B4 Die Türme von Hanoi - grafisch

Inhalt

Einleitung	1
Grafik – einmal anders	2
Die Türme von Hanoi – grafisch	8

Einleitung

Dieses Bonuskapitel setzt voraus, dass du die Kapitel 11 – 13 durchgearbeitet hast sowie den letzten Abschnitt von Bonuskapitel 3 über Rekursion. Es hat darüber hinaus geringfügige Überschneidungen mit dem ersten Abschnitt von Kapitel 14. Dies deshalb, damit du es lesen kannst, bevor du Kapitel 14 bearbeitest.

In diesem Kapitel erfährst du …

- … wie du die Lösung des Spiels »Türme von Hanoi« anstatt durch trockene Textzeilen (Bonuskapitel 3) durch eine grafische Animation darstellen kannst.
- … wie du mit Rechtecken, Objekten der Klasse Rectangle, auf einem Tkinter-Canvas umgehen kannst.
- … wie du von dieser Klasse eine neue ableitest: bewegliche Rechtecke – und wie du sie auf einem Canvas herumschieben kannst.
- … wie du von dem in Python eingebauten Typ Liste eine neue Klasse ableitest
- … und wie du das ganze zu einer ausgewachsenen Simulation zusammenfügst.

 © 2003 by verlag moderne industrie Buch AG & Co. KG, Bonn

Grafik – einmal anders!

➔ Schließe das Turtle-Grafik-Fenster. Gib ein:

```
>>> root = Tk()
>>> cv = Canvas(root,width=240,height=150,bg="yellow")
>>> cv.pack()
```

Mit diesen drei Anweisungen erhältst du stets eine benutzbare Leinwand!

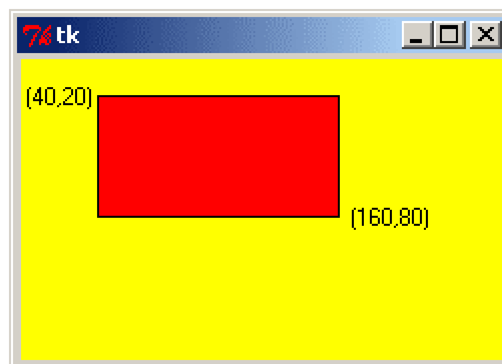
Nach demselben Muster erzeugen wir uns nun Rechtecke. Dafür gibt es die Klasse `Rectangle`, die im Modul `Canvas` definiert ist. (Das Modul `Canvas`, also die Datei `Canvas.py`, ist etwas anderes als die Klasse `Canvas` aus dem Modul `Tkinter`!)

Bevor wir das tun können, muss ich dir noch erklären, wie das Koordinatensystem auf einem Canvas eingerichtet ist. Nämlich so: Der Ursprung des Koordinatensystems, der Punkt mit den Koordinaten (0/0) ist in der linken oberen Ecke des Canvas. Die positive x-Achse zeigt vom Ursprung nach rechts, die positive y-Achse zeigt vom Ursprung nach unten.

All das lässt sich aber leicht mit Rechteck-Objekten erkunden.

➔ Mach weiter mit:

```
>>> from Canvas import Rectangle
>>> r = Rectangle(cv, (40,20),(160,80),fill="red")
```



Dieser Aufruf des Konstruktors `Rectangle` erzeugt ein Rechteck auf dem Canvas `cv`. Die linke obere Ecke des Rechtecks hat die Koordinaten x1 = 40 und y1 = 20. Die rechte untere Ecke des Rechtecks hat die Koordinaten x2 = 160 und y2 = 80. Das Rechteck ist mit der Farbe Rot gefüllt.

Merkt sich das Rechteck `r` eigentlich, auf welchem Canvas es sich befindet? Wir wissen ja, es ist der Canvas, dem wir den Name `cv` gegeben haben.

```
>>> cv
<Tkinter.Canvas instance at 0x00A82678>
```

Dieses haben wir dem Konstruktor `Rectangle` als Argument übergeben. Und die Rechtecke weisen dies einem Attribut `canvas` des Rechteckobjekts zu:

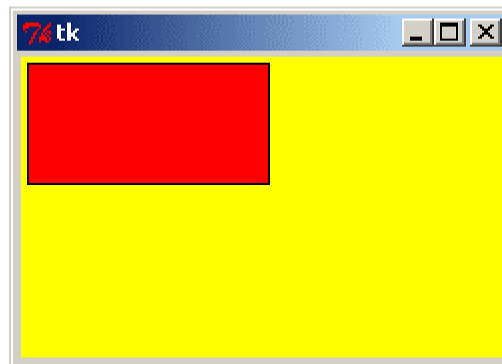
```
>>> r.canvas
<Tkinter.Canvas instance at 0x00A82678>
```

Das ist keine Nebensächlichkeit! Denn über das Attribut `canvas` hat `r` Zugriff auf alle Methoden und Attribute von `cv`!

Wir sehen uns jetzt noch einige wenige Methoden der Klasse `Rectangle` an:

```
>>> r.move(20,50)
>>> r.coords()
[60.0, 70.0, 180.0, 130.0]
>>> r.move(-55,-65)
```

Die Methode `move(dx,dy)` verschiebt das Rechteck um dx in die x-Richtung und um dy in die y-Richtung. Die Methode `coords` gibt die Koordinaten der beiden gegenüberliegenden Eckpunkte als Liste zurück. Sehr praktisch, um auch die Koordinaten einzeln zu bestimmen:




```
>>> x1,y1,x2,y2=r.coords()
>>> x1,y1
(5.0, 5.0)
>>> x2,y2
(125.0, 65.0)
```

Jetzt versuchen wir, das Rechteck schrittweise nach links unten zu schieben:

```
>>> for i in range(20):
        r.move(3,2)
```

Das hat zwar irgendwie geklappt, aber man hat nichts von der Bewegung gesehen. Zu schnell gegangen? Tun wir's wieder zurück!

```
>>> r.move(-60,-40)
```

Da helfen zwei Methoden der Klasse `Canvas`: Die Methode `update` führt ein Neuzeichnen des Canvas aus (denke an `tracer`!), und die Methode `after` legt eine Pause ein, deren Länge in Millisekunden angegeben werden kann:

```
>>> for i in range(20):
        r.move(3,2)
        cv.update()
        cv.after(100)
```

Das ergibt eine langsame Bewegung des Rechtecks nach rechts unten.

➔ Experimentiere nun mit verschiedenen Argumenten für `move` und `after`. Ändere vielleicht mit `cv.config` die Größe des Canvas.

Nach diesen Experimenten mit Rechtecken auf einem Canvas möchte ich uns die folgende Aufgabe stellen:

Wir erstellen eine Klasse, deren Objekte bewegliche rote Rechtecke sind. Die Klasse soll von `Rectangle` abgeleitet werden und über eine zusätzliche Methode `schiebe_nach` verfügen. Der Methoden-Aufruf

```
r.schiebe_nach(x,y)
```

soll das Rechteck auf dem Canvas (mit angemessener Geschwindigkeit) so verschieben, dass sich nach der Schiebung der Mittelpunkt der unteren Seite an der Stelle (x,y) befindet.

Wir nennen diese neue Klasse `Scheibe` – (schon in Hinblick auf die Türme von Hanoi). Zur Konstruktion eines Scheibe-Objekts stehen folgende Angaben zur Verfügung: Der Mittelpunkt der unteren Seite, die Länge und die Höhe. Natürlich muss auch ein Canvas zur Verfügung stehen, auf dem die Scheibe unterzubringen ist. Ein Scheibe-Objekt wird also so zu erzeugen sein:

```
s = Scheibe(canvas, mittelpunkt, laenge, hoehe)
```

Was müssen wir also programmieren?

Eine von `Rectangle` aus `Canvas` abgeleitete Klasse `Scheibe`, einen Konstruktor und eine Methode `schiebe_nach`. Der Code muss also folgenden Aufbau haben:

```
from Canvas import Rectangle
class Scheibe(Rectangle):
    def __init__(self, canvas, mittelpunkt,
                 laenge, hoehe):
        pass
    def schiebe_nach(self, x, y):
        pass
```

➔ Öffne von der IDLE aus ein Editor-Fenster für ein Modul `schiebe.py`.

➔ Schreibe einen Kopfkommentar und das obige Gerüst für den Code der Klasse in die Datei und speichere unter dem Dateinamen ab.

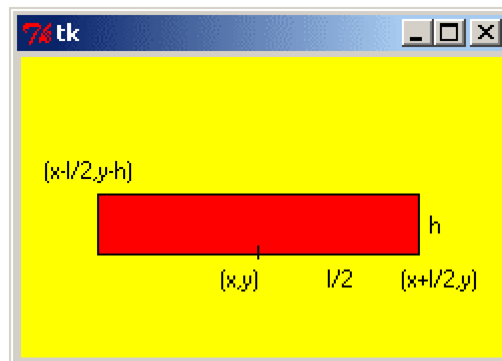
Der Konstruktor ist noch eine eher leichte Übung. Du musst nur aus den Angaben die benötigten Koordinaten der Eckpunkte des Rechtecks ermitteln. Mit denen kann dann der Konstruktor der Oberklasse aufgerufen werden:

```
    def __init__(self,canvas,mittelpunkt,laenge,hoehe):
        # Berechnung von x1,y1, x2,y2
        Rectangle.__init__(canvas,(x1,y1),(x2,y2),
                                   fill = "red")
```

Wie berechnet man nun die Koordinaten der Eckpunkte? Die Koordinaten des Mittelpunkts der Unterseite bekommt man leicht mit:

```
x, y = mittelpunkt
```

Aus untenstehender Grafik kann man leicht die gewünschten Koordinaten ablesen.



➔ Vervollständige nun den Code des Konstruktors von `Scheibe`, speichere das Programm und führe es aus.

➔ Nun erzeuge im vorhandenen Canvas `cv` einige Scheiben und prüfe nach, ob sie die gewünschte Größe und Lage haben. Zum Beispiel:

```
>>> s = Scheibe(cv, (120,140),60, 20)
```

Passt es?

Schwieriger wird die Sache schon mit der Methode `schiebe_nach`. Doch Moment mal, unsere Experimente legen uns nahe, dass der Code jedenfalls etwa so aussehen muss:

```
    def schiebe_nach(self, x, y):
        # Berechnungen von
        # schritte, dx, dy
        for i in range(schritte):
            self.move(dx,dy)
            self.canvas.update()
            self.canvas.after(50)
```

Hier ist es ganz wichtig, dass die Rechtecke das Attribut `canvas` haben. Damit können sie jederzeit dafür sorgen, dass der Canvas neu gezeichnet wird! (So überraschend ist das nun auch wieder nicht. Die `Pen`-Objekte konnten das schließlich auch!)

Wie berechnen wir nun die Anzahl und die Größe der Schritte? Da wollen wir gleich Nägel mit Köpfen machen. Leicht wäre ja zu sagen: Nehmen wir immer zehn Schritte! Aber dann würde die Bewegungsgeschwindigkeit bei langen Strecken hoch und bei kurzen Strecken ganz klein sein. Sie sollte aber eher überall gleich groß sein. Also legen wir fest: ca. 10 Pixel pro Schritt!

Jetzt rechnen wir uns zuerst die Gesamtverschiebung aus:

```
x1,x2,y1,y2 = self.coords()
```

liefert uns die Koordinaten der Rechteck-Ecken. Der untere Mittelpunkt hat xm = (x1 + x2) / 2 und ym = y2. Damit ergeben sich die Verschiebungen: dx = x - xm und dy = y - ym. Die Länge dieser Verschiebung ist – wieder einmal Pythagoras! – d = sqrt (dx ** 2 + dy ** 2).

Wir haben nun die Gesamtverschiebung dx,dy und die Gesamtlänge der Strecke d berechnet. Wie kommen wir davon zur Schrittanzahl?

Um diese zu erhalten, müssen wir fragen, wie oft die vorgesehene Schrittlänge 10 in d enthalten ist? d / 10! Das ist eine Kommazahl zwischen 0 und irgendwo, und eine Kommazahl können wir als Schrittlänge nicht brauchen. Also schneiden wir sie ab: `int(d/10)`. Dies ist nun eine ganze Zahl zwischen 0 und irgendwo ... Null können wir aber auch nicht brauchen: erstens, weil wir die Größe eines Schrittes erhalten, indem wir den Gesamtweg durch die Schrittanzahl dividieren, und durch Null darf man aber bekanntlich nicht dividieren. (Wie wohl Python darauf reagieren würde?) Zweitens aber auch, weil uns 0 Schritte nicht weiterbringen. Also nehmen wir um eins mehr:

```
schritte = 1 + int(d/10)
```

und berechnen daraus die Verschiebung pro Schritt:

```
dx, dy = dx/schritte, dy/schritte
```

➔ Setze diese Überlegungen in die Codierung der Methode `schiebe_nach` um.

➔ Speichere das Programm und teste es: Erzeuge – vielleicht nun auf einem frischen Canvas – eine Scheibe und bewege sie mit mehreren `schiebe_nach`-Aufrufen auf dem Canvas herum:

```
>>> root = Tk()
>>> cv = Canvas(root,width=400,height=300,bg="yellow")
>>> cv.pack()
>>> s = Scheibe(cv, (200,100), 100, 20)
```

Auf- und abschieben:

```
>>> s.schiebe_nach(200,250)
>>> s.schiebe_nach(200,50)
```

Hin- und herschieben:

```
>>> s.schiebe_nach(110,50)
>>> s.schiebe_nach(290,50)
```

Und runter und schräg rauf.

```
>>> s.schiebe_nach(290,250)
>>> s.schiebe_nach(105,25)
```

Wenn dein Code so ähnlich aufgebaut ist, wie der hier abgedruckte, dann sollte es auch bei dir geklappt haben:

```
from Canvas import Rectangle

class Scheibe(Rectangle):
    "Bewegliches Rechteck auf einem Tkinter-Canvas"
    def __init__(self,cv,pos,laenge,hoehe):
        x0, y0 = pos
        x1, x2 = x0-laenge/2.0, x0+laenge/2.0
        y1, y2 = y0-hoehe, y0
        Rectangle.__init__(self,cv,x1,y1,x2,y2,
                                   fill = "red")
    def schiebe_nach(self, x, y):
        from math import sqrt
        x1,y1,x2,y2 = self.coords()
        xm, ym = (x1 + x2)/2, y2
        dx, dy = x-xm, y-ym
        d = sqrt(dx**2+dy**2)
        schritte = int(d/10) + 1
```

```
        dx, dy = dx/schritte, dy/schritte
        for i in range(schritte):
            self.move(dx,dy)
            self.canvas.update()
            self.canvas.after(20)
```

➔ Speichere diesen Code als Datei `scheibe.py` im Verzeichnis `C:\Py4Kids\mylib` ab.

In Zukunft hast du nun mit `from scheibe import Scheibe` bewegliche Rechtecke zur Verfügung!

Die Türme von Hanoi – grafisch!

Wo werden schon Scheiben verschoben? Wenn wir das nun können, dann wollen wir es auch anwenden! Du erinnerst dich sicher noch an die Lösung dieses Spiels, die uns das rekursive Programm `hanoi.py` geliefert hat:

```
def versetze(n,von,nach,hilf):
    if n==1:
        zug(von,nach)
    else:
        versetze(n-1,von,hilf,nach)
        zug(von,nach)
        versetze(n-1,hilf,nach,von)

def zug(von_turm,nach_turm):
    print von_turm, "->", nach_turm

def hanoi(n):
    versetze(n,"A","B","C")

if __name__ == "__main__":
    hanoi(3)
```

Die Ausführung dieses Programms hat folgende Ausgabe ergeben:

```
A -> B
A -> C
B -> C
```

```
A -> B
C -> A
C -> B
A -> B
```

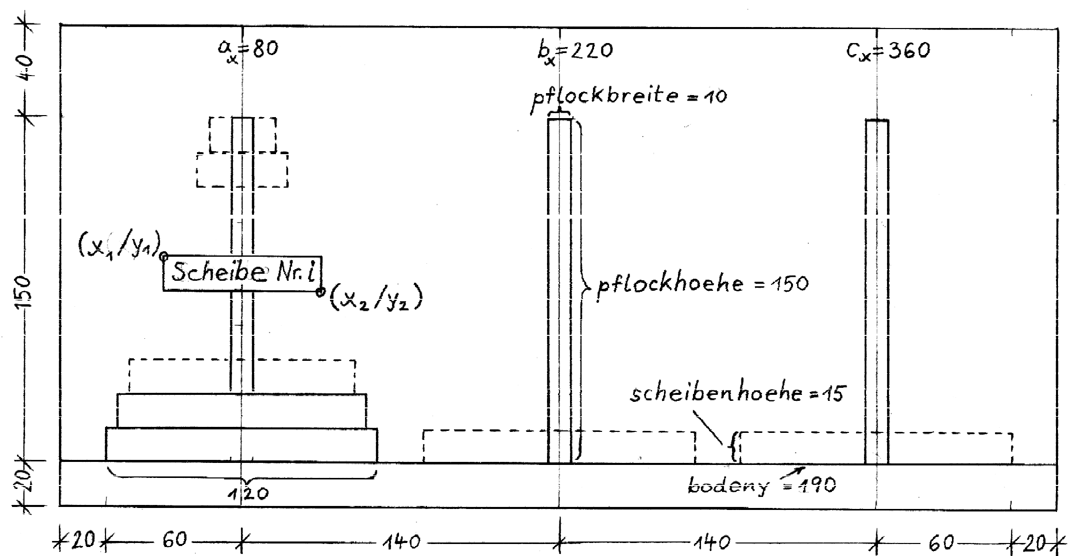
Genug, um das Spiel nachzuspielen, aber nicht sehr attraktiv. Deshalb wollen wir eine grafische Simulation des Spiels erstellen. Diese Arbeit erledigen wir in zwei Abschnitten:

1. Darstellung der Ausgangssituation auf einem Canvas: drei Pflöcke und n Scheiben auf dem ersten Pflock aufgestapelt. Als Scheiben werden wir naheliegenderweise Objekte der eben programmierten Klasse `Scheibe` verwenden.
2. Umprogrammierung der Funktion `zug(von, nach)`, damit die Textausgabe der Züge durch echtes grafisches Verschieben der Scheiben dargestellt wird.

Der erste Punkt ist einfach zu erledigen, wenn vielleicht auch etwas langwierig. Der zweite dagegen erfordert etwas mehr Gedankenarbeit.

Die Szene

Bevor wir zu programmieren beginnen, erstellen wir eine Zeichnung als Entwurf, wie die grafische Darstellung auf dem Bildschirm aussehen soll.



In dieser Grafik sind verschiedene Bestimmungsstücke mit Namen bezeichnet und für alle Objekte die Abmessungen angegeben.

Normalerweise wirst du versuchen, eine derartige grafische Darstellung flexibel zu gestalten. Das heißt, du wirst die Lage der Pflöcke, die Höhe der Scheiben usw. mit Namen versehen, wie sie in der Skizze angegeben sind, und dann die Abmessungen aller geometrischen Objekte mit diesen Namen definieren.

Ich möchte mich hier aber auf das Wesentliche der Problemlösung konzentrieren. Für die Darstellung hier im Buch habe ich Daher habe ich die Koordinaten und Abmessungen der geometrischen Objekte direkt aus der Skizze abgelesen und numerisch ins Programm eingetragen.

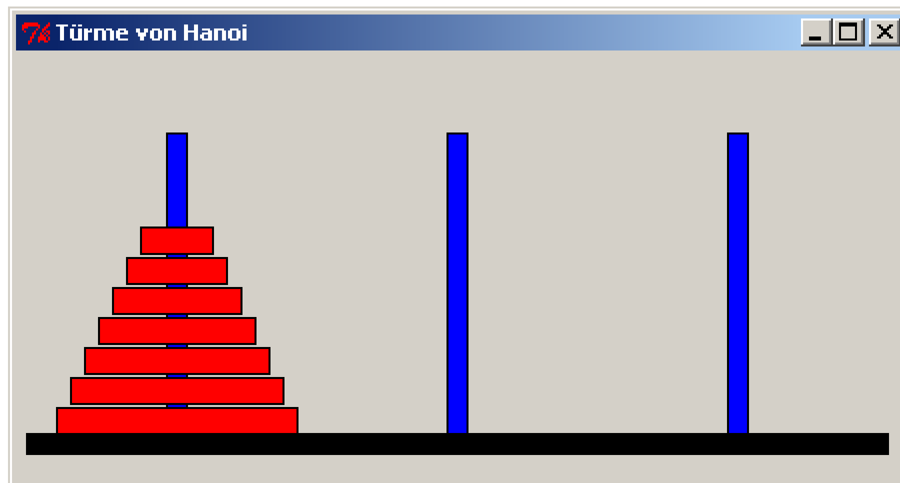
Die Türme werde ich durch Listen darstellen. Anfangs tue ich alle Scheiben in die Liste für Turm A. Bei der Durchführung eines Zugs werde ich dann eine Scheibe aus der Liste des Ausgangsturms entnehmen und der Liste des Zielturms hinzufügen.

Für die Erstellung der »Szene« mache ich mir folgenden Plan:

```
Setup der Szene für Türme von Hanoi (n Scheiben):
    Hauptfenster Tk erzeugen
    Canvas erzeugen und packen
    Drei blaue Rechtecke für die Pflöcke erzeugen
    Ein schwarzes Rechteck für den Boden erzeugen
    turm_a ----> [ ]
    turm_b ----> [ ]
    turm_c ----> [ ]
    für i im bereich 0..(n-1) wiederhole:
        Berechne Lage, Breite und Höhe der i.ten
                                              Scheibe
         s ----> Scheibe Nr. i  (Objekt der Klasse
                                              Scheibe)
        Füge s an turm_a an.
```

Fast alles davon braucht einiges an Überlegung. Ich rate dir dennoch:

➔ Versuche jetzt gleich eine Codierung dieses Programmentwurfs, so dass das Ergebnis des Programmlaufs zum Beispiel für n = 7 aussieht wie folgt. Nach der Abbildung folgen ein paar kleine Hilfestellungen; erst dahinter gebe ich dir eine Codierung dieses Entwurfs an. Versuche es zuerst alleine!



➔ Programmiere zuerst Fenster, Canvas, die drei blauen Pflöcke und den Boden. Erst wenn das funktioniert, gehe zur Programmierung der Scheiben über.

Für die Abmessungen und Lage der Scheibe mit der Nummer i gebe ich dir hier eine kleine Hilfestellung. Der Konstruktor von Scheibe braucht neben dem Canvas drei Angaben für das Rechteck: den Mittelpunkt der unteren Seite, die Länge und die Höhe.

- Die Höhe ist für alle Scheiben gleich. Sie sollte etwas kleiner als der Scheibenabstand 15 sein, zum Beispiel 13.
- Die größte Scheibe soll die Länge 120 haben. Um die Länge der weiteren Scheiben zu bestimmen, lege zunächst die Längendifferenz zweier aufeinander folgender Scheiben fest. Du kannst dafür einen konstanten Wert von 10 verwenden. Besser ist es, die Differenz von der Anzahl n der Scheiben abhängig zu wählen, umso kleiner, je mehr Scheiben da sind. Beispielsweise 100/n. Das ergibt für n = 2 den Wert 50, für n = 10 den Wert 10. Die Scheibe Nr. i hat dann die Länge 120 – i * Längendifferenz.
- Wo ist der Mittelpunkt der unteren Seite der Scheibe mit der Nummer i? Die x-Koordinate ist jedenfalls gleich der x-Koordinate von Pflock A, in meiner Skizze ist das 80. Die y-Koordinate ergibt sich, indem man von der y-Koordinate des Bodens, hier 190, i Mal den Scheibenabstand abzieht. (Beachte, dass es hier gut passt, dass die Scheiben von i = 0 bis (n-1) nummeriert werden.)

➔ Wenn dir Rechnungen mit Papier und Bleistift keinen Spaß machen, kannst du auch den IPI benutzen, um zu experimentieren. Mach mit!

```
>>> root = Tk()
>>> cv = Canvas(root, width=440, height=210)
>>> cv.pack()
>>> pflock=Rectangle(cv,(75,40),(85,190),fill="blue")
```

```
>>> from scheibe import Scheibe
>>> s = Scheibe(cv,(80,190),120,15)
>>> d = 20
>>> s = Scheibe(cv,(80,190-15),120-d,15)
>>> for i in range(2,6):
      s = Scheibe(cv,(80,190-i*15),120-i*d,15)
```

➔ Vervollständige nun den Code, der in der Funktion `Hanoi` die Grafik initialisiert. Entwurfskizze, Programmentwurf, Hinweise und IPI-Experimente solltest du, falls nötig, nochmals zu Hilfe nehmen.

Die Aufgabe ist durchaus umfangreich, doch du hast nun schon ein ganz hübsches Arsenal an Verfahren zur Verfügung, um die Lösung zu erarbeiten.

Ich gebe dir hier zum Vergleich mit deinem eigenen Ergebnis eine Beispiellösung an. Je nach Arbeitsweise, kann deine Lösung auch beträchtlich von meiner abweichen.

```
def hanoi(n):
    root = Tk()                                # ein Fenster
    root.title("Türme von Hanoi")              # mit Titel,
    cv=Canvas(root,width=440,height=210)       # Leinwand
    cv.pack()                                  # aufspannen
    pflock1 = Rectangle(cv,( 75,40),( 85,190),
                           fill='blue')
    pflock2 = Rectangle(cv,(215,40),(225,190),
                           fill='blue')
    pflock3 = Rectangle(cv,(355,40),(365,190),
                           fill='blue')
    boden   = Rectangle(cv,(  5,190),(435,200),
                           fill='black')
    turm_a = []
    turm_b = []
    turm_c = []
    for i in range(n):                    # turm_a aufbauen
        laengen_differenz = 100 / n
        laenge = 120 - i * laengen_differenz
        xm, ym = 80, 190 - i * 15
        s = Scheibe( cv, (xm,ym), laenge, 13)
        turm_a.append(s)
```

➔ Speicher nun `hanoi.py` unter dem Namen `hanoig.py` ab und füge den obigen Code in die Funktion `hanoi` ein, die bisher nur einen Aufruf von `versetze` enthält.

Wenn du in deiner Lösung `versetze` nicht auskommentierst, wirst du mit dem bedauerlichem Umstand konfrontiert, dass die Ausgabe des Programms immer noch die Textausgabe ist. Das wollen wir nun endlich ändern.

Info zum Schieben!

Die Funktion `zug` soll so umgeschrieben werden, dass die oberste Scheibe von `von_turm` nach `nach_turm` verschoben wird. Diese Verschiebung besteht aus zwei verschiedenen Aktionen:

1. Die (letzte) Scheibe muss der Liste `von_turm` entnommen und an die Liste `nach_turm` hinten angefügt werden. Das kannst du mit den Methoden `pop` und `append` erreichen.
2. Dadurch ändert sich aber nicht die Lage der Scheibe auf dem Canvas. Sie muss daher tatsächlich von `von_turm` nach `nach_turm` mit der Methode `schiebe_nach` geschoben werden. Und zwar wollen wir das in drei Schritten machen: Sie soll zuerst angehoben, dann horizontal verschoben und zuletzt auf den Zielturm abgesenkt werden.

```
Funktion zug( von_turm, nach_turm):
    scheibe ----> von_turm.pop()
    scheibe.schiebe_nach( x des von_ turm, 20)
    scheibe.schiebe_nach( x des nach_turm, 20)
    scheibe.schiebe_nach( x des nach_turm,
                          y der Spitze von nach_turm)
    nach_turm.append(scheibe)
```

Da haben wir nun ein Problem. Wir müssen die x-Koordinaten der beiden Türme und die y-Koordinate der »Spitze« des Zielturmes ermitteln. Diese Info steckt in den Türmen nicht drin und muss von anderen Programmstellen (von globalen Variablen?) her bezogen werden. Außer ...

... außer es gelänge uns, die Türme zu Objekten zu machen, die die nötige Information enthalten. Dass diese Türme also Listen sind, die noch Zusatzinformationen enthalten.

Python bietet die Möglichkeit, Unterklassen von eingebauten Typen, also auch vom Typ Liste zu definieren. Diese Möglichkeit können wir nun ausnutzen. Sehen wir uns die Sache zuerst mit dem IPI an:

```
>>> class Turm(list):
     pass

>>> turm_a = Turm()
>>> turm_a
[]
```

Aha! Der Konstruktor der Klasse `Turm` erzeugt eine leere Liste.

```
>>> turm_a.append(3)
>>> turm_a
[3]
>>> turm_a.pop()
3
```

Objekte der Klasse `Turm` haben die Listen-Methoden geerbt!

```
>>> class Turm(list):
     def __init__(self, x,y):
           self.x = x
           self.y = y

>>> turm_a = Turm(80,190)
>>> turm_a.x
80
>>> turm_a.y
190
>>> turm_a
[]
```

Immer noch mit leerer Liste initialisiert. Das ist gerade das, was wir brauchen!

```
>>> turm_a.append(55)
>>> turm_a
[55]
```

Auch diese Turm-Klasse kennt die Listen-Methoden.

Nun wäre es schön, wenn uns diese Türme bekannt geben könnten, wo ihre »Spitze« ist, das heißt, wo die nächste Scheibe hinkommen muss. Vielleicht mittels einer Methode `top()`. Diese Methode sollte ein Tupel von zwei Koordinaten ausgeben:

die x-Koordinate des Turmes selbst und als y-Koordinate den Wert des Ausdrucks (y des Bodens) - Scheibenabstand * (Höhe des Turms).

Dies können wir erreichen, wenn wir bei der Konstruktion des Turms zusätzlich noch den Scheibenabstand in einem Attribut des Turms speichern.

Aus dieser Idee ergibt sich folgender Code für die Klasse `Turm`:

```
class Turm(list):
    def __init__(self, turm_x, boden_y, abstand):
        self.x = turm_x
        self.y = boden_y
        self.h = abstand
    def top(self):
        return self.x, self.y - len(self)*self.h
```

Beachte, wie schlau Türme mittels der Funktion `len` ihre eigene (`self`!) Höhe ermitteln können.

➔ Füge diesen Code in die Datei `hanoig.py` ein.

Die Türme sollen nun nicht mehr bloße Listen sein, sondern Objekte der Klasse `Turm`, also Listen mit zusätzlichen Eigenschaften und Fähigkeiten. Sie müssen daher mit dem Konstruktor der Klasse `Turm` erzeugt werden:

```
turm_a = Turm( 80, 190, 15)
turm_b = Turm(220, 190, 15)
turm_c = Turm(360, 190, 15)
```

➔ Ändere die Konstruktion der Turm-Objekte entsprechend, speichere das Programm und führe es aus. Es sollte dasselbe Ergebnis produzieren wie vorher.

Was uns zu tun bleibt, ist die Änderung der Funktion `zug`. Das Problem der Ermittlung der Koordinaten können wir nun leicht lösen. (Doch wenn immer dir etwas unklar ist, denke daran, eventuell den IPI zu fragen, um dir Klarheit zu verschaffen.)

```
Funktion zug( von_turm, nach_turm):
    scheibe ----> von_turm.pop()
    x1, y1 ----> von_turm.top()
    x2,y2 ----> nach_turm.top()
    scheibe.schiebe_nach( x1, 20)
```

```
        scheibe.schiebe_nach( x2, 20)
        scheibe.schiebe_nach( x2, y2)
        nach_turm.append(scheibe)
```

Nun, das ist ja schon 98 % Python-Code.

➔ Codiere die Funktion `zug`. Die `print`-Anweisung von früher kann entfallen.

➔ Entkommentiere (falls nötig) den Aufruf von `versetze` und setze als Argumente die Türme `turm_a`, `turm_b` und `turm_c` ein.

➔ Setze im Aufruf von `hanoi` im »Hauptprogramm« für `n` den Wert 3 ein und führe das Programm aus.

Ich nehme an, du bist jetzt ziemlich erschöpft, aber glücklich. Das Programmieren der grafischen Darstellung war doch ein wesentlich höherer Aufwand als das Programmieren des Algorithmus für die Lösung des Spiels alleine.

Über die Türme von Hanoi geht die Mär, dass in einem Kloster in der Nähe von Hanoi einige Mönche damit beschäftigt seien, einen Turm von 100 Scheiben von einem Stock auf einen anderen zu verschieben. Es wurde ihnen geweissagt, dass die Welt untergehe, wenn sie mit dieser Arbeit fertig wären. Sie arbeiten ziemlich schnell. Aber sie haben keine Angst.

Dir rate ich, dir eine Erfrischung zu besorgen und den Computer eine Zeit lang Computer sein zu lassen. Das kannst du beispielsweise so machen, dass du `hanoi` für zehn oder elf Scheiben aufrufst. Während der Erholungspause wirf ab und zu mal einen Blick auf den Bildschirm. Sollte dir doch irgendwann fad werden, könntest du überlegen, wie viele Züge notwendig sind, um einen Turm von zehn Scheiben zu versetzen, oder einen Turm von elf Scheiben.