B3 Rekursion oder: »Weniger vom Gleichen«

Inhalt

Einleitung	1
Salami schneiden	2
Salami-Simulation	4
Spiralen	8
Treppen	12
Bäume	14
Lass 100 Bäumchen wachsen	17
Rekursive Funktionen mit Wert: fak & ggt	23
Türme von Hanoi	26
Zusammenfassung	32
Einige Aufgaben …	32
… und eine Frage	32
Hinter den Kulissen	33

Einleitung

In diesem Kapitel lernst du ein sehr allgemeines Verfahren kennen, um bestimmte Probleme zu lösen. Das Verfahren heißt Rekursion. Die Probleme, die du damit lösen kannst, sind solche, die eine »Ähnlichkeit mit sich selbst« haben. Mir ist klar, dass du mit dieser Beschreibung momentan nichts anfangen kannst. Ich bin aber sicher, dass das nach dem Durcharbeiten dieses Kapitels ganz anders ist.

In diesem Kapitel …

- … lernst du, dass du mit Rekursion gewisse Probleme sehr leicht lösen kannst, die mit anderen Verfahren nur sehr schwer lösbar sind.
- … begegnest du mehreren Fragestellungen der Informatik, die ganz grundlegend sind.
- … wirst du einige sehr nette Grafiken erstellen.
- … und an manchen Stellen aus dem Staunen nicht heraus kommen.

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

Salami schneiden

In diesem Abschnitt möchte ich dir erklären, was Rekursion ist. Oder genauer: wie ein gegebenes Problem durch Rekursion gelöst werden kann und was den Unterschied zu herkömmlichen Lösungsverfahren ausmacht.

Da es hier noch nicht in erster Linie um Programmieren geht, sondern eher um das Ausdenken eines Algorithmus, also einer Lösungsvorschrift, nehmen wir uns kein Programmierproblem, sondern ein Problem aus dem Alltagsleben. Eine Salami ist in gleich dicke Scheiben zu schneiden.

```
Algorithmus: Schneide-iterativ(Salami):
    Solange Salami lang genug ist:
        Schneide eine Scheibe ab.
    Lege das letzte Ende dazu.
```

Da bekannt ist, dass immer wieder dasselbe gemacht werden muss, hat sich hier eine `while`-Schleife angeboten.

Auf genauso natürliche und einsichtige Weise kann ich eine andere Problemlösung formulieren:

```
Algorithmus: Schneide(Salami):
    Wenn Salami lang genug ist:
        Schneide eine Scheibe ab.
        Schneide(kürzere Salami)
    Sonst:
        Lege das letzte Ende dazu
```

Das Merkwürdige an dieser Erklärung ist, dass das, was erklärt werden soll – nämlich wie man eine Salami in Scheiben schneidet (1. Zeile der Beschreibung) – in der Erklärung selbst vorkommt (4. Zeile der Beschreibung). Die Problemlösung bezieht sich also auf sich selbst. Allerdings wird sie dabei naturgemäß auf eine kürzere Salami angewendet, als ursprünglich, denn gerade vorher wurde ja eine Scheibe abgeschnitten.

Wenn dir das schwer verständlich ist, versetze dich gedanklich – oder meinetwegen auch wirklich – in die Situation vor einer Party. Du stehst mit deiner Freundin in der Küche. Sie drückt dir eine Salami in die Hand und sagt:

»Salami schneiden!«

»Wie geht das?«

»Wenn noch genug Salami da ist, schneide eine Scheibe ab und dann ...«

»... und dann?« Du stehst vor derselben Situation wie am Anfang, mit dem kleinen, aber nicht unwesentlichen Unterschied, dass die Salami jetzt kürzer ist.

»Salami schneiden!«

Wirst du nochmals »Wie geht das?« fragen? Sie hat es dir doch gerade erklärt. Und du erinnerst dich auch noch genau daran: »Wenn noch genug Salami ...«

So arbeitest du ohne zu fragen weiter. Scheibe abschneiden – Salami schneiden, d.h. Scheibe abschneiden, Salami schneiden ... Nimmt das Salamischneiden je ein Ende? Ja, irgendwann ist die Salami so kurz, dass du keine Scheibe mehr abschneiden kannst! Du legst das Restchen zu den übrigen Scheiben – und servierst den Salami-Aufschnitt!

Problemlösungen, die sich in dieser Weise auf sich selbst beziehen, nennt man rekursiv. Damit solche rekursiven Lösungen funktionieren, müssen zwei Bedingungen erfüllt sein.

1. Es muss eine Situation geben, die einfach gelöst werden kann. Diese Situation nennt man den Basisfall der Rekursion. Im Salami-Beispiel ist der Basisfall dann gegeben, wenn die Salami zu kurz ist, um noch eine Scheibe abzuschneiden.
2. In den anderen Fällen wird die Problemlösungsvorschrift rekursiv neuerlich angewendet. Daher nennt man dies auch den rekursiven Fall. Im rekursiven Fall wird dieselbe Lösungsvorschrift (»Salami schneiden!«) auf »Weniger vom Gleichen« (nämlich eine kürzere Salami) angewendet.

Nachdem wir die Sache jetzt durchdacht haben, kommt es mir vor, dass es bei der Formulierung des rekursiven Algorithmus klarer ist, den Basisfall zuerst anzuführen. Es wird dann gleich in der ersten Zeile klargestellt, wie das Ende aussehen muss. Damit schaut die – mit der ersten im Übrigen gleichwertige - Beschreibung des Algorithmus so aus.

```
Algorithmus: Schneide(Salami):
    wenn die Salami dünner als eine Scheibe ist:
        lege das letzte Ende zu den übrigen Scheiben
    sonst:
        Schneide eine Scheibe ab.
        Schneide(kürzere Salami)
```

Salami-Simulation

Damit du siehst, was diese beiden Party-Vorbereitungs-Algorithmen mit Programmieren zu tun haben, werden wir sie nun mit Python simulieren.

Dazu stellen wir uns die Salami durch ihre Länge beschrieben vor: also durch eine Zahl. Für die Dicke der Scheiben wählen wir 1 – was vielleicht nicht sehr realistisch, aber dafür leicht zu durchschauen ist.

Natürlich müssen wir den Funktionen, die wir jetzt schreiben werden, die (Länge der) Salami als Argument übergeben. Im Übrigen können wir die obigen Beschreibungen der beiden Algorithmen schon als Programmentwürfe verwenden:

➔ Öffne ein neues Editor-Fenster, schreibe den Kopfkommentar für ein Programm `salami.py`, speichere es ab und gib folgenden Code ein:

```
def schneiden_iterativ(salami):
    while salami > 1:
        salami = salami - 1
        print 1,  # Scheibe abgeschnitten!
    print salami  # Das letzte Ende
```

➔ Speichere das Programm ab und führe es aus! Nun kannst du den IPI iterativ Salami schneiden lassen:

```
>>> salami = 7.23
>>> schneiden_iterativ(salami)
1 1 1 1 1 1 1 0.23
>>>
```

Sieben Scheiben und ein kleines Fuzzerl!

➔ Füge darunter den Code für die rekursive Fassung an:

```
def schneiden(salami):
    if salami < 1:       # Basisfall
        print salami     # Das letzte Ende
    else:                # rekursiver Fall
        print 1,
        schneiden(salami-1)
```

Typisch für rekursive Programme ist hier,

1. dass zwischen dem Basis-Fall und dem rekursiven Fall mittels einer Verzweigungs-Anweisung unterschieden wird.
2. dass der Parameter beim rekursiven Aufruf im Vergleich zum ursprünglichen einem »kleineren Problem« entspricht. Damit führen wir die Lösung auf »weniger vom Gleichen« zurück. Dadurch wird sichergestellt, dass nach einer endlichen Zahl von rekursiven Aufrufen der Basis-Fall erreicht wird.

➔ Speichere das Programm ab, führe es aus und prüfe nun mit dem IPI das rekursive Salamischneiden.

```
>>> salami = 7.23
>>> schneiden(salami)
1 1 1 1 1 1 1 0.23
>>>
```

Das gleiche Ergebnis!

Wir sehen uns jetzt noch gemeinsam den Ablauf des Funktionsaufrufs `schneiden(salami)` im Debugger an. Dies erhellt dir vielleicht noch ein Stück die Arbeitsweise rekursiver Funktionen.

➔ Wähle im Python-Shell-Fenster DEBUG|DEBUGGER. Das Debugger-Fenster geht auf. Stelle sicher, dass die Option SOURCE angehakt ist.

➔ Ordne das Python-Shell-Fenster, das Debugger-Fenster und das Editor-Fenster mit dem Quellcode von `schneiden` so auf dem Bildschirm an, dass du alle drei Fenster ungehindert siehst.

Solltest du in der folgenden Übung den Faden verlieren, kannst du sie an jeder Stelle abbrechen, indem du im Debugger auf QUIT klickst. Kehre dann genau hierher zurück, um von neuem zu beginnen.

➔ Starte im IPI den Debugger:

```
>>>
[DEBUG ON]
>>> schneiden(7.23)
```

➔ Klicke 2 Mal auf STEP.

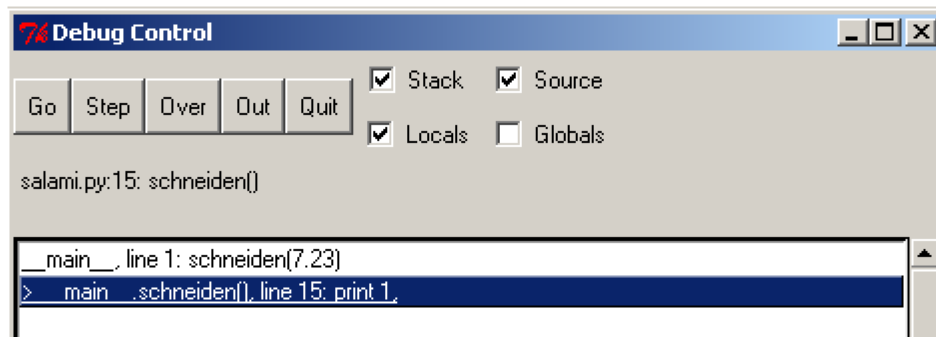
Das führt zum Aufruf von `schneiden(salami)` mit einem Wert von `salami` von `7.23`, wie im Abschnitt LOCALS des Debugger-Fensters zu sehen ist.

➔ Klicke auf STEP.

Liegt der Basisfall vor? (Wirf immer wieder einen Blick in das Fenster mit dem Quellcode.)

➔ Klicke auf STEP.

Nein, du gelangst in den rekursiven Fall:



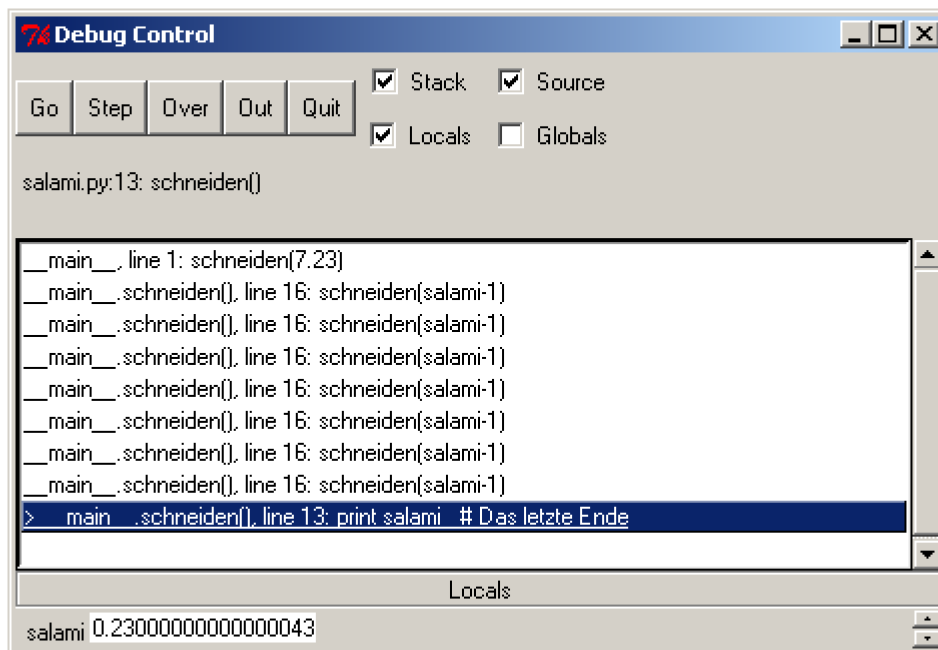
Damit du jetzt nicht in die Tiefen des Quellcodes der IDLE entführt wirst, womit du sicherlich nichts anfangen könntest, klicke jetzt *unbedingt* auf OVER!

Damit wird `print 1` einfach ausgeführt, der IPI schreibt 1 in sein Fenster und der erste rekursive Aufruf von `schneiden` steht an.

➔ Klicke auf STEP!

Jetzt beginnt die zweite Ausführung von `schneiden`. Dementsprechend hat `salami` jetzt den Wert `6.23`.

➔ Bevor du weitermachst: Sieh dir genau das unten stehende Bild an. Du musst das Programm nun so weit ausführen, bis das Debugger-Fenster so aussieht. Folge dazu den *darunter* stehenden Anweisungen.

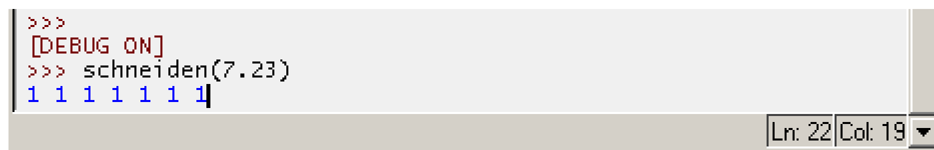


➔ Klicke 2 Mal auf STEP.

➔ Nun – und in der Folge stets, wenn die Anweisung `print 1` auszuführen ist, klicke auf OVER.

➔ Fünf Folgen von STEP-STEP-STEP-OVER und danach noch drei einzelne STEP sollten dich zu dieser Situation im Debugger-Fenster hinführen.

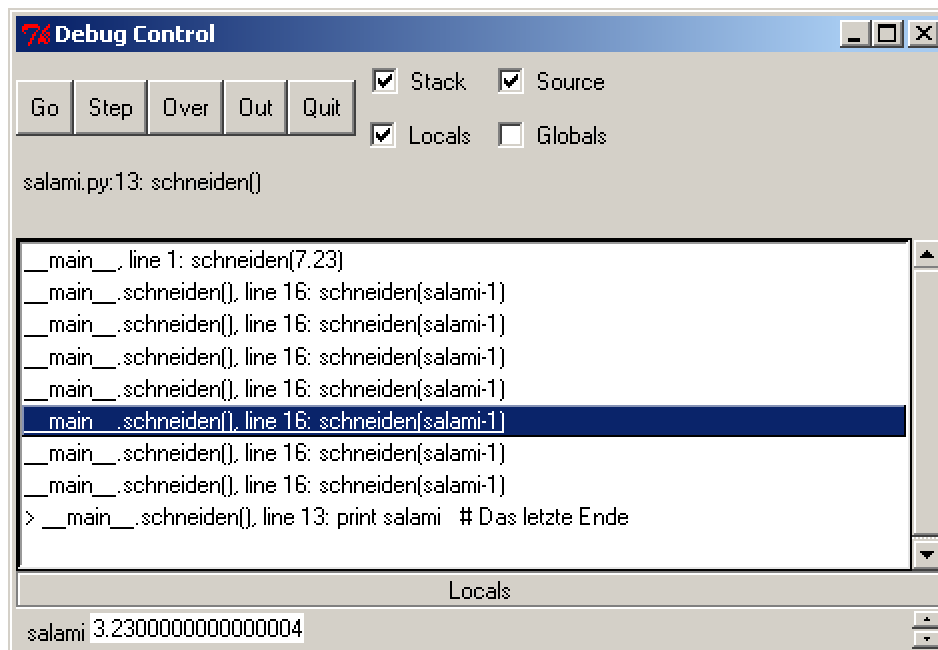
Von der Salami sind mittlerweile sieben Scheiben abgeschnitten worden:



Die Programmausführung ist nun im Basis-Zweig. Das liegt natürlich daran, dass `salami` nur mehr `0.23` ist.

➔ Klicke auf das Debugger-Fenster und bewege mit den Auf-/Ab-Cursortasten die Zeilenmarkierung.

Du siehst hier, dass im Augenblick acht aufeinander folgende Aufrufe von `salami_schneiden_rekursiv` aktiv sind. Jeder Aufruf hat sein eigenes lokales Wörterbuch mit einem Eintrag: `salami` – im Debugger-Fenster unten zu sehen. Doch in jedem Aufruf hat `salami` einen anderen Wert, insgesamt alle Werte von 7.23 herunter bis 0.23.



➔ Klicke auf OVER.

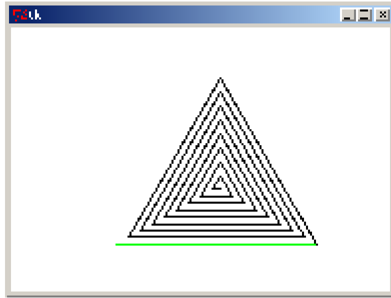
➔ Schließe das Debugger-Fenster.

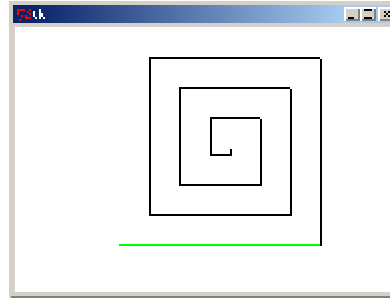
Der Programmaufruf wird mit der letzten `print`-Anweisung beendet. Die Programmausführung springt 8 Mal in die aufrufende Funktion zurück, doch dort ist nichts mehr zu tun, weil der rekursive Aufruf stets die letzte Anweisung in der Funktion war. Man spricht in einem solchen Fall von einer so genannten Endrekursion. Du wirst besser verstehen, wie das gemeint ist, wenn wir unser erstes nicht endrekursives Programm geschrieben haben.

Wie so oft, wenn du etwas Neues lernst, wurde dir hier zunächst ein besonders einfacher Fall serviert – so einfach, dass dabei die Methode noch nicht ihre großen Vorzüge ausspielen kann. Die kommen etwas weiter unten raus! Doch vorher machen wir noch ein einfaches grafisches Programm. Es soll dich lehren, rekursive Strukturen zu *sehen*.

Spiralen

Gesucht ist ein Turtle-Grafik-Programm, das folgende Zeichnungen von Spiralen erstellen kann:





Der erste Eindruck ist, dass das irgendwie auch mit unserer Funktion `polyschritt` gehen muss, doch mit der Besonderheit, dass die Strecken, die die Turtle zurücklegt, immer kürzer werden.

Bevor wir darüber genauer nachdenken, schauen wir uns einmal die beiden Spiralmuster an. Ich habe jeweils die erste Linie in einer anderen Farbe – im Druck heller – gezeichnet. Wenn du dir diese ersten Linien wegdenkst, verbleiben in beiden Fällen ganz gleichartige Spiralen – doch haben sie eine Strecke weniger und ihre erste Strecke ist kürzer als bei der ganzen Spirale – also: »Weniger vom Gleichen«. Wir formulieren unsere Idee:

```
Spirale zeichnen:
Parameter: laenge (der 1. Strecke)
      Strecke mit laenge zeichnen
      Turtle drehen
      Spirale mit (laenge-10) für die 1. Strecke zeichnen
```

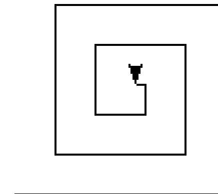
Dabei wird `laenge` immer kleiner. Und – ah, ja! – wir müssen dafür sorgen, dass die Rekursion einmal aufhört. Einen Basisfall erfinden. Das ist hier nicht schwer: Wenn `laenge` nicht mehr positiv ist – aufhören! Das tragen wir noch in den Entwurf ein:

```
Spirale zeichnen:
Parameter: laenge (der 1. Strecke)
     wenn laenge <= 0 ist:
         zurückkehren (zum Aufrufer!)
     sonst:
         Strecke mit laenge zeichnen
         Turtle drehen
         Spirale mit (laenge-10) für die 1. Strecke zeichnen
```

Manche Dinge sind ziemlich spannend und man hat wenig Lust, allzu gründlich darüber nachzudenken, sondern will sie gleich ausprobieren. Geht es dir gelegentlich auch so? Ich hab jetzt Lust, diese Idee gleich in einer ganz einfachen Form dem IPI vorzuwerfen. Sagen wir nur für »Quadratspiralen«. Wenn das einmal gelingt, können wir die Sache weiterentwickeln!

➔ Mach mit!

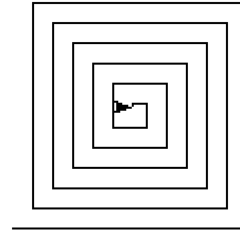
```
>>> from turtle import *
>>> reset()
>>> def quspirale(laenge):
        if laenge < 0:
                return
        else:
                forward(laenge)
                left(90)
                quspirale(laenge-10)

>>> quspirale(105)
```



Funktioniert!! Erzeugt das daneben gezeigte Bild.

Jetzt will ich, dass die Spirale enger wird: Drücke so oft die Tastenkombination Alt+P, bis die Kopfzeile der Funktionsdefinition von `quspirale` wieder erscheint, und ändere in der letzte Zeile des Funktionskörpers `10` zu `5` ab:

```
>>> def quspirale(laenge):
        if laenge < 0:
                return
        else:
                forward(laenge)
                left(90)
                quspirale(laenge-5)
```



```
>>> reset();quspirale(117)
```

Denke daran, je nach Bedarf `tracer` aus- beziehungsweise wieder einzuschalten. Für schnelle Zeichnungen: `tracer(0)`. Um gedanklich nachzuvollziehen, warum was geschieht: `tracer(1)`.

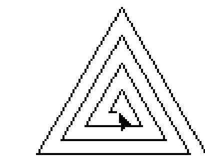
Ich denke, ich sollte für die Differenz zwischen zwei aufeinander folgenden Längen auch einen Parameter vorsehen, dann kann ich von Aufruf zu Aufruf einen anderen Wert dafür einsetzen. Und damit ich auch Dreiecksspiralen und andere erzeugen kann, auch für den Winkel. Es ist so weit:

➔ Öffne ein neues Editor-Fenster, erzeuge eine Datei `rekursion.py` (Kopfkommentar nicht vergessen!) und gib folgenden Code ein:

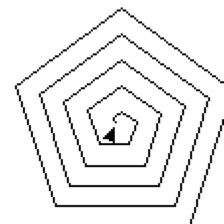
```
from turtle import *
from mytools import polyschritt

def spirale(laenge, winkel, diff = 10):
    if laenge < 0:      # Basisfall
        return
    else:               # rekursiver Fall
        polyschritt(laenge, winkel)
        spirale(laenge-diff, winkel, diff)
```

➔ Speichern, ausführen und im IPI ausprobieren!

```
>>> reset();spirale(100,120,8)
```



```
>>> reset();spirale(75,72,3)
```



Findest du es lästig, vor jedem `spirale`-Aufruf `reset()` hinschreiben zu müssen? Vielleicht sollten wir das als erste Anweisung in unser `spirale`-Programm hineinnehmen? Probiere das aus:

➔ Ändere die Funktion `spirale` so, dass du als erste Anweisung `reset()` einfügst.

➔ Teste die geänderte `spirale`. Was geht vor?

War doch keine so gute Idee. Bei jedem rekursiven Aufruf wird jetzt das Turtle-Grafik-Fenster gelöscht. Dazu sollten wir uns merken:

Warnung! Code, der vor der Ausführung einer rekursiven Funktion nur einmal ausgeführt werden soll, darf *niemals in die rekursive Funktion hineingeschrieben* werden! Sonst würde er nämlich bei jedem rekursiven Aufruf ausgeführt werden.

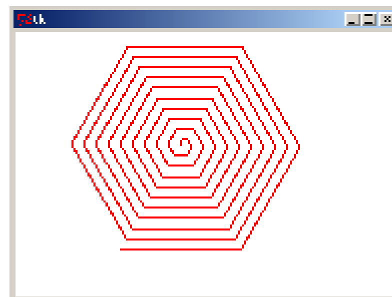
Stattdessen ist es oft sinnvoll, eine passende Startfunktion zu schreiben, die gleich ein paar Dinge auf einmal erledigt. Zum Beispiel so was:

```
from mytools import polyschritt, hopp

def start(farbe = "black", trace=0):
    reset()
    tracer(trace)
    hopp(120,225) # schickt Turtle nach rechts unten
    width(2)      # damit sie etwas dicker zeichnet
    color(farbe)  # Farbe gefällig?
```

➔ Füge den Code dieser Funktion in die Datei `rekursion.py` ein. Sichere sie und führe sie aus.

➔ Nun ist der Weg frei für so etwas wie:

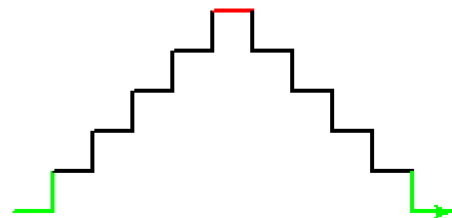
```
>>> start("red")
>>> spirale(120, 60, 2)
```



`spirale` ist ein endrekursives Programm. Daher lässt es sich anstatt mit Rekursion leicht auch mit einer Schleife programmieren:

Treppen

Neben Spiralen gibt es noch viele andere grafische Figuren, bei denen ein Teil dem ursprünglichen Ganzen ähnlich ist, zum Beispiel Treppen.



Gesucht ist also nun eine Funktion, die eine Treppe zeichnet, die n Stufen hinauf und wieder hinunter führt, wie sie nebenstehend abgebildet ist. In dem Bild ist n = 5.

Die erste Frage ist, was ist rekursiv an dieser Grafik? Diese Treppe aus fünf Stufen besteht aus einer Stufe hinauf, dann wird eine Treppe mit vier Stufen gezeichnet – »Weniger vom Gleichen« – und schließlich wieder eine Stufe abwärts.

Und was ist der Basisfall, die einfachstmögliche Treppe? Typisch mathematisch gedacht ist das die Treppe mit 0 Stufen. Ich habe sie in der Zeichnung rot gekennzeichnet (heller!): Sie besteht einfach aus einer Strecke, die so lang ist wie die Stufenbreite.

Damit ergibt sich folgende Idee für die Funktion `treppe`:

```
Funktion treppe:
Parameter: breite (Breite der Stufen)
           n (Anzahl der Stufen)
    wenn n gleich 0 ist:
        gerade Strecke mit breite zeichnen
    sonst:
        Stufe hinauf (breite)
        treppe (breite,  n-1 Stufen)
        Stufe hinunter (breite)
```

Natürlich muss dabei noch genau geklärt werden, was die Teile »Stufe hinauf« und »Stufe hinunter« sind. Ich schlage dir vor, das durch eigene kleine Funktionen zu lösen, denen nur die `breite` als Argument übergeben wird. Wichtig ist dafür, dass du darauf achtest, dass nach jeder Stufe (ob hinauf oder hinunter) die Turtle wieder nach rechts schaut, genau wie in der Ausgangslage.

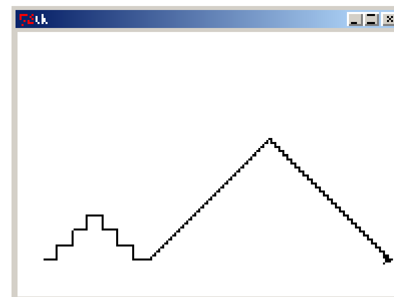
Dann wird »Stufe hinauf« so auszuführen sein:

```
Stufe hinauf (breite):
    breite nach vor gehen
    90 Grad nach links drehen  # Turtle schaut nach oben
    breite nach vor gehen
    90 Grad nach rechts drehen
```

➔ Entwirf auf ähnliche Weise den Code-Abschnitt Stufe hinunter.

➔ Versuche nun an Hand dieser Teile des Programmentwurfs die Funktion `treppe` mit ihren beiden Hilfsfunktionen `stufe_hinauf`, `stufe_hinunter` (in der Datei `rekursion.py`) zu programmieren.

➔ Teste deinen Code: Prüfe zuerst, dass die beiden Hilfsfunktionen genau das machen, was geplant ist. Dann probiere aus, ob `treppe(20,0)` richtig funktioniert. Fahre fort mit `treppe(20,1)`. Sollte etwas am Code nicht stimmen, muss es sich schon bis hierher gezeigt haben.



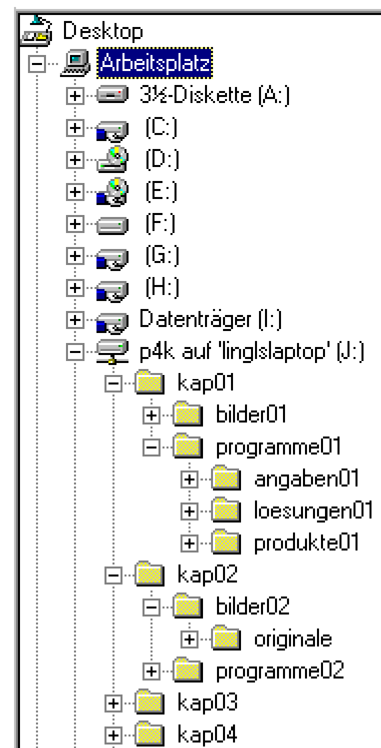
Kann deine `treppe`-Funktion nebenstehende Abbildung erzeugen? Eine vollständige Lösung findest du, wie immer, auf der Buch CD. Eine vereinfachte Lösung wird in der folgenden Übung benutzt.

Wenn dir die Arbeitsweise dieser rekursiven `treppe`-Funktion noch geheimnisvoll vorkommt, so lade ich dich ein, einen Blick hinter die Kulissen zu tun – im letzten Abschnitt dieses Kapitels. Ich zeige dort, wie du mit dem Debugger dieses Geheimnis lüften kannst.

Muster 12: Definition einer rekursiven Funktion:

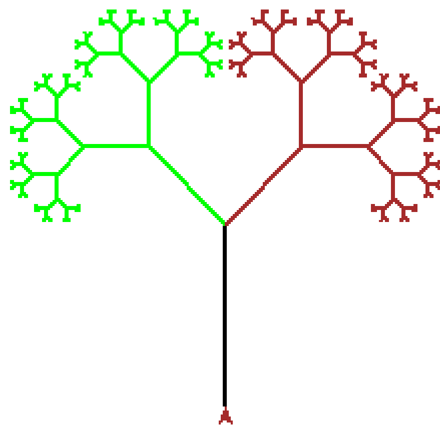
`def` ***`Funktionsname`***`(`*`parm1, ...`*`):`	[Funktionskopf]
`if` *`basisfall`*`:`	[Abbruchbedingung]
`Anweisungen`	
`else:`	[rekursiver Fall]
`Anweisungen...`	[können fehlen]
`Funktionsname``(`*`param1', ... `*`)`	[**rekursiver Aufruf, mit »kleineren« Parametern**]
`Anweisungen...`	[... können fehlen]
`...`	[**eventuell weitere rekursive Aufrufe**]

Bäume



Bäume kommen in der Informatik so häufig vor, dass man oft den Wald vor lauter Bäumen nicht sieht. Hier rechts hast du zum Beispiel einen Verzeichnisbaum.

Alle diese Bäume haben ähnliche Eigenschaften. Wir wollen uns hier aber nur damit beschäftigen, Bäume zu *zeichnen*. Etwa so:



Woraus besteht denn *der ganze Baum*? Hast du schon einen Blick für rekursive Strukturen? Wenn ja, dann musst du aber ziemlich gut drauf sein! Normalerweise muss man einiges an Erfahrung sammeln, um solche Strukturen selbstständig aufzufinden. Deshalb beschreibe ich dir hier, wie ich diesen Baum sehe:

```
Der ganze Baum:
    Stamm
    nach links ein kleinerer Baum
    nach rechts ein kleinerer Baum
```

Ich habe meine Sichtweise übrigens wieder einmal durch verschiedene Farben (im Druck: Grauwerte) unterstrichen.

Das ist zwar noch kein Programmentwurf, aber immerhin eine Programmidee. Diesmal haben wir zwei Mal »Weniger vom Gleichen«.

Irgendwie schaut die Aufgabe ideal nach Turtle-Grafik aus. Bevor wir die Idee etwas genauer als Programmentwurf formulieren, müssen wir uns überlegen: Durch welche Daten wird die Gestalt des Baums festgelegt? Wir beobachten, dass die Äste der beiden kleineren Bäume etwa halb so groß sind wie die des großen Baums. Wir

können als Angabe die Länge des Stamms nehmen und für die kleineren Bäume dann das 0,5fache dieser Länge einsetzen.

Ein anderes Kennzeichen ist der Winkel zwischen den Zweigen. In dem abgebildeten Beispiel ist der Winkel 90 Grad. Wir übernehmen einmal diesen fixen Wert in unseren Programmentwurf. Später können wir noch einmal darüber nachdenken, diesen Winkel variabel zu gestalten.

Unsere Idee besteht darin, nach dem Zeichnen des Stammes einen kleineren Baum zu zeichnen, das heißt einen mit der halben Stammlänge. Das kann natürlich nicht endlos so weitergehen. Vereinbaren wir Folgendes: Wenn die Stammlänge kleiner als 2 ist, wird kein weiterer kleinerer Baum angefügt. (Das wird unser Basisfall.)

Somit gelangen wir zu folgendem Entwurf:

```
funktion baum (stamm):
    wenn stamm < 2:
        zurückkehren
    Strecke der Länge stamm zeichnen
    # linker Teilbaum:
    Turtle um 45° nach links drehen
    baum ( stamm / 2 ) # »Weniger vom Gleichen«
    Turtle um 45° nach rechts drehen
    # rechter Teilbaum:
    Turtle um 45° nach rechts drehen
    baum (stamm / 2) # noch einmal
    Turtle um 45° nach links drehen
    # und zurück an den Ausgangspunkt!
    Turtle um stamm rückwärts bewegen
```

Ganz wichtig ist hier die letzte Anweisung, die in unserer ersten Programmidee noch nicht vorgekommen ist: Sie bringt die Turtle am Ende der Baumzeichnung an ihren Ausgangspunkt, den Anfang des Stammes. Das ist deshalb so wichtig, denn nach einem ersten Baum soll ja *genau von dort aus* ein zweiter Baum gezeichnet werden.

➔ Führe diesen rekursiven Algorithmus mit Papier und Bleistift aus. Nimm kariertes Papier, nimm als Längeneinheit die Länge eines Kästchens, versetze dich gedanklich in die Turtle und zeichne `baum(12)` auf.

Funktioniert es auf dem Papier, so sollte es auch als Programm funktionieren. Der Programmentwurf ist schnell in Python codiert:

➔ Öffne in der IDLE ein neues Editor-Fenster. Bereite es zum Schreiben des Programms `baum01.py` vor. (Kopfkommentar und so weiter ...)

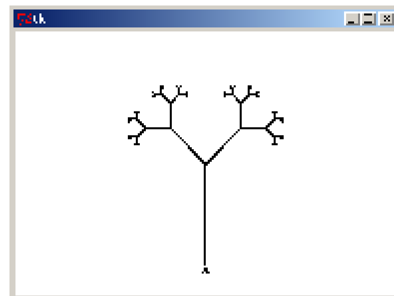
➔ Füge den folgenden Code der Funktion `baum()` ein.

```
from turtle import *

def baum(stamm):
    if stamm < 2:
        return  # Rücksprung in die aufrufende Funktion
    forward(stamm)
    left(45)
    baum(0.5 * stamm)
    right(45)
    right(45)
    baum(0.5 * stamm)
    left(45)
    backward(stamm)
```

➔ Speichere das Programm und führe es aus.

➔ Teste die Funktion `baum` im IPI:

```
>>> from mytools import hopp
>>> left(90)
>>> hopp(-100)
>>> baum(100)
>>>
```



Wieder ein Erfolgserlebnis. Es ergibt sich nebenstehendes Bild:

Was haben wir erreicht? Es ist uns gelungen, eine ganz schön komplizierte Grafik mit einer Funktion zu erstellen, die aus nur acht Anweisungen besteht. Zwei davon sind allerdings rekursive Aufrufe.

Spätestens jetzt wirst du wohl überzeugt sein, dass Rekursion für manche Probleme eine sehr leistungsfähige Methode ist. Wenn nicht, dann versuche eine Funktion zu schreiben, die dieses Bäumchen ohne rekursive Aufrufe erzeugt – und dann noch so vielseitig weiterzuentwickeln ist, wie du es im Folgenden sehen wirst.

Lass 100 Bäumchen wachsen!

Unser einfaches Bäumchen aus dem vorigen Abschnitt können wir nun als Ausgangspunkt für eine regelrechte Evolution von Bäumen nehmen.

Man kann an der Funktion `baum` nämlich allerhand ändern, hinzufügen und variabel gestalten.

Beispiel 1: Betrachten wir den Faktor, um den sich der Stamm beim »kleineren Baum« verkürzt. Dieser Faktor ist in unserer ersten `baum`-Funktion 0.5. Man könnte ebenso gut einen kleineren oder größeren wählen. Wenn wir diesen Faktor als Argument übergeben, wird die Funktion `baum` bereits Bäume verschiedener Gestalt erzeugen:

➔ Erzeuge aus `baum01.py` das neue Programm `baum02.py`.

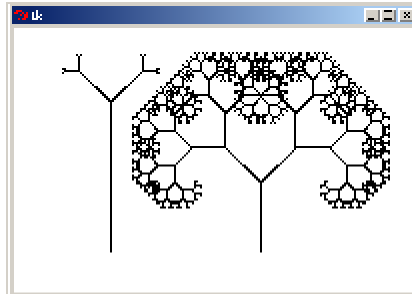
➔ Ändere den Code von `baum` wie folgt ab:

```
def baum(stamm, faktor):
              # faktor ist der Verkürzungsfaktor
    if stamm < 2:
        return
    forward(stamm)
    left(45)
    baum(faktor*stamm, faktor)
    right(90)                          # 2 * 45 = 90
    baum(faktor*stamm, faktor)
    left(45)
    backward(stamm)
```

Beachte: Das Argument, das für den ersten Parameter eingesetzt wird, ergibt sich jetzt durch Multiplikation mit `faktor` an Stelle der fixen Zahl 0.5. Da aber unsere neue Funktion `baum` jetzt zwei Parameter hat, müssen beim Funktionsaufruf von `baum` auch zwei Argumente übergeben werden. Doch das zweite Argument ändert sich nicht, es ist einfach wieder `faktor`.

Setzt man nun bei einem Aufruf von `baum` für Faktor eine kleine Zahl ein, dann verkürzen sich die Äste des Baumes rasch, setzt man eine große Zahl (aber natürlich kleiner als 1!) ein, dann verkürzen sich die Äste nicht so rasch. Sehen wir uns das an:

➔ Zeichne durch geeignete Aufrufe von `baum` zwei Bäume ins Turtle-Grafik-Fenster. Natürlich musst du vor jedem Aufruf die Turtle an eine passende Stelle im Fenster führen, wo der Baum beginnen soll, und nach oben orientieren!



baum(150, 0.3) und *baum(70, 0.7)*

Wir erkennen auch, dass wegen des größeren Wertes von `faktor` mehr rekursive Aufrufe erforderlich sind, bis das Argument kleiner als 2 geworden ist und somit der Basisfall erreicht ist.

Für den ersten Baum:

```
>>> 150*0.3
45.0
>>> 45.0*0.3
13.5
>>> 13.5*0.3
4.0499999999999998
>>> 4.0499999999999998*0.3
1.2149999999999999    # schon kleiner als 2!!!
```

Es werden also nur drei rekursive Aufrufe wirksam, dann kommt der Basisfall, bei dem nichts gezeichnet wird.

Für den zweiten Baum wird die Grenze 2 erst beim zehnten rekursiven Aufruf unterschritten, denn:

```
>>> 70*(0.7**9)
2.8247524899999985   # noch größer als 2
>>> 70*(0.7**10)
1.9773267429999986   # bereits kleiner als 2
```

Daher entsteht so ein dichter Baum!

Wir sagen: Beim ersten Baum wird die *Rekursionstiefe* 4 erreicht, beim zweiten Baum die Rekursionstiefe 10.

Beispiel 2: In vielen Fällen möchte man die Rekusionstiefe direkt kontrollieren, indem man beispielsweise angibt, dass die Rekursion nach fünf Stufen abbrechen soll.

Dies erreicht man leicht, indem man einen eigenen Parameter für die Rekursionstiefe einführt, dessen Wert bei jedem rekursiven Aufruf um 1 vermindert wird. Der Basisfall ist erreicht, wenn dieser Parameter den Wert 0 hat:

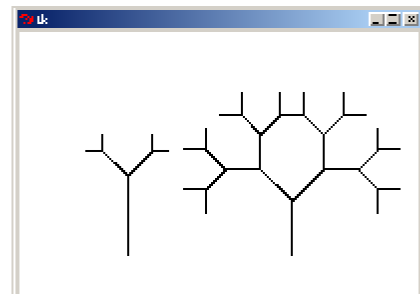
➔ Erzeuge ausgehend von `baum02.py` das Programm `baum03.py` mit folgender `baum`-Funktion:

```
def baum(stamm, faktor, tiefe):
        # tiefe ist die gewünschte Rekursionstiefe
    if tiefe == 0:
        return
    forward(stamm)
    left(45)
    baum(faktor*stamm, faktor, tiefe - 1)
    right(90)
    baum(faktor*stamm, faktor, tiefe - 1)
    left(45)
    backward(stamm)
```

Abgeändert wurde:

- die Parameterliste im Funktionskopf
- die Argumentlisten in den Funktionsaufrufen, wobei die Tiefe immer um 1 vermindert wird, so dass sie schließlich 0 erreichen muss
- die Bedingung für den Basisfall. Sie ist jetzt `tiefe==0`.

➔ Spiele mit dieser neuen Baumfunktion, um ihre Eigenschaften zu erforschen. Ein Beispiel siehst du wieder in nebenstehendem Bild:



```
>>> baum(80,0.45,3)
>>> baum(55, 0.8, 5)
```

Hier erkennst du: Du kannst die Rekursionstiefe direkt am Baum abzählen, indem du außen entlang fährst und die Anzahl der Äste mitzählst.

Beispiel 3: Warum müssen eigentlich die Äste immer einen Winkel von 90 Grad miteinander bilden? Wollen wir diesen Winkel variabel gestalten, müssen wir ihn ebenfalls als Argument übergeben können. Also ist ein neuer Parameter `winkel` angesagt.

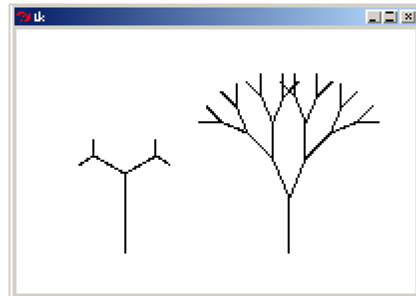
Wo wird der Wert von `winkel` im Code gebraucht? An drei Stellen, wo sich die Turtle drehen muss. Statt `left(45)` müssen wir dann `left(0.5*winkel)` schreiben und `right(90)` muss durch `right(winkel)` ersetzt werden.

➔ Erzeuge ausgehend von `baum03.py` das Programm `baum04.py` mit folgender `baum`-Funktion:

```
def baum(stamm, faktor, winkel, tiefe):
        # tiefe ist die gewünschte Rekursionstiefe
    if tiefe == 0:
        return
    forward(stamm)
    left(0.5*winkel)
    baum(faktor*stamm, faktor, winkel, tiefe - 1)
    right(winkel)
    baum(faktor*stamm, faktor, winkel, tiefe - 1)
    left(0.5*winkel)
    backward(stamm)
```

➔ Experimentiere auch mit dieser neuen Baumfunktion, um ihre Eigenschaften zu erforschen. Wieder siehst du nebenstehend ein Beispiel:

```
>>> baum(80,0.45,120,3)
>>> baum(55,0.8,45,5)
```



Beispiel 4: Bei Bäumen in der Natur ist ja normalerweise der Stamm dicker als die Äste. Auch werden die Äste immer dünner, je weiter draußen sie sind.

Das können wir erreichen, indem wir die Strichdicke der Turtle umso dünner wählen, je tiefer die Rekursion ist. Als einfachste Lösung bietet sich an, die Strichdicke gleich der Rekursionstiefe zu setzen. Dazu brauchen wir nur in der Funktion `baum` als erste Anweisung `width(tiefe)` einzufügen.

Hier möchte ich dich auf ein kleines Problem hinweisen, das damit verbunden ist. Die Strichdicke der Turtle spielt eine Rolle bei den Aufrufen von `forward` und von

backward. Weil bei rekursiven Aufrufen von baum die Strichdicke entsprechend der Rekursionstiefe verringert wird, hat die Turtle, wenn sie von der Rekursion zurückkehrt und die backward-Anweisung ausführt, eine geringere Strichdicke als bei der forward-Anweisung.

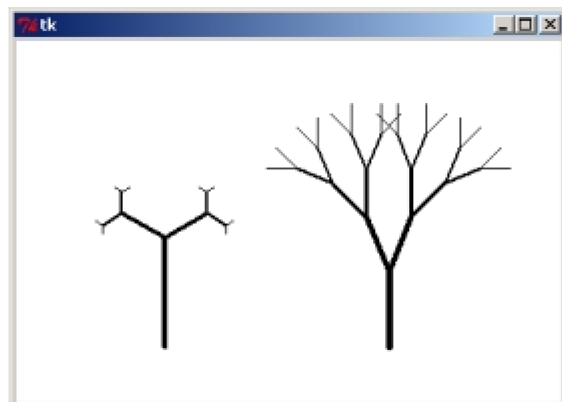
Das fällt in unserem Programm an sich nicht auf – solange wir beispielsweise die Federfarbe der Turtle nicht ändern. Auf der anderen Seite ist es aber gar nicht nötig, den Stamm mit backward ein zweites Mal zu zeichnen. Wir können daher dieses Problem nachhaltig dadurch lösen, dass wir vor der backward-Anweisung die Feder anheben und *danach wieder senken*. Letzteres ist unbedingt nötig, da nach dem Abschluss der Zeichnung eines Baums ja meistens wieder einer gezeichnet werden muss. Dazu muss die Feder unten sein.

➔ Erzeuge baum05.py aus baum04.py mit folgender baum-Funktion:

```
def baum(stamm, faktor, winkel, tiefe):
    if tiefe == 0:
        return
    width(tiefe)
    forward(stamm)
    left(0.5*winkel)
    baum(faktor*stamm, faktor, winkel, tiefe - 1)
    right(winkel)
    baum(faktor*stamm, faktor, winkel, tiefe - 1)
    left(0.5*winkel)
    up()
    backward(stamm)
    down()
```

➔ Experimentiere wieder mit dieser neuen Baumfunktion, um ihre Eigenschaften zu erforschen. Nebenstehend siehst du ein Beispiel, wie unsere Bäume jetzt aussehen können.

```
>>> baum(80,0.45,120,4)
>>> baum(55,0.8,45,5)
```



Beispiel 5: In der Natur sind die Bäume auch nicht so ebenmäßig mit lauter gleich langen Ästen. (Vielleicht mit Ausnahme von den Bäumen, von denen die immer gleichen ebenmäßigen Golden Delicious stammen.)

Wir wollen jetzt jedenfalls die Länge der Äste vom Zufall bestimmen lassen. Auch das ist einfach: `faktor` ist ja stets eine Zahl zwischen 0 und 1. Eine solche zufällige Zahl liefert uns die Funktion `random` aus dem Modul `random`. Das heißt, wir schreiben in jeden Aufruf von Baum `random()*stamm` als Wert für den nächsten Stamm hinein. Den Parameter `faktor` können wir dadurch ganz einsparen:

➔ Erzeuge ausgehend von `baum05.py` das Programm `baum06.py` mit folgender `baum`-Funktion:

```
from random import random

def baum(stamm, winkel, tiefe):
    if tiefe == 0:
        return
    width(tiefe)
    forward(stamm)
    left(0.5*winkel)
    baum(random()*stamm, winkel, tiefe - 1)
    right(winkel)
    baum(random()*stamm, winkel, tiefe - 1)
    left(0.5*winkel)
    up()
    backward(stamm)
    down()
```

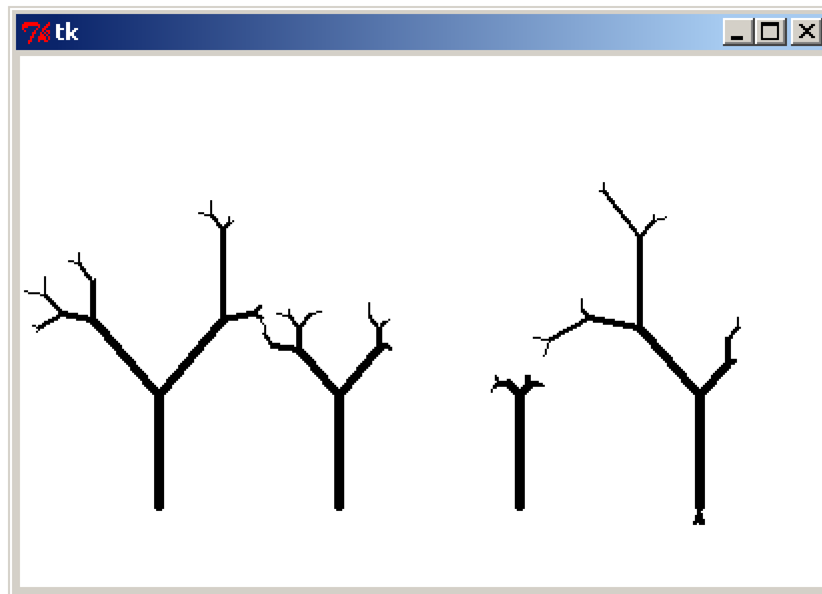
➔ Experimentiere wieder mit dieser neuen Baumfunktion, um ihre Eigenschaften zu erforschen. Hier ist wieder ein Beispiel, wie unsere Bäume jetzt aussehen können. Beachte, dass nun ein und derselbe Aufruf verschiedene Bäume erzeugen kann. Wir verdanken das dem Spiel des Zufalls!

Auf der nächsten Seite findest du eine Grafik mit vier verschiedenen Bäumen, die von dieser Funktion gezeichnet worden sind.

Falls du Lust hast, nun deine Kreativität spielen zu lassen und noch weitere `baum`-Funktionen zu erfinden – es gibt zahllose Möglichkeiten. Hier nur einige Ideen, die du beliebig kombinieren kannst:

- Du kannst zum Beispiel `faktor` zwar zufällig wählen, aber nicht im Bereich 0 bis 1, sondern in einem kleineren Bereich, etwa 0.25 bis 0.75.

- Du kannst den Winkel zwischen den Ästen in einem gewissen Bereich zufällig bestimmen.
- Du kannst die Äste asymmetrisch ansetzen, der linke etwas mehr geneigt als der rechte, oder ...
- Du kannst bei jeder Verzweigung drei oder mehr Äste entspringen lassen ...



Vier Aufrufe von `baum(55,80,5)`

Rekursive Funktionen mit Wert: fak & ggt

Nun wollen wir uns einige Beispiele jenseits der Turtle-Grafik anschauen, wo die Idee der Rekursion genutzt werden kann.

Zunächst beschäftigen wir uns mit Funktionen, die Werte zurückgeben.

Die Faktorielle – Funktion rekursiv

Erinnere dich, wie die Faktorielle-Funktion definiert war:

Ein Beispiel: 6! = 6 * 5 * 4 * 3 * 2 * 1 = 6 * (5 * 4 * 3 * 2 * 1) = 6 * 5!

Allgemein: n! = n* (n-1) * (n-2) * ... * 2 * 1 = n * ((n-1) * (n-2) * ... * 2 * 1)) = n * (n - 1)!

»Weniger vom Gleichen«

Aber natürlich muss auch hier das Zurückgehen auf »Weniger vom Gleichen« einmal ein Ende haben, nämlich bei n = 0. Denn wir wissen: 0! = 1.

Deshalb beschreiben Mathematiker die Faktorielle-Funktion oft durch eine rekursive Definition.

$$n! = \begin{cases} n * (n-1)! & \text{für } n > 0 \\ 1 & \text{für } n = 0 \end{cases}$$

Dies lässt sich mühelos zu einer Funktionsdefinition in Python umbauen. Wir nennen die Funktion `fakt` im Unterschied zur früheren Funktion `fak`:

```
def fakt(n):
    if n == 0:
        return 1
    else:
        f1 = fakt(n-1)   # zunächst (n-1)! berechnen
        f = n * f1       # dann mit n multiplizieren
        return f         # Produkt zurückgeben
```

➔ Schreibe diesen Code in eine Datei `rekfunktionen.py` und teste dann die Funktion:

```
>>> for n in range(11):
      print fakt(n),

1 1 2 6 24 120 720 5040 40320 362880 3628800
```

Der Code für diese Funktion ist leicht zu verstehen. Er ist aber eigentlich unnötig umständlich. Denn die Zuweisung der Zwischenergebnisse `fakt(n-1)` und `n*f1` an die lokalen Namen `f1` und `f` ist deshalb überflüssig, weil auf diese Werte in der Folge nicht mehr zugegriffen wird. Daher kannst du das gesuchte Produkt gleich nach seiner Berechnung direkt zurückgeben:

```
def fakt(n):
    if n == 0:
        return 1
    else:
        return n * fakt(n-1)
```

➔ Ändere den Code von `fakt` dementsprechend ab und teste ihn erneut!

Die Funktion ggt rekursiv

Die Funktion »größter gemeinsamer Teiler« wir im Bonuskapitel 1, **Polys und der Herr Euklid**, behandelt. Grundlage für die Programmierung dieser Funktion war der euklidische Algorithmus. Grundlage für diesen wiederum ist die folgende Erkenntnis:

Der *größte gemeinsame Teiler* von a und b *ist gleich* dem *größten gemeinsamen Teiler* von b und dem Rest, der bei der Division von a durch b auftritt.

Die Berechnung des größten gemeinsamen Teilers wird also hier auch auf die Berechnung von »Weniger vom Gleichen« zurückgeführt. Also hatten wir schon damals eine Rekursionsvorschrift vor uns, ohne es zu wissen. Wie schaut der zugehörige Basisfall aus? Auch den habe ich dort schon beschrieben:

Wenn bei der Division a und b der Rest 0 auftritt, dann ist die kleinere Zahl b bereits der gesuchte größte gemeinsame Teiler ... In diesem Fall führt unsere Rekursionsvorschrift im nächsten Schritt auf die Bestimmung des größten gemeinsamen Teilers von b und 0. Ist also die Rekursion so weit gekommen, dass die zweite Zahl 0 ist, dann ist die erste der gesuchte größte gemeinsame Teiler.

```
Funktion: größter gemeinsamer Teiler von a und b
    wenn b gleich 0 ist:
        Rückgabe a
    sonst:
        Rückgabe des größten gemeinsamen Teilers von b
                und dem Rest bei der Division von a durch b
```

Beachte, dass das zweite Argument immer kleiner wird, weil der Rest immer kleiner ist als der Divisor. Spätestens wenn b gleich 1 geworden ist, wird der Rest 0. Ende!

Auch dieser Entwurf kann äußerst elegant codiert werden. Ich nenne die Funktion diesmal `ggtr(a,b)`. Das `r` steht für rekursiv, wieder zwecks Unterscheidung von der iterativen Fassung aus Bonuskapitel 1.

```
def ggtr(a,b):
    if b == 0:
        return a
    else:
        return ggtr(b, a % b)
```

Würdest du mit mir übereinstimmen, dass wir hier ein alternatives »Python-Juwel« vor uns haben?

➔ Schreibe diesen Code auch in die Datei `rekfunktionen.py`.

➔ Teste ihn im IPI. Hier einige Testrechnungen:

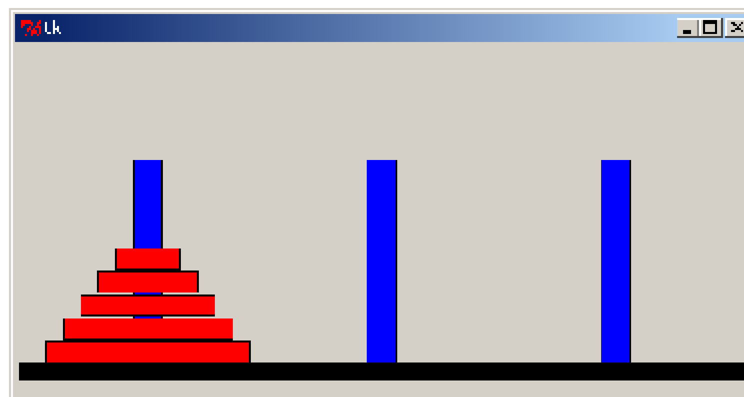
```
>>> ggtr(1001,101)
1
>>> ggtr(1001,10001)
1
>>> ggtr(1001,100001)
11
>>> ggtr(1001,101101)
1001
```

➔ Führe weitere Testrechnungen durch und überprüfe jeweils, dass `ggtr` dieselben Ergebnisse liefert wie `ggt` aus Kapitel 9.

Die Türme von Hanoi

Zum Abschluss dieses Kapitels möchte ich mit dir die Lösung für ein kleines Denkspiel programmieren. Es ist bekannt unter dem Namen »Die Türme von Hanoi«. Hier zunächst die Aufgabenstellung:

Gegeben sind ein Turm von Scheiben mit abnehmender Größe, die auf einem Pflock A aufgestapelt sind und zwei weitere Pflöcke: der Zielpflock B und der Hilfspflock C.

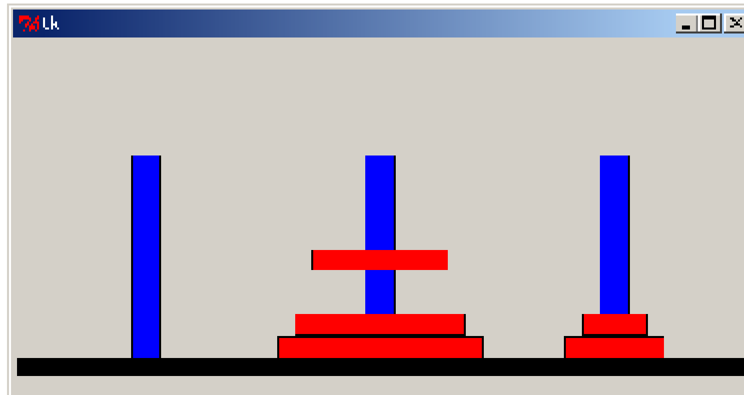


Ausgangssituation für das Spiel »Türme von Hanoi«

Die Aufgabe ist, die Scheiben von Pflock A nach dem Zielpflock B zu verschieben. Dabei müssen jedoch folgende zwei Regeln eingehalten werden.

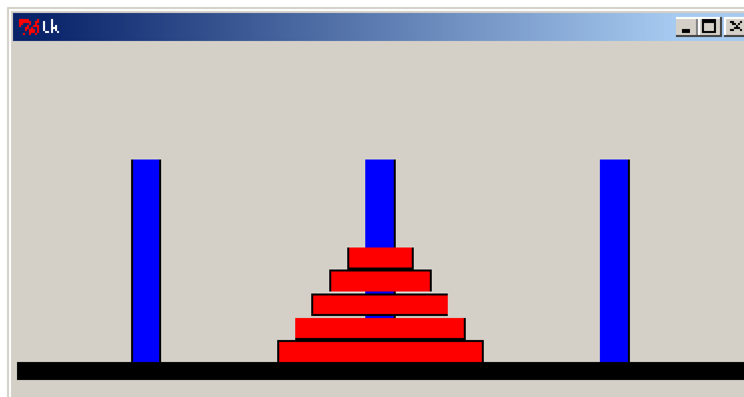
1. Bei jedem Zug darf nur eine Scheibe bewegt und auf einem der drei Pflöcke abgelegt werden.
2. Es darf niemals eine größere Scheibe auf eine kleinere gelegt werden.

Eine nach diesen Regeln erlaubte Zwischensituation ist zum Beispiel diese:



Erlaubte Spielsituation im Spiel »Türme von Hanoi«: Eine Scheibe wird gerade auf Stock B abgelegt. Auf Stock C dürfte sie nicht abgelegt werden, weil sie zu groß ist.

Das Ziel des Spiels ist, schließlich die folgende Situation zu erreichen:



Zielsituation im Spiel »Türme von Hanoi«

Unser Ziel hingegen ist, eine Funktion `hanoi` zu schreiben, der wir die Höhe des Ausgangsturmes als Argument übergeben und die uns eine Zugfolge ermittelt und ausgibt, die die Aufgabe löst. Als weitere Argumente übergeben wir noch die Bezeichnungen von Ausgangsstock, Zielstock und Hilfsstock.

Die Ausgabe soll in ganz einfacher Gestalt erfolgen: Wenn eine Scheibe von Stock A nach Stock C zu verschieben ist, soll das Programm dies so darstellen:

```
A -> C
```

Ein Programmlauf für einen Turm von drei Scheiben sollte so aussehen:

```
>>> hanoi(3,"A","B","C")
A -> B
A -> C
B -> C
```

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

`hanoi(3,"A","B","C")` bedeutet also »Verschiebe einen Turm von 3 Scheiben von Stock A nach Stock B, wobei Stock C als Hilfsstock dient« .

Schalte jetzt deinen Computer ab!

Wenn du ein Problem programmieren willst, ist die wichtigste Voraussetzung, dass du das Problem selbst genau verstanden hast. Für die Türme von Hanoi heißt das, du musst jetzt erst einmal erforschen, wie dieses Spiel mit Kopf und Hand zu lösen ist.

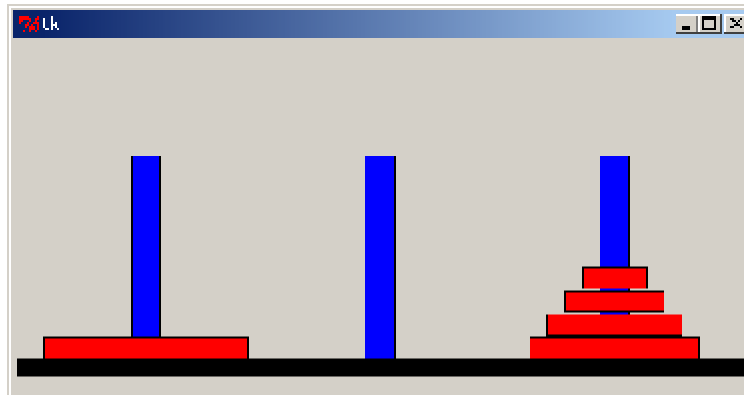
Ich rate dir daher Folgendes:

- ➔ Schalte deinen Computer ab und nimm dir Zeit, Lösungswege für das Spiel Türme von Hanoi zu erforschen.
- ➔ Suche dir Material für einen 5er-Turm. Als »Scheiben« kannst du verschieden große Münzen verwenden, zum Beispiel 2 €, 50 Cent, 1 €, 20 Cent, 10 Cent. oder, wenn dir die zu klein sind, nimm fünf verschieden große Bücher, die du der Größe nach aufeinander stapelst.
- ➔ Markiere auf deinem Arbeitstisch drei Positionen mit A, B, C und lege zunächst einen nur drei »Scheiben« hohen Stapel zur Markierung A.
- ➔ Führe die sieben oben abgedruckten Spielzüge aus. Als Ergebnis muss dann der 3er-Turm bei der Markierung B liegen.
- ➔ Versuche nun Lösungen für einen 4er-Turm und danach für einen 5er-Turm zu finden. Kannst du die Lösungswege beschreiben? (Alte Programmiererinnenregel: Wenn du ein Problem verbal beschreiben kannst, kannst du die Lösung auch programmieren!) Welche Zwischenstadien müssen jedenfalls erreicht werden? Was ist den Lösungen für den 3er-, 4er- und 5er-Turm gemeinsam?
- ➔ Hast du Ideen, wie du das programmieren könntest?
- ➔ Wenn dir dazu nichts einfällt, zähle ab, wie viele Züge du brauchst, um den 4er-Turm zu verschieben. Und wie viele Züge für den 5er-Turm.

Fast jeder, der sich zum ersten Mal mit diesem Spiel beschäftigt, findet, dass das ziemlich schwierig aussieht. Wahrscheinlich geht es dir auch so. Mach' dir nichts draus, das ist normal!

Der Weg zu einer Lösung

Ziemlich klar dürfte sein, dass im Laufe der Lösung einmal die größte Scheibe von Stock A nach Stock B bewegt werden muss. Das geht aber nur, wenn auf ihr nichts drauf liegt und Stock B leer ist. (Die größte Scheibe darf nicht auf eine kleinere gelegt werden.) Daher müssen in dieser Situation alle übrigen Scheiben auf Stock C liegen:

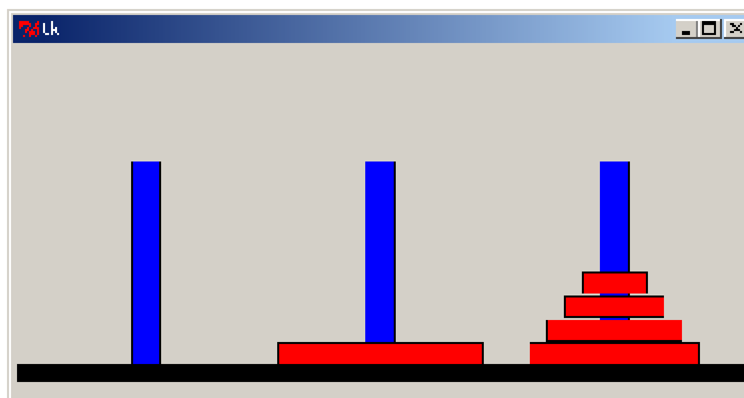


Jetzt kann die größte Scheibe nach B verschoben werden.

Um diese Lage zu erreichen, muss der Turm aus den oberen vier Scheiben von A nach C verlagert werden. Wobei durchaus Stock B als Hilfsstock benutzt werden kann. Hast du nicht schon einmal einen 4er-Turm verschoben? Von A nach B zwar, aber das kann ja keinen großen Unterschied ausmachen.

Und vor allem: Ist das nicht »Weniger vom Gleichen«?

Und nun kann die unterste Scheibe von A nach B verschoben werden:



Die größte Scheibe ist bereits am Ziel.

Zuletzt musst du wieder den 4er-Turm verschieben, diesmal von C nach B! Dabei kannst du den Stock A als Hilfsstock benutzen. Also nochmals »Weniger vom Gleichen«! Damit ist das anfangs dargestellte Ziel erreicht.

Formulieren wir einmal diese Gedanken:

```
funktion versetze:
Parameterliste: anzahl (der Scheiben)
                start (Bezeichnung des Ausgangsstocks)
                ziel (Bezeichnung des Zielstocks)
                hilf (Bezeichnung des Hilfsstocks)
    wenn anzahl gleich 1 ist:
        führe Zug von start nach ziel aus
    sonst:
        versetze (anzahl – 1)
                vom Ausgangsstock start
                zum Hilfsstock hilf unter
                Benutzung von ziel als Hilfsstock
        versetze 1 von start nach ziel
        versetze (anzahl – 1)
                vom Hilfsstock hilf
                zum Zielstock ziel unter
                 Benutzung von start als Hilfsstock
```

Erläuterung:

Der Aufruf

```
versetze(5,"A","B","C") # 5 Scheiben A->B, C ist Hilfsstock
```

würde dann dazu führen, dass

```
versetze(4,"A","C","B") # 4 A->C, mit B als Hilfsstock
versetze(1,"A","B","C") # 1 A->B
versetze(4,"C","B","A") # 4 C->B, mit A als Hilfsstock
```

auszuführen ist.

Es wird dich vielleicht überraschen, aber damit haben wir den Code von `hanoi` auch schon fertig:

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

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

def zug(von,nach):
    print von, "->", nach
```

Dabei wird die Arbeit nur von `versetze` geleistet. Die anderen beiden Funktionen sind Hilfsfunktionen. Die einzige Anweisung von `zug`, eine `print`-Anweisung, hätten wir ebenso gut in den Basisfall von `versetze` direkt hineinschreiben können. Solltest du dich jedoch irgendwann einmal entscheiden, die Darstellung der Spielzüge zu ändern (weil dir beispielsweise A->B zu spartanisch ist), dann kannst du mit unserer Lösung die Funktion `versetze` unverändert lassen und brauchst nur `zug` abzuändern. Denn wir haben durch die Einführung der Funktion `zug` die *Berechnung* der Lösung von der *Darstellung* der Lösung getrennt!

➔ Schalte deinen Computer wieder ein und mplementiere `hanoi` in einem Modul `hanoi.py`.

➔ Lasse `hanoi` die Lösung für fünf, eventuell auch sechs Scheiben berechnen. Spiele praktisch mit deinen realen »Scheiben« die Lösung nach und überprüfe, dass sie wirklich zum Ziel führt.

Am Ende des Kapitels über die Rekursion angelangt, hast du, hoffe ich, die Einsicht gewonnen, dass es Probleme gibt, die mit Rekursion auf »natürliche« Weise elegant gelöst werden können, während iterative Lösungen (vermutlich) technisch viel verwickelter ausfallen müssten. Es sind dies Probleme, die eine »rekursive Struktur« haben. Damit meine ich, dass sie auf Teilprobleme führen, die »Weniger vom Gleichen« sind. Die Lösung besteht dann darin, ein paar kleine Schritte mit der Lösung von »Weniger vom Gleichen« zu verbinden, bis man auf einen einfachen »Basisfall« stößt, dessen Lösung sehr einfach ist.

Zusammenfassung

- Rekursion bezeichnet ein Verfahren zur Lösung von Problemen, bei dem das gegebene Problem auf ein kleineres gleichartiges zurückgeführt wird.
- Bei rekursiven Lösungen muss immer zwischen einem Basisfall und dem rekursiven Fall unterschieden werden.
- Wird der Basisfall vergessen, entsteht eine endlose Rekursion. Das Programm wird mit einer Fehlermeldung beendet.

Einige Aufgaben ...

Aufgabe 1: Gegeben ist folgende Funktionsdefinition:

```
def herunterzaehlen(n):
    print n,
    herunterzaehlen(n-1)
```

Welche Wirkung hat nun der folgende Aufruf dieser Funktion?

```
>>> herunterzaehlen(10)
```

Aufgabe 2: Schreibe eine iterative Fassung der `spirale`-Funktion. Sie soll Spiralen mit einer Schleife und der Funktion `polyschritt` erzeugen.

Aufgabe 3: Schreibe eine nicht rekursive Funktion, die die gleiche Treppe wie oben im Text abgebildet zeichnet.

Aufgabe 4: Ändere `baum03.py` so ab, dass an den Enden der Zweige des Baumes kleine rote Äpfel gezeichnet werden (`apfelbaum.py`).

Aufgabe 5: Die Quersumme von 176589224 ist 44. Sie ergibt sich als Summe von 4 und der Quersumme von 17658922. 4 berechnet sich als `176589224 % 10`. 17658922 berechnet sich als `176589224 // 10`. Gesucht ist eine Funktion, die die Quersumme einer ganzen Zahl zurückgibt. Erstelle dafür eine rekursive und eine iterative Version.

... und zu diesem Kapitel nur eine wichtige Frage

- Warum steht in der Überschrift dieses Kapitels: »Weniger vom Gleichen«?

Hinter den Kulissen

Ich lade dich hier ein, die Arbeitsweise von `treppe` mit einer vereinfachten Version im Debugger zu zeigen. Bei dieser Version ist die Treppenbreite fix auf 20 eingestellt. Du brauchst nun auf die `breite` nicht weiter zu achten und kannst dich besser auf das Wesentliche konzentrieren. Der Debugger wird hier (wie schon oben

beim Salamischneiden) nicht zur Fehlersuche, sondern zur Verfolgung des Ablaufs des Programms benutzt.

Der Code dieser vereinfachten Version sieht so aus:

```
def treppe(stufen):
    if stufen == 0:    # Basisfall
        forward(20)
    else:              # rekursiver Fall
        stufe_hinauf()
        treppe(stufen - 1)
        stufe_hinunter()

def stufe_hinauf():
    forward(20)
    left(90)
    forward(20)
    right(90)

def stufe_hinunter():
    right(90)
    forward(20)
    left(90)
    forward(20)
```

➔ Schreibe diesen Code in eine Datei `treppe_demo.py`. Kopiere an den Anfang der Datei den Code von `start` aus `rekursion.py`. Speichere die Datei und führe sie aus.

➔ Halte das *PYTHON SHELL*-Fenster und das Editor-Fenster offen und gib dem IPI ein:

```
>>> start()
```

Dadurch wird das Turtle-Grafik-Fenster geöffnet und die Turtle in die linke untere Ecke gestellt.

➔ Öffne das Debugger-Fenster über den Menüpunkt DEBUG|DEBUGGER.

Wie funktioniert treppe?

Im Folgenden musst du mit diesen vier Fenstern arbeiten: Python Shell, Editor, Debugger und Turtle-Grafik.

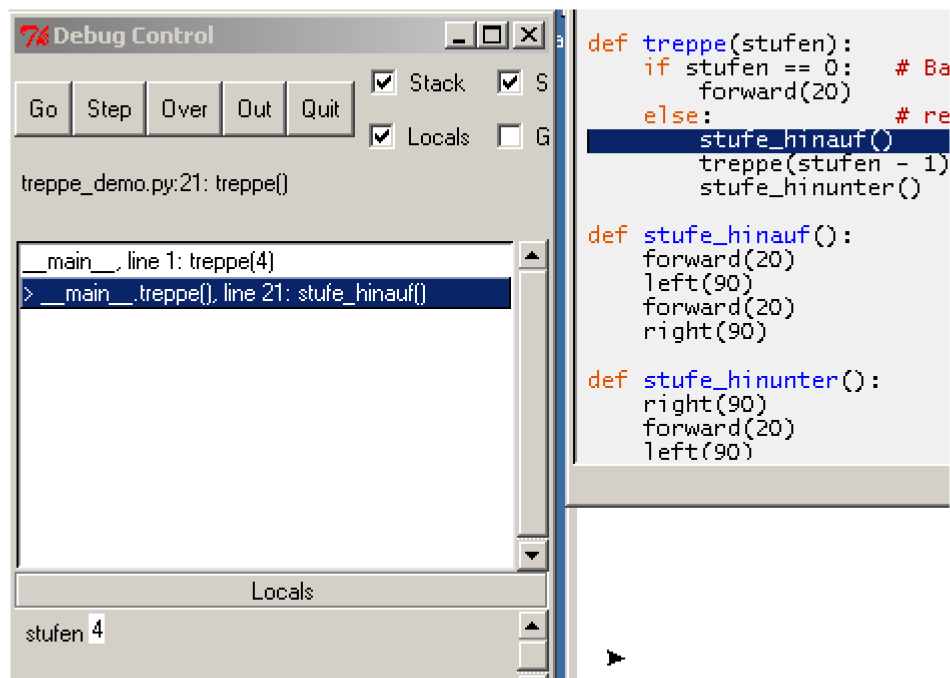
➔ Stelle auf deinem Desktop eine übersichtliche Fensteranordnung her:

Verfolge den Ablauf des Aufrufs `treppe(4)` wie folgt:

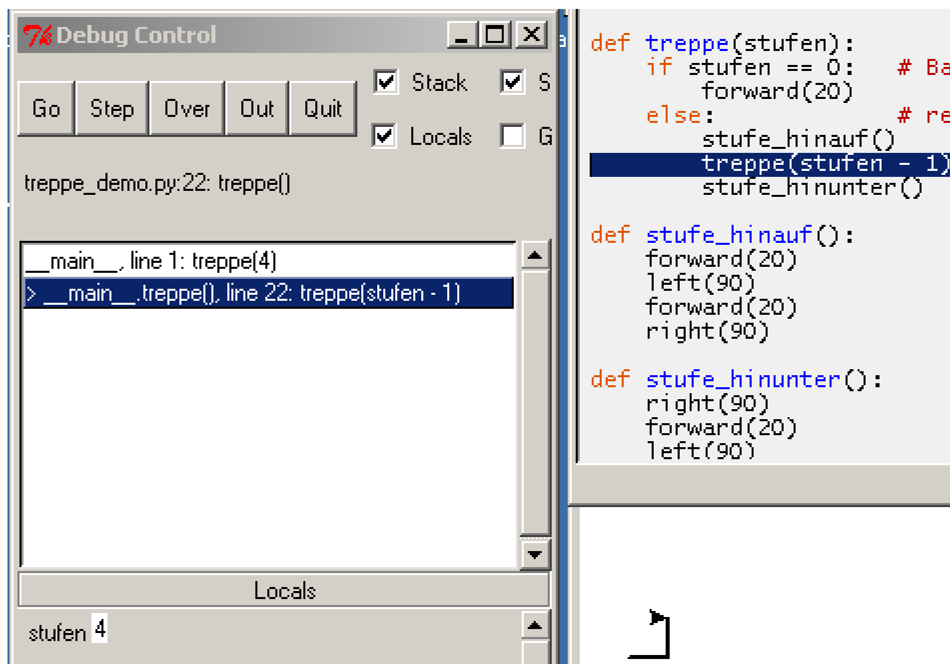
➔ Gib dem IPI die Anweisung `treppe(4)` ein.

➔ Klicke im Debug-Fenster 2 Mal auf STEP. Die Ausführung von `treppe(4)` beginnt. Im Debugger-Fenster wird angezeigt, dass der lokale Name `stufen` auf den Wert 4 verweist.

➔ Klicke 2 Mal auf STEP. Nun steht die Ausführung von `stufe_hinauf` bevor.

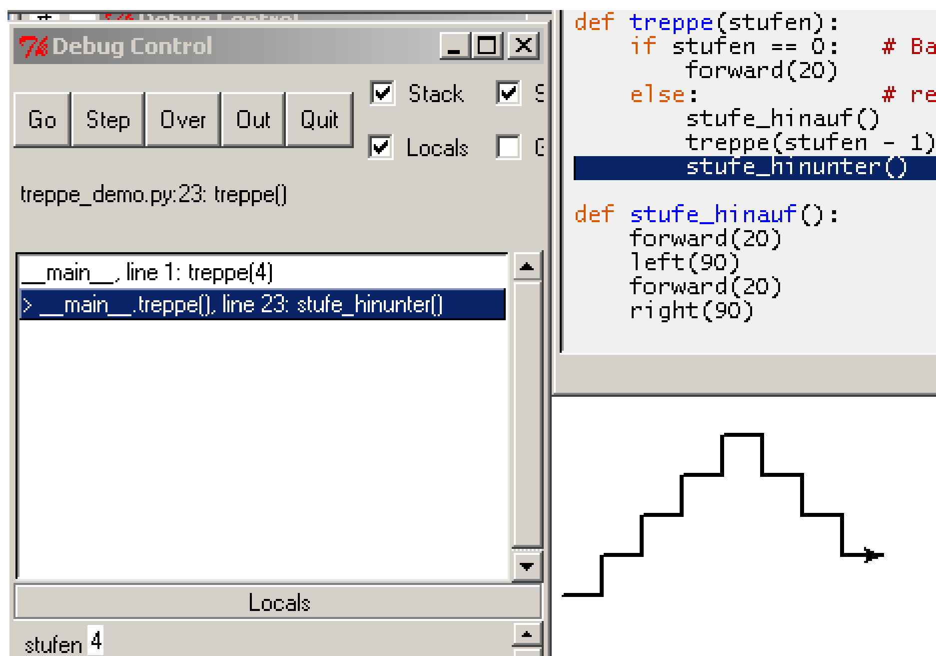


➔ Wie `stufe_hinauf` ausgeführt wird, ist nicht sonderlich interessant. Klicke daher auf OVER.



Nun hat die Turtle die erste Stufe gezeichnet. Der lokale Name `stufen` hat immer noch den Wert 4 und die Ausführung von `treppe(stufen-1)` steht an. Das heißt, im nächsten Aufruf von `treppe` wird 3 als Argument eingesetzt werden. Vertraue darauf, dass `treppe(3)` funktioniert, und führe es gleich ganz aus:

➔ Klicke auf OVER.

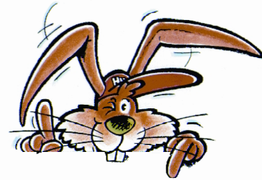


Die Anweisung `treppe(stufen-1)` mit dem Wert 4 von `stufen` ist ausgeführt worden. Jetzt ist noch `stufe_hinunter` auszuführen:

➔ Klicke auf OVER.

Die Treppe wird fertig gezeichnet und die Ausführung von `treppe(4)` beendet. (Der letzte ausgeführte Aufruf bleibt im Debugger-Fenster stehen.)

Was geschieht aber *genau* während der Ausführung von `treppe(3)`?



Egal, ob du nun das Gefühl hast, die Sache schon ganz zu verstehen, oder noch einiges unklar ist: Ich rate dir, die Ablaufverfolgung von `treppe(4)` nochmals vorzunehmen und diesmal jeden Schritt treppauf und treppab zu betrachten. Lenke in jeder Situation dein Augenmerk darauf, wie viele Aufrufe von Treppe noch aktiv beziehungsweise schon abgeschlossen sind.