahl zwischen 1 und 100 aus und gib sie ein:')
while True:
    chk = sys.stdin.readline()
    i = int(chk)
    if i == num:
        print('Du hast die Zahl erraten')
```

```
        break
    elif i < num:
        print('Probiere es höher')
    elif i > num:
        print('Probiere es niedriger')
```

Mit choice kannst du aus einer Liste einen zufälligen Eintrag aussuchen. Zum Beispiel:

```
>>> import random
>>> liste1 = [ 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' ]
>>> print(random.choice(liste1))
c
>>> liste2 = [ 'Eis', 'Pfannkuchen', 'Trüffel', 'Pudding' ]
>>> print(random.choice(liste2))
Pudding
```

Und wenn du zum Beispiel die Dinge mischen willst, kannst du shuffle verwenden. Damit werden die Inhalte der Liste neu gemischt und wieder gespeichert.

```
>>> import random
>>> liste1 = [ 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' ]
>>> liste2 = [ 'Eis', 'Pfannkuchen', 'Trüffel', 'Pudding' ]
>>> random.shuffle(liste1)
>>> print(liste1)
['h', 'e', 'a', 'b', 'c', 'g', 'f', 'd']
>>> random.shuffle(liste2)
>>> print(liste2)
['Trüffel', 'Pfannkuchen', 'Eis', 'Pudding']
```

Das ‘sys’ Modul

Im sys sind nützliche Funktionen des ‘systems’ enthalten. Diese Funktionen sind auch innerhalb von Python sehr wichtig. Die nützlichsten Funktionen sind: exit, stdin, stdout und version

Mit der exit Funktion kannst du auch die Python Konsole beenden. Wenn du zum Beispiel folgendes eintippst:

```
>>> import sys
>>> sys.exit()
```

Somit stoppt die Konsole. Das schaut in Windows, am Mac oder in Linux immer ein wenig verschieden aus—aber das Resultat ist das gleiche: die Python Konsole läuft nicht mehr.

Mit stdin haben wir schon in Kapitel 6 für Benutzereingaben verwendet. Ein kleines Beispiel schaut so aus:

```
>>> import sys
>>> meine_variable = sys.stdin.readline()
Das ist eine Testeingabe mit der Tastatur
>>> print(meine_variable)
Das ist eine Testeingabe mit der Tastatur
```

Mit stdout erreichst du das Gegenteil—du schreibst damit Nachrichten auf die Konsole. Es ist so ähnlich wie print, aber funktioniert mehr wie eine Datei, und deswegen ist es manchmal praktischer stdout statt print zu verwenden.

```
>>> import sys
>>> l = sys.stdout.write('Das ist ein Test')
Das ist ein Test>>>
```

Schon gemerkt wo der Promt (>>>) erscheint? Das ist kein Fehler, dass es am Ende der Nachricht ist. Anders als print springt nämlich stdout.write nicht automatisch auf die nächste Zeile. Damit stdout.write auch auf die nächste Zeile springt, musst du Folgendes eingeben:

```
>>> import sys
>>> l = sys.stdout.write('Das ist ein Test\n')
Das ist ein Test
>>>
```

stdout.write gibt auch die Anzahl der ausgegebenen Buchstaben aus–tippe print(l) um das Resultat zu sehen).
\n ist Teil einer sogenannten *escape* Sequenz für einen Zeilenumbruch (das gleiche wie du bei Drücken der Enter Taste bekommst). Wenn du einen String mit einem Zeilenumbruch in der Mitte erzeugen willst und die Enter Taste direkt eingibst, wird es aber einen Fehler geben:

```
>>> s = 'test test
  File "<stdin>", line 1
    s = 'test test
                ^
SyntaxError: EOL while scanning string literal
```

Stattdessen kannst du die Escape Sequenz für den Zeilenumbruch verwenden:

```
>>> s = 'test test\ntest'
```

Zu guter Letzt kannst du mit version die Version von Python überprüfen:

```
>>> import sys
>>> print(sys.version)
3.1.1+ (r311:74480, Nov  2 2009, 14:49:22)
[GCC 4.4.1]
```

Das ‘time’ Modul

Das Zeit Module (time) in Python enthält Funktionen um. . . ... natürlich um die Zeit darzustellen. Wenn du aber die offensichtlichste Funktion time aufrufst, wird das Ergebnis dich wahrscheinlich überraschen:

```
>>> import time
>>> print(time.time())
1264263605.43
```

time

Die Zahl die von time() zurückgegeben wird, entspricht der Anzahl von Sekunden die seit dem 1. Januar 1970 vergangen sind (seit 00:00 um exakt zu sein). Das klingt auf den ersten Blick nicht sehr nützlich, kann es aber sein. Wenn du zum Beispiel die Zeit, die dein Programm braucht, stoppen willst. Dann musst du dir nur die Startzeit deines Programmes merken und die Endzeit und dann vergleichen. Wie lange dauert zum Beispiel das Ausgeben von den Zahlen von 0 bis 100000? Packen wir das in eine Funktion und probieren es aus:

```
>>> def eine_menge_zahlen(max):
...     for x in range(0, max):
...         print(x)
```

Dann rufen wir die Funktion auf:

```
>>> eine_menge_zahlen(100000)
```

Wenn wir die Zeit wissen wollen, wie lange das dauert, können wir eine Funktion des time Modul verwenden:

```
>>> def eine_menge_zahlen(max):
...     t1 = time.time()
...     for x in range(0, max):
...         print(x)
...     t2 = time.time()
...     print('Es dauerte %s Sekunden' % (t2-t1))
```

Wenn wir es wieder aufrufen (diesmal mit einer Million):

```
>>> eine_menge_zahlen(1000000)
0
1
2
3
.
.
.
99997
99998
99999
Es dauerte 34.92557406425 Sekunden
```

Wie funktioniert das? Wenn wir das erste Mal die time()-Funktion aufrufen, speichern wir den Wert in der Variable t1. Danach werden in der Schleife alle Nummern ausgegeben. Danach rufen wir die time() Funktion wieder auf und speichern das Ergebnis in der Variable t2. Da es einige Sekunden dauerte um alle Zahlen auszugeben, ist der Wert in t2 höher (da ja schon mehr Sekunden seit dem 1.1.1970 vergangen sind). Wenn du dann also von der Endzeit t2 die Anfangszeit t1 abziehst, dann erhältst du die Zeit, die für die Ausgabe der Zahlen benötigt wurde.

Andere Funktionen, die im time Modul enthalten sind: asctime, ctime, localtime, sleep, strftime und strptime.

asctime

Die Funktion asctime nimmt ein Datum als Tupel (ein Tupel ist eine Liste mit Werten, die nicht geändert werden können) und wandelt sie in eine lesbare Form um. Du kannst die Funktion auch ohne Argumente aufrufen und es gibt das aktuelle Datum und die Zeit in einer lesbaren Form zurück:

```
>>> import time
>>> print(time.asctime())
Sat Jan 23 18:43:14 2010
```

Um die Funktion mit einem Argument aufzurufen, müssen wir erst ein Tupel mit den korrekten Werten für Datum und Zeit erzeugen. Lass uns Werte für das Tupel t erzeugen:

```
>>> t = (2010, 1, 23, 18, 45, 48, 5, 23, 0)
```

Die Werte in dieser Sequenz sind Jahr, Monat, Tag, Stunden, Minuten, Sekunden, Tag der Woche (0 ist Montag, 1 ist Dienstag, und so weiter, 6 ist Sonntag) dann der Tag des Jahres und zum Schluss ob Winterzeit oder Sommerzeit (0 ist Winterzeit). Wenn wir also asctime mit dem obigen Tupel aufrufen, bekommen wir:

```
>>> t = (2010, 1, 23, 18, 45, 48, 5, 23, 0)
>>> print(time.asctime(t))
Sat Jan 23 18:45:48 2010
```

Beim Tupel erstellen musst du vorsichtig sein. Ansonsten bekommst du ein falsches unsinniges Datum zurück, wie zum Beispiel:

```
>>> t = (2010, 1, 23, 18, 45, 48, 6, 23, 0)
>>> print(time.asctime(t))
Sun Jan 23 18:45:48 2010
```

Weil nun der Wert für 'Tag der Woche' auf 6 gestellt ist, denkt asctime nun, dass der 27. Januar 2010 ein Sonntag (Sunday) ist. Richtig ist Samstag.

ctime

Mit der ctime Funktion kannst du die Sekunden, die seit dem 1.1.1970 verstrichen sind in eine lesbare Form umwandeln:

```
>>> import time
>>> t = time.time()
>>> print(t)
1264269322.18
>>> print(time.ctime(t))
Sat Jan 23 18:55:22 2010
```

localtime

Ein Tupel kommt zurück, wenn du localtime verwendest:

```
>>> import time
>>> print(time.localtime())
time.struct_time(tm_year=2010, tm_mon=1, tm_mday=23, tm_hour=20,
tm_min=49, tm_sec=3, tm_wday=5, tm_yday=23, tm_isdst=0)
```

Und dieses Tupel kannst du wieder in asctime stecken:

```
>>> import time
>>> t = time.localtime()
>>> print(time.asctime(t))
Sat Jan 23 20:52:30 2010
```

sleep

Die Funktion sleep ist nützlich, wenn du dein Programm für eine bestimmte Zeit anhalten willst. Wenn du zum Beispiel jede Sekunde eine Zahl ausgeben willst, wäre diese Schleife etwas zu schnell:

```
>>> for x in range(1, 61):
...     print(x)
...
1
2
3
4
```

Alle Zahlen von 1 bis 60 würden sofort ausgegeben werden. Aber du kannst Python sagen, dass es nach jeder Zahl für eine Sekunde schlafen (sleep) soll.

```
>>> for x in range(1, 61):
...     print(x)
...     time.sleep(1)
...
```

Jetzt gibt es eine kurze Pause (1 Sekunde) zwischen den Zahlen. Einen Computer zu sagen, dass er warten soll, scheint vielleicht nicht sehr nützlich, aber es gibt Situationen wo es sehr wohl nützlich ist. Denke an deinen Wecker, wenn er in der Früh klingeln anfängt. Wenn du auf den Sleep- oder Snooze-Knopf drückst, kannst du 10 Minuten weiterschlafen, oder bis jemand dich zum Frühstück ruft. Die sleep Funktion ist manchmal genau so nützlich.

strftime

Mit der strftime Funktion kannst du die Art wie Datum und Zeit dargestellt werden beeinflussen. Du übergibst der strftime Funktion einen String und es kommt ein Datum/Zeit Tupel heraus. Schauen wir uns strftime zuerst an. Kurz vorher haben wir aus einem Tupel einen String erzeugt indem wir asctime verwendet haben:

```
>>> t = (2010, 1, 23, 18, 45, 48, 5, 23, 0)
>>> print(time.asctime(t))
Sat Jan 23 18:45:48 2010
```

Die meiste Zeit funktioniert das ganz gut. Aber was, wenn du die Art wie das Datum dargestellt ist, ändern willst? Wenn du nur das Datum ohne die Zeit haben willst. Das geht dann mit strftime:

```
>>> print(time.strftime('%d %b %Y', t))
23 Jan 2010
```

Wie du siehst, braucht strtime 2 Argumente: das Erste für die Art der Anzeige und das Zweite mit dem Tupel für die Zeit und Datumswerte. Die Kürzel %d %b %Y sagen Python: zeige nur den Tag, das Monat und das Jahr. Wir könnten auch das Monat als Nummer ausgeben:

```
>>> print(time.strftime('%d/%m/%Y', t))
23/01/2010
```

Damit sagst du Python ‘zeige den Tag dann einen Schrägstrich, dann den Monat als Nummer, dann wieder einen Schrägstrich, dann das Jahr. Es gibt eine Anzahl von verschiedenen Varianten, die du für die Anzeige verwenden kannst.

%a	ein Abkürzung für den Wochentag in Englisch (Mon, Tues, Wed, Thurs, Fri, Sat, und Sun)
%A	der lange Name des Wochentags in Englisch (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday)
%b	die englische Abkürzung für das Monat (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct , Nov, Dec)
%B	der Name des Monats in Englisch (January, February, March, April, May, and so on)
%c	Datum und Zeit im gleichen Format wie asctime
%d	der Tag des Monats als Zahl (von 01 bis 31)
%H	die Stunde des Tages im 24 Stunden Format (von 00 bis 23)
%I	die Stunde des Tages im 12 Stunden Format (from 01 to 12)
%j	der Tag des Jahres als Zahl (von 001 bis 366)
%m	der Monat als Zahl (von 01 bis 12)
%M	die Minute als Zahl von (von 00 bis 59)
%p	Vormittags oder Nachmittags als AM oder PM
%S	die Sekunden als Zahl
%U	die Nummer der Woche als Zahl (von 00 bis 53)
%w	der Tag der Woche als Zahl. Sonntag ist 0, Montag ist 1, bis Samstag mit der Nummer 6
%x	ein einfaches Datumsformat (normalerweise Monat/Tag/Jahr—zum Beispiel 01/23/10)
%X	ein einfaches Zeitformat (normalerweise Stunden, Minuten, Sekunden—zum Beispiel 20:52:30)
%y	das Jahr in 2 Ziffern (2010 wäre dann 10)
%Y	das Jahr in 4 Ziffern (zum Beispiel 2010)

strptime

Die Funktion strptime ist fast das Gegenteil von strftime—es nimmt einen String und wandelt den String in einen Tupel um. Dazu gibst du der Funktion die gleichen Parameter mit, wie sie auch strftime versteht:

```
>>> import time
>>> t = time.strptime('01 Jan 2010', '%d %b %Y')
>>> print(t)
time.struct_time(tm_year=2010, tm_mon=1, tm_mday=1, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=4, tm_yday=1, tm_isdst=-1)
```

Wenn unser String jedoch so aussieht Tag/Monat/Jahr (zum Beispiel, 23/01/2010) dann würden wir folgendes schreiben:

```
>>> import time
>>> t = time.strptime('23/01/2010', '%d/%m/%Y')
>>> print(t)
time.struct_time(tm_year=2010, tm_mon=1, tm_mday=23, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=5, tm_yday=23, tm_isdst=-1)
```

Oder es wäre Monat/Tag/Jahr, dann passen wir es so an:

```
>>> import time
>>> t = time.strptime('01/23/2010', '%m/%d/%Y')
>>> print(t)
time.struct_time(tm_year=2010, tm_mon=1, tm_mday=23, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=5, tm_yday=23, tm_isdst=-1)
```

Und dann können wir auch noch beide Funktionen kombinieren um einen String von einem ins andere Format zu verwandeln. Bauen wir dazu eine Funktion:

```
>>> import time
>>> def wandle_datum(datums_string, format1, format2):
...     t = time.strptime(datums_string, format1)
...     return time.strftime(format2, t)
...
```

Diese Funktion rufen wir mit 3 Parametern auf. Einem String fürs Datum, das Format in dem dieser String ist und in welchem Format der String zurückkommen soll.

```
>>> print(wandle_datum('01/23/2010', '%m/%d/%Y', '%d %B %Y'))
23 January 2010
```

D

Antworten zu “Probiere es aus”

Hier kannst du die Antworten zu den Fragen finden, die in jedem Kapitel unter ‘Probiere es aus’ gestellt wurden.

Antworten zu Kapitel 2

Lösung zu Kapitel 2, Übung 1

Die Antwort zu Kapitel 2, **Übung 1** könnte so aussehen:

```
>>> spielsachen = [ 'Auto', 'Nintendo Wii', 'Computer', 'Fahrrad' ]
>>> speisen = [ 'Pfannkuchen', 'Schokolade' ]
>>> favouriten = spielsachen + speisen
>>> print(favouriten)
['Auto', 'Nintendo Wii', 'Computer', 'Fahrrad', 'Pfannkuchen', 'Schokolade']
```

Lösung zu Kapitel 2, Übung 2

Die Antwort zu **Übung 2** ist das Ergebnis von zwei Multiplikationen, deren Ergebnis miteinander addiert werden. Zuerst wird 3 mal 25 multipliziert und dann 10 mit 32. Die beiden Ergebnisse werden addiert. Die folgende Formel führt zur Lösung:

```
>>> print(3 * 25 + 10 * 32)
395
```

Aber vielleicht hast du auch Klammern verwendet, nachdem wir das in Kapitel 2 so genau behandelt haben. Vielleicht hast du folgende Formel aufgeschrieben:

```
>>> print((3 * 25) + (10 * 32))
395
```

Das Ergebnis ist wieder das gleiche, weil die Multiplikation vor der Addition ausgeführt wird. In jeder Formel werden die zwei Multiplikationen zuerst ausgeführt und dann addiert. Aber die zweite Formel ist etwas besser als die Erste—weil der Leser sofort erkennt, welche Teile zuerst berechnet werden. Jemand, der sich mit der Reihenfolge der Operatoren nicht so gut auskennt, könnte ansonsten meinen, dass in der ersten Formel 3 mit 25 multipliziert wird, dann 10 dazugezählt und das Ergebnis mit 32 multipliziert wird (das Ergebnis wäre dann 2720 und das wäre völlig falsch). Mit den Klammern wird es etwas klarer, was zuerst berechnet wird.

Lösung zu Kapitel 2, Übung 3

Die Antwort wird ungefähr so ausschauen:

```
>>> vorname  = 'Santa'
>>> nachname = 'Klaus'
>>> print('Mein Name ist %s %s' (vorname, nachname))
Mein Name ist Santa Klaus
```

Antworten zu Kapitel 3

Lösung zu Kapitel 3, Übung 1

Ein Rechteck ist wie ein Quadrat, aber bei einem Rechteck sind zwei Seiten länger als die anderen zwei. So könntest du der Schildkröte befehlen ein Rechteck zu zeichnen.

- bewege dich um eine Anzahl von Pixel nach vorne
- drehe dich nach links
- bewege dich eine kleinere Anzahl von Pixel nach vorne
- drehe dich nach links
- bewege dich um eine Anzahl von Pixel nach vorne
- drehe dich nach links
- bewege dich eine kleinere Anzahl von Pixel nach vorne

Zum Beispiel würde der folgende Code das Rechteck in Abbildung D.1 erzeugen.

```
>>> import turtle
>>> schildkroete = turtle.Pen()
>>> schildkroete.forward(150)
>>> schildkroete.left(90)
>>> schildkroete.forward(50)
>>> schildkroete.left(90)
>>> schildkroete.forward(150)
>>> schildkroete.left(90)
>>> schildkroete.forward(50)
```

Ein Dreieck zu zeichnen ist schon etwas komplizierter, weil du die Grad und die Linienlänge rausfinden musst. Wenn du noch nicht viel darüber in der Schule gehört hast, könnte das ein wenig schwerer sein. Hier wäre der Code, der das Bild in Abbildung D.2 erzeugt:

```
>>> import turtle
>>> schildkroete = turtle.Pen()
>>> schildkroete.forward(100)
>>> schildkroete.left(135)
>>> schildkroete.forward(70)
>>> schildkroete.left(90)
>>> schildkroete.forward(70)
```

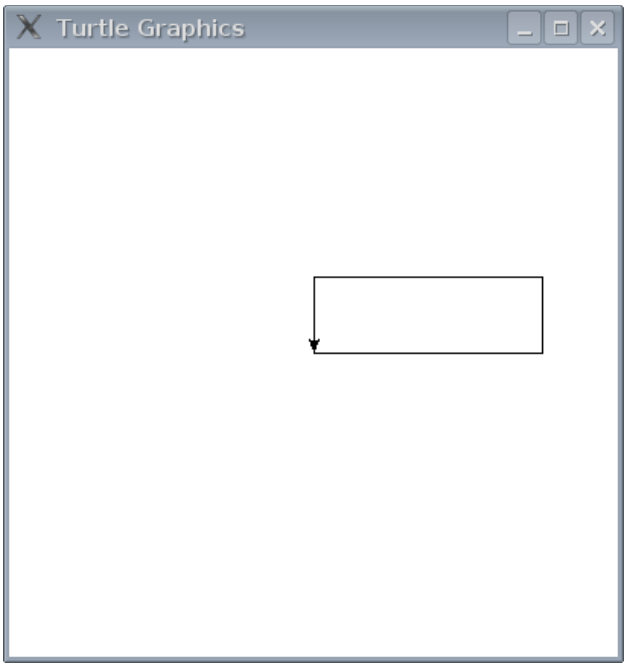


Abbildung D.1: Die Schildkröte malt ein Rechteck.

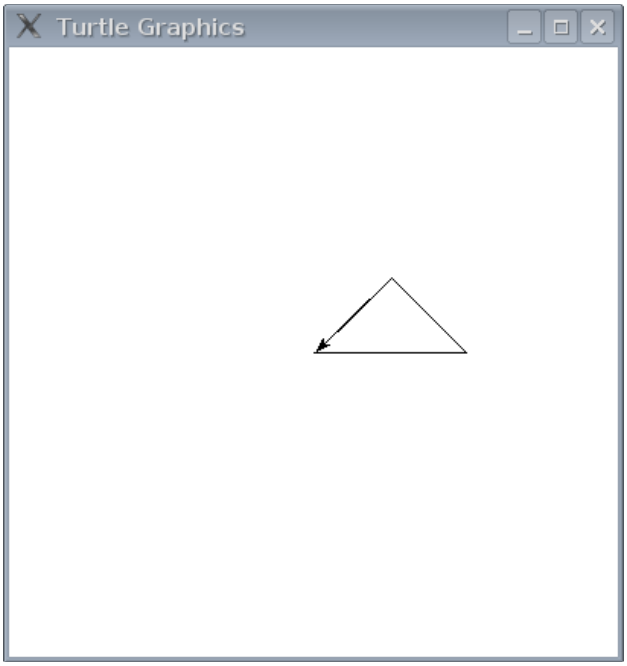


Abbildung D.2: Die Schildkröte malt ein Dreieck.

Antworten zu Kapitel 5

Lösung zu Kapitel 5, Übung 1

Die Schleife wird nach der ersten print Zeile verlassen. Also wird der Code folgendes ausgeben:

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

Der Grund, warum nur einmal etwas ausgegeben wird, ist folgender: die Variable x ist am Anfang 0. Da 0 weniger als 9 ist, wird der break Befehl ausgeführt und beendet die Schleife.

Lösung zu Kapitel 5, Übung 2

Bei 3 Prozent Zinsen kannst du die Zinsen errechnen, indem du die Geldsumme mit 0.03 multiplizierst. Am Anfang erzeugen wir am besten eine Variable und lassen sie auf die Geldsumme zeigen.

```
>>> geldsumme = 100
```

Die Zinsen nach einem Jahr ergeben sich indem die Geldsumme mit 0.03 multipliziert wird:

```
>>> geldsumme = 100
>>> print(geldsumme * 0.03)
3.0
```

Das sind 3€! Nicht schlecht, nachdem wir nichts dafür getan haben. Jetzt geben wir diesen Wert aus, addieren ihn zur Geldsumme und berechnen die Zinsen für die größere Geldsumme von neuem. Insgesamt 10 mal.

```
>>> geldsumme = 100
>>> for jahr in range(1, 11):
...     zinsen = geldsumme * 0.03
...     print('Die Zinsen für das %s Jahr sind %s' % (jahr, zinsen))
...     geldsumme = geldsumme + zinsen
...
Die Zinsen für das 1 Jahr sind 3.0
Die Zinsen für das 2 Jahr sind 3.09
Die Zinsen für das 3 Jahr sind 3.1827
Die Zinsen für das 4 Jahr sind 3.278181
Die Zinsen für das 5 Jahr sind 3.37652643
Die Zinsen für das 6 Jahr sind 3.4778222229
Die Zinsen für das 7 Jahr sind 3.58215688959
Die Zinsen für das 8 Jahr sind 3.68962159627
Die Zinsen für das 9 Jahr sind 3.80031024416
Die Zinsen für das 10 Jahr sind 3.91431955149
```

In der ersten Zeile erzeugen wir eine for-Schleife mit der jahr Variable und der range Funktion, die von 1 bis 10 zählt. Die zweite Zeile berechnet die Zinsen, indem die Variable geldsumme mit 0.03 multipliziert wird. In der nächsten Zeile kommt die print Funktion mit allen Platzhaltern (%s) damit die Werte von den Variablen jahr und zinsen eingefügt werden. Die vielen Nachkommastellen sind vielleicht verwirrend. Aber du erkennst, wie du jedes Jahr ein wenig mehr Zinsen bekommst, da die Geldsumme steigt. Im nachfolgenden Code geben wir noch die Geldmenge dazu, damit du einen besseren Überblick bekommst.

```
>>> geldsumme = 100
>>> for jahr in range(1, 11):
...     zinsen = geldsumme * 0.03
...     print('Zinsen für %s Euro im %s Jahr sind %s' % (geldsumme, jahr, zinsen))
...     geldsumme = geldsumme + zinsen
...
Zinsen für 100 Euro im 1 Jahr sind 3.0
Zinsen für 103.0 Euro im 2 Jahr sind 3.09
Zinsen für 106.09 Euro im 3 Jahr sind 3.1827
Zinsen für 109.2727 Euro im 4 Jahr sind 3.278181
Zinsen für 112.550881 Euro im 5 Jahr sind 3.37652643
Zinsen für 115.92740743 Euro im 6 Jahr sind 3.4778222229
Zinsen für 119.405229653 Euro im 7 Jahr sind 3.58215688959
Zinsen für 122.987386542 Euro im 8 Jahr sind 3.68962159627
Zinsen für 126.677008139 Euro im 9 Jahr sind 3.80031024416
Zinsen für 130.477318383 Euro im 10 Jahr sind 3.91431955149
```

Antworten zu Kapitel 6

Lösung zu Kapitel 6, Übung 1

Das Umwandeln von einer for-Schleife in eine Funktion ist recht einfach. Die Funktion wird ungefähr so ausschauen:

```
>>> def berechne_zinsen(geldsumme, zinssatz):
...     for jahr in range(1, 11):
...         zinsen = geldsumme * zinssatz
...         print('Zinsen für %s Euro im %s Jahr sind %s' % (geldsumme, jahr, zinsen))
...         geldsumme = geldsumme + zinsen
...
```

Wenn du den Code mit der Funktion aus dem vorigen Kapitel vergleichst, fällt dir vielleicht auf, dass abgesehen von der ersten Zeile alles gleich ist (0.03 wurde jetzt zum Parameter zinssatz). Weil geldsumme schon eine Variable war, blieb da alles gleich. Die Ausgabe ist auch identisch, wenn du die Funktion verwendest:

```
>>> berechne_zinsen(100, 0.03)
Zinsen für 100 Euro im 1 Jahr sind 3.0
Zinsen für 103.0 Euro im 2 Jahr sind 3.09
Zinsen für 106.09 Euro im 3 Jahr sind 3.1827
Zinsen für 109.2727 Euro im 4 Jahr sind 3.278181
```

```
Zinsen für 112.550881 Euro im 5 Jahr sind 3.37652643
Zinsen für 115.92740743 Euro im 6 Jahr sind 3.4778222229
Zinsen für 119.405229653 Euro im 7 Jahr sind 3.58215688959
Zinsen für 122.987386542 Euro im 8 Jahr sind 3.68962159627
Zinsen für 126.677008139 Euro im 9 Jahr sind 3.80031024416
Zinsen für 130.477318383 Euro im 10 Jahr sind 3.91431955149
```

Lösung zu Kapitel 6, Übung 2

Die Änderung, dass auch das Jahr nun ein Parameter ist, benötigt auch nur eine kleine Anpassung:

```
>>> def berechne_zinsen(geldsumme, zinssatz, jahre):
...     for jahr in range(1, jahre):
...         zinsen = geldsumme * zinssatz
...         print('Zinsen für %s Euro im %s Jahr sind %s' % (geldsumme, jahr, zinsen))
...         geldsumme = geldsumme + zinsen
...
```

Nun kannst du einfach die Geldsumme, den Zinssatz und die Jahre bei jedem Aufruf der Funktion mitgeben und nach Wünschen ändern:

```
>>> berechne_zinsen(1000, 0.05, 6)
Zinsen für 1000 Euro im 1 Jahr sind 50.0
Zinsen für 1050.0 Euro im 2 Jahr sind 52.5
Zinsen für 1102.5 Euro im 3 Jahr sind 55.125
Zinsen für 1157.625 Euro im 4 Jahr sind 57.88125
Zinsen für 1215.50625 Euro im 5 Jahr sind 60.7753125
```

Lösung zu Kapitel 6, Übung 3

Dieses kleine Programm ist ein wenig komplizierter, als das bisherige. Zuerst müssen wir das sys Modul importieren um Tastatureingaben verarbeiten zu könnnen. Und wir fragen den Benutzer nach den einzelnen Werten. Sonst bleibt es gleich.

```
>>> import sys
>>> def berechne_zinsen():
...     print('Wieviel willst du sparen?')
...     geldsumme = float(sys.stdin.readline())
...     print('Gib den Zinssatz ein')
...     zinssatz = float(sys.stdin.readline())
...     print('Wieviele Jahre bleibt das Geld auf der Bank')
...     jahre = int(sys.stdin.readline())
...     for jahr in range(1, jahre+1):
...         zinsen = geldsumme * zinssatz
...         print('Zinsen für %s Euro im %s Jahr sind %s' % (geldsumme, jahr, zinsen))
...         geldsumme = geldsumme + zinsen
...
```

Wenn wir die Funktion nun aufrufen bekommen wir folgendes:

```
>>> berechne_zinsen()
Wieviel willst du sparen?
500
Gib den Zinssatz ein
0.06
Wieviele Jahre bleibt das Geld auf der Bank?
12
Zinsen für 500 Euro im 1 Jahr sind 30.0
Zinsen für 530.0 Euro im 2 Jahr sind 31.8
Zinsen für 561.8 Euro im 3 Jahr sind 33.708
Zinsen für 595.508 Euro im 4 Jahr sind 35.73048
Zinsen für 631.23848 Euro im 5 Jahr sind 37.8743088
Zinsen für 669.1127888 Euro im 6 Jahr sind 40.146767328
Zinsen für 709.259556128 Euro im 7 Jahr sind 42.5555733677
Zinsen für 751.815129496 Euro im 8 Jahr sind 45.1089077697
Zinsen für 796.924037265 Euro im 9 Jahr sind 47.8154422359
Zinsen für 844.739479501 Euro im 10 Jahr sind 50.6843687701
Zinsen für 844.739479501 Euro im 11 Jahr sind 53.7254308963
Zinsen für 949.149279168 Euro im 12 Jahr sind 56.9489567501
```

Antworten zu Kapitel 8

Lösung zu Kapitel 8, Übung 1

Es gibt eine lange Variante ein Achteck zu zeichnen und eine kurze kompliziertere. Bei der langen Variante ist viel zu tippen.

```
import turtle
schildkroete = turtle.Pen()
>>> schildkroete.forward(50)
>>> schildkroete.right(45)
>>> schildkroete.forward(50)
>>> schildkroete.right(45)
>>> schildkroete.forward(50)
>>> schildkroete.right(45)
>>> schildkroete.forward(50)
>>> schildkroete.right(45)
>>> schildkroete.forward(50)
>>> schildkroete.right(45)
>>> schildkroete.forward(50)
>>> schildkroete.right(45)
>>> schildkroete.forward(50)
>>> schildkroete.right(45)
>>> schildkroete.forward(50)
```

Wie du sehen kannst, sagen wir der Schildkröte, dass sie sich 50 Pixel nach vorne bewegen soll und dann 45 Grad nach rechts drehen. Wir machen das ganze 8 mal. Was ziemlich oft ist. Die kurze Variante ist Folgende (die das folgende Achteck in Abbildung D.3 erzeugt):

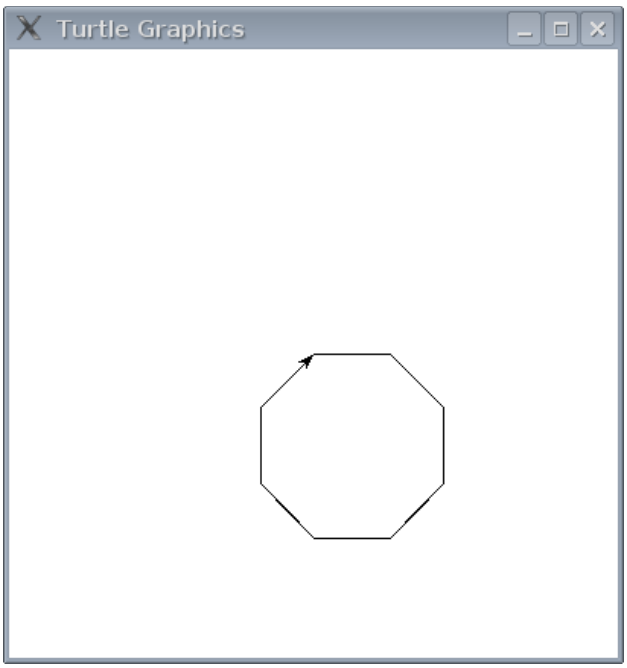


Abbildung D.3: Die Schildkröte zeichnet ein Achteck.

```
>>> for x in range(0,8):
...     schildkroete.forward(50)
...     schildkroete.right(45)
```

Lösung zu Kapitel 8, Übung 2

Wenn du dich nochmals in Kapitel 8 schlau machst, findest du die Funktion, die nötig ist, um die Formen mit Farbe zu füllen. Ändern wir die achteck-Funktion, damit wir gleich die Farbe mitgeben können.

```
>>> def achteck(rot, gruen, blau):
...     schildkroete.color(rot, gruen, blau)
...     schildkroete.begin_fill()
...     for x in range(0,8):
...         schildkroete.forward(50)
...         schildkroete.right(45)
...     schildkroete.end_fill()
```

In der Funktion setzen wir die Farbe und schalten dann 'ausfüllen' ein. Wir zeichnen danach das Achteck und schalten 'ausfüllen' aus, um das Achteck zu füllen. Für ein blaues Achteck (siehe Bild D.4) geben wir Folgendes ein:

```
>>> achteck(0, 0, 1)
```

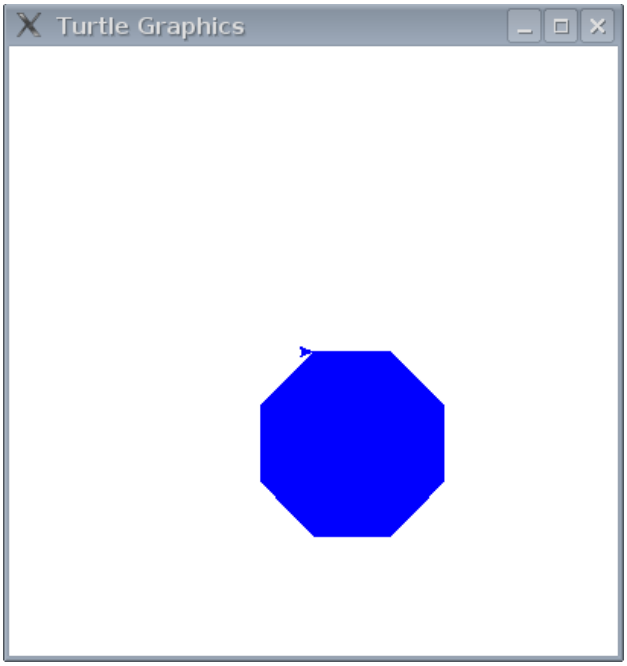


Abbildung D.4: Die Schildkröte malt ein blaues Achteck.

Index