

Tkinter Kurzreferenz

Geschichte:

Tcl ('tickle', tool command language) wurde ursprünglich ab 1988 von John Ousterhout an der Berkeley-University in Kalifornien als quelloffene Makrosprache entwickelt, um komfortabel kleine Programme im Betriebssystem schreiben zu können – etwa für die shell in Unix. Wegen ihres großen Erfolges entstand zusätzlich ein plattformübergreifendes 'toolkit' namens **Tk**, um schnell und einfach GUIs in Fensterbetriebssystemen (X-Window,...) schreiben zu können. Diese Kombination nennt man **Tcl/Tk** ('tickle toolkit'), sie wurde auf viele Plattformen übertragen (Windows, Linux, Apple-OS,...) und steht dort gratis zur Verfügung.

Python selbst besitzt keine Grafikfähigkeiten. Das Modul **Tkinter** ist nur ein Interface, das aus Python heraus Kommandos an Tcl/Tk sendet und Antworten empfängt. Da das Toolkit in allen wichtigen Betriebssystemen verfügbar ist, enthält jede Python-Vollinstallation auch das Tcl/Tk und erlangt damit Grafikfähigkeit. Tkinter ist die Standard-GUI für Python, die IDLE selbst ist Tk-Programm.

Alternativen:

Für Windows, Linux und Apple gibt es noch drei 'große' Grafikbibliotheken:

- Qt ('cute') war früher kostenpflichtig von Trolltech, heute gratis. Ein großes komplettes Paket, die KDE von Linux (z.B. in KUBUNTU) oder der Opera-Browser sind Qt-Programme
- GTK+ (Gimp Toolkit) entstand ursprünglich aus den Bibliotheken des Grafikprogramms Gimp. Es findet Anwendung als Grafikoberfläche in GNOME, die Du von UBUNTU kennst.
- WxWindows entstand aus den Bibliotheken des Unix-Toolkits Xt

Es gibt noch zahlreiche weitere Grafikanbindungen für Python, die aber meist weniger für GUIs, sondern als spezialisierte Grafikpakete für besondere Aufgaben gedacht sind (PyGame für Spiele, pyglets für Unix screenlets, etc).

Prinzip:

In Tkinter wird nicht ein vorbereitetes GUI-Layout eingebunden (wie bei Qt, GTK,...) sondern wir beschreiben den Aufbau des Fensters durch Einfügen einzelner Elemente mithilfe von Python-Anweisungen. Diese Elemente werden als **widgets** (window gadgets) bezeichnet.

Vorgehensweise:

- 1.) die Bibliothek Tkinter (in Python 3 heißt sie tkinter) einbinden
- 2.) Das Hauptfenster erzeugen, indem Du die Klasse Tk() aufrufst. Dieses Fenster wird von Windows verwaltet, es wird gerne als 'root Window' bezeichnet, da alle weiteren widgets von ihm abstammen.
- 3.) Ein widget nach dem anderen erzeugen. Seine Eigenschaften werden als Optionen in der Klammer nach dem parent-window angegeben (dictionary) oder später mittels configure() eingestellt bzw. geändert. Jedes widget hat voreingestellte Defaultwerte. Du musst nur die nicht passenden überschreiben.
- 4.) Das widget wird mit einem packer in das parent-Fenster gesetzt. Möglichkeiten sind pack, grid und place. Jedes child-Element hat ein parent-Element.
- 5.) Ein Programm, das mithilfe einer GUI bedient wird, läuft nicht mehr streng nach Programmtext ab, sondern wird durch Benutzereingaben gesteuert. Wir haben keinen linearen Programmfluss, sondern wechseln zwischen unterschiedlichen 'Zuständen' des Programms. Die 'Hauptschleife', die alle Eingaben des Benutzers sammelt und für alle Ausgaben zuständig ist, heißt **mainloop()**. Vorsicht: mainloop niemals innerhalb der

IDLE aufrufen und die IDLE im Single-Task Modus starten.

(Die Tkinter-Programmierung ist einfacher als etwa die Java-Programmierung: dort ist der mainloop gleich dem Aufbau des Fensterinhaltes und damit noch weiter vom 'Programmablauf' entfernt.)

6.) Als fertiges, alleinstehendes Programm vergeben wir die Endung .pyw und aktivieren den mainloop().

```
from Tkinter import *

def fenster(parent):
    l = Label(parent, text="hallo!")
    l.pack()

root = Tk()
fenster(root)
# root.mainloop()
```

Für ein komplexeres Beispiel müssen Informationen zwischen Widgets ausgetauscht werden. Entweder verwenden wir globale Variable, oder wir benützen die elegante objektorientierte Programmierung.

Als Beispiel ein Fenster mit einem Label, das die Anzahl der Clicks auf eine Button mitzählt.

Zuerst die 'einfache' Variante

```
from Tkinter import *

def fenster(parent):
    global l
    global anzahl
    anzahl = 0
    l = Label(parent, text="0", font=("Arial",30) ,padx=20, pady=20)
    l.pack(side=TOP)
    b = Button(parent, text="click mich!", font=("Arial",20) ,padx=20, pady=20,
               command=anzeige)
    b.pack(side=TOP)

def anzeige():
    global l
    global anzahl
    anzahl += 1
    l.configure(text=str(anzahl))

root = Tk()
fenster(root)
# root.mainloop()
```

und hier mit Klasse:

```
from Tkinter import *

class GUI:

    def __init__(self,parent):
        self.anzahl = 0

        self.l = Label(parent, text="0", font=("Arial",30) ,padx=20, pady=20)
        self.l.pack(side=TOP)

        b = Button(parent, text="click mich!", font=("Arial",20) ,padx=20, pady=20,
                   command=self.anzeige)
```

```

        b.pack(side=TOP)

    def anzeige(self):
        self.anzahl += 1
        self.l.configure(text=str(self.anzahl))

root = Tk()
fenster = GUI(root)
# root.mainloop()

```

Die wichtigsten Widgets:

Tk()	das toplevel widget, von Windows verwaltet
Frame()	Rahmen zur Gruppierung weiterer widgets
Label()	widget zum Anzeigen von Text oder Bildern
Button()	Schaltfläche
Entry()	einfache Texteingabe durch den Benutzer
Radiobutton()	button-set, ein einziger ist gewählt
Checkbutton()	button-set, jeder einzelne frei wählbar
Menu()	Menüleiste
Message()	Anzeige von mehrzeiligem Text
Text()	Anzeige von Text (und weiteren Elementen), Inhalt durch den Benutzer editierbar
Canvas()	2D Zeichenbereich (Leinwand) für geometrische Objekte
Scrollbar()	Laufleiste für scrollbaren Text, Canvas,...
Listbox()	Auswahlliste
Scale()	Schieberegler

Fonts

Tk verwendet font-Deskriptoren, da der exakte Wortlaut eines Fontnamens häufig sehr kompliziert ist (speziell Adobe Postscript). Wir geben ein Tupel an, das die Fontfamilie, die Größe und optional den Stil enthält (freilich könnte man auch einen genauen X Window descriptor angeben).

z.B. ("Arial",20), ("Verdana",8,"italic"), ("Times",12,"bold")

Farben

Wir geben sie als Textstring an, wie man es aus HTML gewohnt ist: #RGB, #RRGGBB für 4bit und 8bit Farbkanäle, #RRRRGGGGBBBB für 16bit.

Eine sehr praktische Sammlung weiterer Widgets, die nur auf Tkinter aufbaut, sind die **Python Megawidgets**. Man kann sie sich als 'Pmw' gratis herunterladen und installieren. Man erhält damit Balloon-Help, hübsche Combo-Boxes, einfachere Menüs, NoteBooks (Karteireiter) etc. sowie einen Mechanismus, neue Megawidgets zu entwerfen. Die Widgets sind oft komfortabler zu programmieren als ihre Tkinter-Pendants.

Hier eine Liste häufig benötigter Optionen. (Nicht jedes Widget besitzt jede Option!)

bg background	Hintergrundfarbe	
fg foreground	Vordergrund(Text)Farbe	
bd borderwidth	Rahmendicke. Wichtig bei nichtflachen Reliefs. Wird zu width und height addiert.	
relief	Relieftyp für den Rahmen. RAISED, SUNKEN, FLAT, RIDGE, SOLID, GROOVE	
font	zu verwendende Schriftart	
cursor	Typ des Mausursors, wenn er über dem Widget ist	
width height	Breite des Widgets. Achtung: default kollabiert Höhe	
justify	horizontale Ausrichtung des Inhaltes. RIGHT LEFT CENTER	
anchor	Plazierung des Inhalts. N NE S E W ,...	
command	Adresse der aufzurufenden Aktion	
padx pady	zusätzlicher Abstand um den Inhalt des Widgets (innerhalb von border)	
text	anzuweisender Text	
wrlength	maximale Zeichenzahl pro Zeile für mehrzeilige Textausgaben	
bitmap image	schwarzweißes Bild, eine X11-bitmap Farbbild (Pixmap), gif oder PNG. Mit 'PIL' auch jpg etc.	

Grafiken:

Bilder müssen zuerst als Python-Objekt erzeugt werden, erst dann können sie in Widgets eingesetzt werden.

```
Label(parent, bitmap="meinbild", background=None, foreground="#F00").pack()

img = Photoimage(file="MeinBild.gif")
Label(parent, image=img).pack()
```

Canvas:

wichtige Grafikobjekte (ohne Spezialoptionen)

create_arc(x0,y0,x1,y1, start=0, extent=90)	umschließendes Rechteck und Winkelangaben
create_line(x1,y1,x2,y2)	Mehr Koordinaten ergeben einen Linienzug
create_oval(x0,y0,x1,y1)	Ellipse im umschließenden Rechteck
create_rectangle(x0,y0,x1,y1)	Rechteck
create_text(x0,y0, text="...", font=(...))	

create_polygon	ausgefüllter Linienzug

einige Optionen:

outline= Umrissfarbe
fill= Farbe für Inhalt
width= Strichstärke

```

canvas = Canvas(parent, width=400, height=300, bg="#BBB")
canvas.create_line(100,100,200,200, width=3, fill="#00F")

line1 = canvas.create_line(100,100,200,200, width=1, fill="#000")
line1.coords(100,200,200,100)
line1.itemconfigure(width=8)

```

Zuerst wird der Canvas als Zeichenbereich erzeugt. Dann können wir als Aufruf von Canvas-Methoden die grafischen Elemente einfügen. Jedes dieser Elemente ist ein Objekt – über seine Methode coords() können wir seinen Ort, über itemconfigure() sei Aussehen ändern, wenn wir seine Referenz in einer Variable abgespeichert haben.

Packer

pack – für einfach gruppierte Anordnungen	
side	Seite, an die der Inhalt geschoben wird. LEFT RIGHT TOP BOTTOM
padx pady	zusätzlicher Außenabstand (wie CSS-Margin)
ipadx ipady	zusätzlicher Innenabstand (wie CSS-Padding)
expand	ob das Element ausgedehnt werden soll. X Y BOTH
fill	ob der zur Verfügung stehende freie Platz genutzt werden soll, falls die Elemente größer sind. NO YES

grid – für Anordnungen in einem festen Raster	
column row	in welche Spalte und Zeile das Element eingetragen werden soll
columnspan rowspan	falls ein Element mehrere Zellen beansprucht
padx pady	

place – speziell für aus Grafiken zusammengesetzte GUIs	
x y	Koordinaten, an die das Element gesetzt werden soll
relx rely	Relative Koordinaten (0 bis 1) der Platzierung bezüglich master
anchor	Verankerung der x/y Koordinate im Fenster (CENTER, N, NE,...)
width, height relwidth, relheight	Breite und Höhe absolut Breite und Höhe relativ bezüglich master widget

Tipps:

- Wie zentriert man ein widget in einem anderen?

```
mymaster = Frame(parent,width=500,height=400) # umgebender frame
mymaster.pack_propagate(0) # frame behält seine Größe
w = .....Widget.... # zu zentrierendes Element
w.place(relx=0.5, rely=0.5, anchor=CENTER) # zentriert einbauen
```

- Wenn Du mit Bildern arbeitest: eine Bilddatei ist nur so lange im Zugriff, wie die Variable lebt, welche sie beschreibt.

```
class SuperButton(Button):
    def __init__(self,...):
        . . . . .
        pic = PhotoImage(file="bild.gif")
        self.configure(image=pic)
        . . . . .
```

klappt nicht wunschgemäß. Das Bild blitzt kurz auf, dann ist es verschwunden – vom garbage collector ausgemustert. Mit einem Klassenattribut behält man eine dauerhafte Referenz auf das Bild (eigentlich klar...):

```
self.pic = PhotoImage(file="bild.gif")
self.configure(image=self.pic)
```

(Ohne Klasse muss man eine globale Variable benutzen)