

## Teil IX

# 2D-Grafiken mit Turtle und TKInter

# Turtle-Grafiken

- Anschaulicher Beschreibungssprache zum Zeichnen einfacher Grafiken
- Eine Schildkröte läuft über die Bildfläche
- Sie läuft nur gerade Strecken
- Dabei kann sie die Strecke zeichnen (oder auch nicht)
- Am Ende angelangt, kann sie sich drehen
- Sehr einfach zu verwenden
- Insbesondere für Fraktale gut geeignet

## Einfacher Formalismus

- Bewegung um Länge  $x$  mit Zeichnung
- Bewegung um Länge  $x$  ohne Zeichnung
- Drehung um Winkel  $\alpha$  im Uhrzeigersinn
- Drehung um Winkel  $\alpha$  gegen den Uhrzeigersinn

## Turtle-Grafiken: erste Befehle

- `reset()`: Zeichenfläche erzeugen/zurücksetzen
- `forward(laenge)`: `laenge` Pixel in Blickrichtung laufen
- `left(winkel)`: Um `winkel` Grad nach links drehen
- `right(winkel)`: Um `winkel` Grad nach rechts drehen
- `goto(x,y)`: Zu den Koordinaten `x,y` gehen
- `penup()`: Stift hochnehmen (nicht mehr zeichnen)
- `pendown()`: Stift absenken

```
import turtle as t
t.reset()
for i in range(10):
    t.forward(50)
    t.left(36)
```

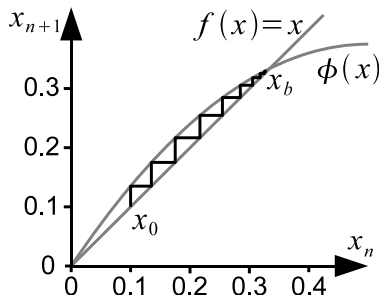
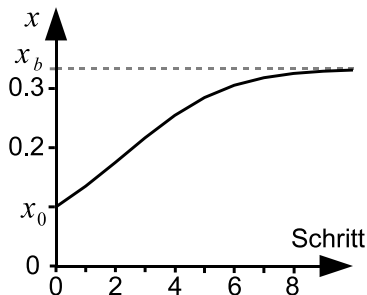
## Turtle-Grafiken: weitere Befehle

- `undo()` letzten Schritt rückgängig machen
- `showturtle()`, `hideturtle()` Schildkröte anzeigen/verstecken
- `pencolor(color)` Farbe auswählen
- `circle(radius, extend, steps)` Kreis malen
- `dot(size, color)` Punkt malen
- `speed(n)` Malgeschwindigkeit 1-10 wählen
- `setworldcoordinates(llx, lly, urx, ury)` neue Bezugspunkte
- `exitonclick()` Bild einfrieren bis zum Schließen durch Klick
- ...

# Chaos: logistische Abbildung

## Iterationsfunktionen / diskrete Abbildungen

- $x_{n+1} = \Phi(x_n)$ : Wert zur Zeit  $n + 1$  ist Funktion des Wertes zur Zeit  $n$
- Verwendung z.B. in der Populationsdynamik
- Logistische Abbildung:  $\Phi(x) = rx(1 - x)$
- z.B.  $r = 1.5, x_0 = 0.1$



## Zeichnen der Abbildung mit Turtle-Grafik

- Datei erstellen (logabb.py)
- Turtle einbinden

```
import turtle
```

- Die logistische Abbildung:

```
def log_abb_1(r,x):  
    return r*x*(1-x)
```

- Initialisierung

```
def init(n):  
    turtle.reset()  
    turtle.setworldcoordinates(-1.0,-0.1, n+1, 1.1)  
    turtle.speed(0)  
    turtle.hideturtle()
```

- Funktion zum Plotten:

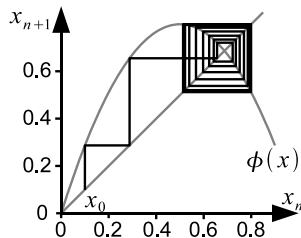
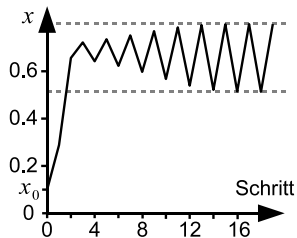
```
def plot(abb, x_0, r, numsteps, farbe):  
    x = x_0  
    turtle.pencolor(farbe)  
    turtle.penup()  
    turtle.goto(0, x)  
    turtle.pendown()  
    for i in range(numsteps):  
        x=abb(r,x)  
        turtle.goto(i+1,x)
```

- Plotten für  $r = 1.5, x_0 = 0.1$

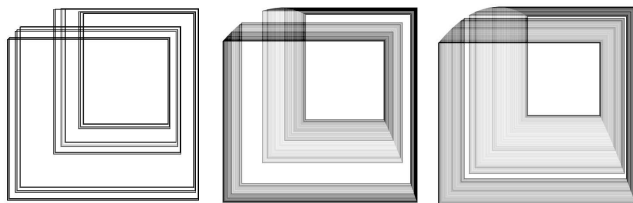
```
from logabb import *  
n=100  
init(n)  
plot(log_abb_1, 0.1, 1.5, n, "blue")
```

## Übergang ins Chaos

- z.B.  $r = 3.2, x_0 = 0.1$



- $r = 3.57, r = 3.58, r = 3.59$





## Auswirkung von Rundungsfehlern

- Bisherige Abbildung

```
def log_abb_1(r,x):  
    return r*x*(1-x)
```

- Nun umgeformte Variante (äquivalent!)

```
def log_abb_2(r,x):  
    return r*(x-x**2)
```

- Beide Varianten für  $r = 4.0$

```
from logabb import *  
n=100  
init(n)  
plot(log_abb_1, 0.1, 4.0, n, "blue")  
plot(log_abb_2, 0.1, 4.0, n, "red")
```

# Rekursion

- Bei einer Rekursion wird eine Funktion durch sich selbst definiert
- Beim Aufruf der Funktion wird die Funktion selbst wieder aufgerufen
- Abbruchkriterium verhindert endlose Rekursion

## Fibonacci-Folge (1202)

- Fibonacci beschrieb damit das Wachstum von Kaninchenpopulationen
- $f_n = f_{n-1} + f_{n-2}$
- $f_0 = 0$
- $f_1 = 1$

```
def fib(n):  
    if n<=1:  
        return n  
    else:  
        return fib(n-1)+fib(n-2)
```

## Koch-Kurve (1904)

- Eines der ersten entdeckten Fraktalen Objekte
- Überall stetig, nirgends differenzierbar
- Beginn mit einer Strecke
- Mittleres Drittel wird entfernt
- Stattdessen zwei gleichlange Strecken, die ein Dreieck bilden



- Kann ebenfalls mit Rekursion erzeugt werden
- Wir benötigen eine Funktion, die die Kurve malt
- Um die Kurve zu malen, müssen wir erst alle vier Teilstrecken malen
- Für jede Teilstrecke, müssen deren vier Teilstrecken gemalt werden
- ...

## Rekursive Koch-Funktion

```
import turtle

def kochkurve(laenge, ebene):
    if(ebene>0):
        kochkurve(laenge/3.0, ebene-1)
        turtle.left(60)
        kochkurve(laenge/3.0, ebene-1)
        turtle.right(120)
        kochkurve(laenge/3.0, ebene-1)
        turtle.left(60)
        kochkurve(laenge/3.0, ebene-1)
    else:
        turtle.forward(laenge)
```

## Initialisierung

```
def init():  
    turtle.reset()  
    turtle.setworldcoordinates(-0.1,-0.9, 1.1, 0.3)  
    turtle.speed(0)  
    turtle.hideturtle()
```

## Ausführung

```
if __name__ == '__main__':  
    n=int(raw_input("Anzahl Rekursionen:"))  
    init()  
    kochkurve(1.0, n)  
    turtle.right(120)  
    kochkurve(1.0, n)  
    turtle.right(120)  
    kochkurve(1.0, n)  
    turtle.exitonclick()
```

# Koordinatenbasierte GUIs

## GUIs für Python

- GUI: Graphical User Interface
- Inzwischen verschiedene GUIs verfügbar
  - TKinter: Python Schnittstelle für Tcl/TK (im Standard enthalten)
  - EasyGUI: baut auf TKinter auf, einfach aber beschränkt
  - PyGTK: basiert auf GTK+ (Linux)
  - wxPython: benutzt OS-Grafikbibliotheken
  - PythonCard: baut auf wxPython auf, einfach aber beschränkt
  - PyQt: basiert auf Qt (z.B. KDE)
  - PythonWin: nur für Windows
  - ...

# Tcl/TK

- Tool Command Language / ToolKit
- Relativ einfach
- Ausreichend für mittelgroße Anwendungen

## widgets

- Gängiges Wort für jegliche GUI-Komponente
- `frame` Kontainer für alles mögliche, z.B. ein Fenster
- `canvas` Leinwand/Zeichenfläche
- `button`
- `checkbox`
- `radiobutton`
- `label` z.B. Text
- `menu`
- `scrollbar`
- ...

# Module Tkinter

## Hello world

```
import Tkinter as tk
root = tk.Tk()
root.title("Hello-Fenster")
label = tk.Label(root, text="Hallo_Welt",
                 font="times_32_bold")
label.pack()
button = tk.Button(root)
button["text"] = "Hallo-Knopf"
button["background"] = "blue"
button.pack()
root.mainloop()
```

- `tk.Tk()` erzeugt das Hauptfenster
- `root.title` legt den Titel fest
- `widget.pack` macht das widget sichtbar
- `root.mainloop()` wartet auf Ereignisse



## Prinzipielle Vorgehensweise

- Festlegen, wie die GUI aussehen soll
- zugrundeliegende Funktionalität implementieren
- GUI-Elemente mit Funktionalität verbinden

## Hello world als Klasse

```
import Tkinter as tk
class MyVis:
    def __init__(self, root):
        self.label = tk.Label(root, text="Hallo_Welt",
                               font="times_32_bold")

        self.label.pack()
        self.button1 = tk.Button(root)
        self.button1["text"] = "Hallo-Knopf"
        self.button1["background"] = "blue"
        self.button1.pack()

root = tk.Tk()
myvis = MyVis(root)
root.mainloop()
```

## Änderung der Anordnung

```
import Tkinter as tk
class MyVis2:
    def __init__(self, root):
        self.button1 = tk.Button(root,
            text="Hallo-Knopf", background="blue")
        self.button1.pack(side = tk.LEFT)
        self.button2 = tk.Button(root)
        self.button2.configure(text="Exit",
            background="red")
        self.button2.pack(side = tk.LEFT)

root = tk.Tk()
myvis = MyVis2(root)
root.mainloop()
```

- Button-Optionen können bei der Konstruktion angegeben werden
- Button-Optionen können mit configure geändert werden
- side-Option für pack bestimmt Ausrichtung

## Aktionen aufrufen - Binding

- Bestimmte Ereignisse erzeugen ein „event“
- Z.B. Tastendruck, Mausklick, Mausbewegung, ...
- Ereignisse können an widgets (Buttons,...) gebunden werden
- Der Knopf wartet dann auf das jeweilige Ereignis
- Tritt es auf, wird die assoziierte Funktion aufgerufen

```
class MyVis2:
    def __init__(self, root):
        ...
        self.root = root
        self.button1.bind("<Button-1>", self.action1)
        self.button2.bind("<Button-1>", self.action2)
    def action1(self, event):
        print "Hallo_Welt!"
        self.button1["background"] = "red"
    def action2(self, event):
        self.root.destroy()
```

## Aktionen aufrufen - command

- Bisher: Knopf reagiert nicht auf kompletten Klick
- Event "<Button-1>" ist gleich "<ButtonPress-1>"
- Loslassen entspricht "<ButtonRelease-1>"
- Mit `command` werden mehrere Events an ein widget gebunden

```
class MyVis3:
    def __init__(self, root):
        self.button1 = tk.Button(root,
                                  text="Hallo-Knopf", background="blue",
                                  command=self.action1)
        self.button2 = tk.Button(root,
                                  command=self.action2)
        ...
    def action1(self):
        ...
    def action2(self):
        ...
```

## Parameter an Aktionen übergeben

- Bisher z.B.: `command=self.action1`
- `command` ist die Funktion (deren Name)
- Mit Parameter: `command=self.action1(x)?`
- Hier ist `command` der Rückgabewert der Funktion (`None`)
- Lösung: Funktion ohne Parameter angeben, die eine Funktion mit Parametern aufruft ( $\Rightarrow$  lambda-Funktion)

```
command=lambda: self.action1("Hallo_Welt")
```

- Wert der Variablen zum Zeitpunkt des Events wird verwendet!

```
command=lambda: self.action1(var1, var2)
```

- Variablenwerte zum Zeitpunkt der Erstellung:

```
command=lambda x=var1, y=var2: self.action1(x, y)
```

- Funktionsheader: `def action1(self, x, y):`

# widget Hierarchie

```
class MyApp:
    def __init__(self, root):
        # buttons frame mit einem Knopf
        self.buttons_frame = tk.Frame(root)
        self.buttons_frame.pack(side=tk.TOP)
        self.button1 = tk.Button(self.buttons_frame)
        self.button1.pack(side=tk.LEFT)
        # top frame
        self.top = tk.Frame(root)
        self.top.pack(side=tk.TOP)
        # linker + rechter Frame (Kinder von top)
        self.left_frame = tk.Frame(self.top, background="blue",
            borderwidth=5, height=200, width=80)
        self.left_frame.pack(side=tk.LEFT)   ###
        self.right_frame = tk.Frame(self.top, background="red",
            borderwidth=5, relief=tk.RIDGE, width = 220)
        self.right_frame.pack(side=tk.RIGHT, fill=tk.BOTH)
```

## widget-Hierarchie

- Bisher wurden alle widgets direkt als Kinder von root erzeugt
- widgets können selbst Kinder haben
- Beliebige Verschachtelungen möglich
- Für alle widgets kann ein Padding angegeben werden

## Geometrie-Manager: pack

- Anordnung im wesentlichen über `side=...` (LEFT, TOP, ...)
- `fill` gibt an, wie sich die Objekte ausdehnen sollen
  - `tk.NONE` gar nicht (minimale Ausdehnung)
  - `tk.X`, `tk.Y` nur in X/Y-Richtung
  - `tk.BOTH` in beide Richtungen
- `expand` Das widget versucht, möglichst viel Fläche zu belegen
- `anchor` Damit kann das widget sich in einem bestimmten Teil der Fläche platzieren (`tk.N`, `tk.NW`, `tk.NE`, ..., `tk.CENTER`)

## Geometrie-Manager: grid

- Anordnung der widgets in einem Gitter
- Z.B. `widget.grid(row=2, column=7)`

# Zeichnen mit Tkinter

## Das canvas Widget

- Kann wie jedes widget in ein Frame gesetzt werden
- Kann auch ohne Frame erzeugt werden:

```
cv = tk.Canvas(width=800, height=600)
cv.pack()
```

- Kann zum Zeichnen verwendet werden:

```
cv.create_line(50,50,650, 300)
cv.create_polygon(250, 50, 500, 50, 500)
cv.create_text(400, 100, text="Hallo_Welt")
```

- Weitere Methoden:
  - create\_oval
  - create\_bitmap
  - ...



# Animation mit Tkinter

```
#!/usr/bin/python
from Tkinter import *
import time, math

cv = Canvas(width=800, height=600)
cv.pack()
oval = cv.create_oval(300, 50, 400, 150)
cv.itemconfig(oval, fill='blue')
i = 0.0
while True:
    cv.move(oval, 10*math.cos(i), 10*math.sin(i))
    cv.update()
    time.sleep(0.02)
    i += 0.05
```