

# Programming Python

## Teil II: Graphisches Programmieren mit Tkinter

Heiko Schröder<sup>1</sup>

**Version 0.6**

<sup>1</sup>Nach der englischen Online Vorlage »An introduction to Tkinter« von Fredrik LUNDH (siehe <http://www.pythonware.com/library/tkinter>)



# Vorwort zur Version 0.6

Dies ist die Fortsetzung des ersten Teils »Programming Python, Der Pythonkern«. Es handelt sich dabei genausowenig wie im ersten Teil um eine *Referenz*, als vielmehr um eine möglichst schnelle Art und Weise, eine Übersicht der grundsätzlichen Gedanken zu erhalten. Genauso wie der erste Teil wird dieser zweite noch auf Stil und Rechtschreibung getestet. Dieser Text ist noch nicht ganz vollständig. Es fehlen noch z.B. *Canvas-Widgets* und *Scroll-bars*, die allerdings nach und nach entwickelt werden. So kann es sein, dass Einiges von diesen fehlenden Dingen schon in der Online-Version vorhanden ist, aber *noch nicht* in den herunterladbaren Dateien.

In dieser Version wurden die Canvases und die Turtle hinzugefügt.

Das Release-Schedule sieht aus wie folgt:

- Version 0.7x: Abschluss der fehlenden Teile. Technische Versionen
- Version 0.8: Korrigierte Fassung (hoffentlich keine Rechtschreibfehler mehr)
- Version 0.9x: Von den Schülerinnen des 11. Jahrgangs des EvB Grosshansdorf freigegebene Fassungen. Lediglich stilistische Korrekturen.

Freigabe der Final-Version 1.0 ist für den Mai 2003 geplant.

Zu diesem Text gehört die gepackte Zip-Datei *tkinexam.zip*, welche die in diesem Text vorkommenden Beispiele enthält.

Als Voraussetzung, diesen Text zu verstehen, sei dringend das Kapitel »Klassen« des 1. Teils, Pythonkern, empfohlen.

# Inhaltsverzeichnis

<b>1</b>	<b>Das Turtle-Modul</b>	<b>VIII</b>
1.1	Turtle-Beispiel . . . . .	IX
1.2	Das Füllen von geschlossenen Figuren . . . . .	X
1.3	Turtle-Funktionen . . . . .	XII
1.3.1	Geradlinige Bewegung . . . . .	XII
1.3.2	Drehung, Winkel, Kreis . . . . .	XIII
1.3.3	Erscheinungsbild und Konfiguration . . . . .	XIII
1.3.4	Bildschirmprozeduren . . . . .	XIV
1.3.5	Sonstiges . . . . .	XIV
1.4	Turtle-Fenster sind <i>Canvas</i> . . . . .	XV
1.5	Turtle-Klassen . . . . .	XV
<b>2</b>	<b>Was ist Tkinter</b>	<b>1</b>
<b>3</b>	<b>»Hallo, Tkinter«: Das erste Programm</b>	<b>2</b>
3.1	Starten des ersten Programms . . . . .	2
3.2	Details . . . . .	3
<b>4</b>	<b>Das zweite Programm</b>	<b>5</b>
4.1	Starten des Programms . . . . .	6
4.2	Details . . . . .	6
4.3	Radiobutton . . . . .	8

<i>INHALTSVERZEICHNIS</i>	IV
<b>5 Tk_Intern - die Übersicht</b>	<b>9</b>
<b>6 Tk_Intern: <i>Canvas</i></b>	<b>13</b>
6.1 Zeichnen von Ellipsen und Kreisen . . . . .	14
6.2 Arbeiten mit den gezeichneten Objekten . . . . .	15
6.2.1 Konfigurieren eines Items . . . . .	15
6.2.2 Löschen eines Items . . . . .	15
6.2.3 Bewegen und Verändern eines Items mit Hilfe von Tags	16
6.2.4 Überdecken und Verstecken von Items . . . . .	17
6.2.5 Das <i>all</i> und das <i>current</i> -Tag . . . . .	18
6.3 Weitere Canvas-Items . . . . .	18
6.4 Canvas und Turtle . . . . .	20
6.5 Canvas und Scrollbars . . . . .	22
6.6 Canvas und Events . . . . .	24
<b>7 Allgemeine Konfiguration von Widgets</b>	<b>27</b>
7.1 Farbe . . . . .	28
7.2 Schriftsatz . . . . .	30
7.2.1 Eigene Fonts als aufrufbare Instanzen . . . . .	30
7.3 Textformatierung . . . . .	31
7.4 Rahmen . . . . .	31
7.5 Aussehen des Cursors . . . . .	32
7.6 Referenz zum Styling von Widgets . . . . .	33
7.6.1 Häufige Optionen, die von fast allen Widgets unter- stützt werden . . . . .	34
7.6.2 Seltene Optionen, die nicht von allen Widgets unter- stützt werden . . . . .	35

<b>8 Ereignisse (Events) und Bindungen</b>	<b>37</b>
8.1 Arten von Events . . . . .	38
8.2 Tastenbindungen . . . . .	40
8.3 Das Event-Objekt und seine Attribute . . . . .	40
8.4 Instanzen- und Klassenbindungen . . . . .	41
8.5 Protokolle . . . . .	42
<b>9 Anwendungsfenster</b>	<b>44</b>
9.1 Base Windows . . . . .	44
9.2 Menüs . . . . .	45
9.3 Werkzeugleisten (Toolbars) . . . . .	47
9.4 Statuszeilen . . . . .	48
<b>10 Standard Dialoge und Message-Boxen</b>	<b>50</b>
10.1 Message-Boxes . . . . .	50
10.1.1 showinfo . . . . .	51
10.1.2 showwarning . . . . .	52
10.1.3 showerror . . . . .	53
10.1.4 Question-Boxes . . . . .	54
10.1.5 Message-Box Optionen . . . . .	54
10.2 Dateneingabe . . . . .	55
10.2.1 <i>strings</i> mit <i>askstring</i> . . . . .	55
10.2.2 <i>numerical values</i> und <i>askinteger</i> und <i>askfloat</i> . . . . .	56
10.2.3 <i>filenames</i> . . . . .	57
10.2.4 <i>colors</i> . . . . .	58

# Abbildungsverzeichnis

1.1	Turtle in Ausgangsposition . . . . .	IX
1.2	Zeichnen von Rosetten mit der <i>Turtle</i> . . . . .	X
1.3	Füllen von Kreisen . . . . .	XII
1.4	Turtle-Zeichnungen mit drei verschiedenen Stiften . . . . .	XVI
3.1	Das erste Programm <i>hello1.py</i> . . . . .	3
4.1	Programm mit Buttons . . . . .	6
6.1	Verschiedene Canvases . . . . .	21
6.2	Turtle vom Typ <i>RawPen()</i> in Aktion . . . . .	23
6.3	Canvas mit Scrollbars . . . . .	24
9.1	Einfaches Menü . . . . .	46
9.2	Menü mit Icons . . . . .	48
10.1	MessageBox . . . . .	52
10.2	Warnungsbox . . . . .	53
10.3	Errorbox . . . . .	54
10.4	Fragebox . . . . .	56
10.5	<i>askopenfilename</i> und die zugehörige Box . . . . .	58
10.6	Farbauswahl mit <i>askcolor</i> . . . . .	59

# Tabellenverzeichnis

5.1	Übersicht der Tkinter Widgetklassen . . . . .	10
5.2	Übersicht der Geometrie-Manager . . . . .	11
5.3	Übersicht der Tkinter-Klassen, die keine Widgets darstellen .	11
5.4	Übersicht zusätzlicher Module für Tkinter . . . . .	12



# Kapitel 1

## Das Turtle-Modul

Das *Turtle*-Modul ist die einfachste Methode, um Zeichnungen mit Python zu programmieren. Die Turtle selbst war ursprünglich eine programmierbare »Schildkröte«, die nach entsprechenden Befehlsvorgaben auf dem Boden hin- und herlaufen konnte und, falls der Zeichenstift abgesenkt war, ihren zurückgelegten Weg aufzeichnete. Diese Idee wurde während der 80-er Jahre in Dänemark entwickelt und war das erfolgreichste Konzept der Sprache *Comal*, die einst als Programmiersprache für Schulen entwickelt wurde und aus einem Konglomerat aus Basic-Elementen und Pascal bestand. Ähnlich Python war Comal eine Interpretersprache, die aber bei *weitem* nicht so gut durchdacht war und eher darauf ausgerichtet zu sein schien, mit den geringsten Mitteln möglichst schnellen Erfolg zu erzielen, auch auf Kosten eines sauberen Programmierstils<sup>1</sup>.

Um die Turtle zu aktivieren, muss das Modul geladen werden, zum Beispiel durch

```
import turtle
```

oder noch besser

```
from turtle import *
```

---

<sup>1</sup>Zum Beispiel konnte Comal erkennbare syntaktische Fehler der Schüler von selbst berichtigen. Dahinter stand die durchaus richtige Ansicht, dass das Auswendiglernen einer Syntax von geringem Wert ist. Andererseits ist es aber ein schlechtes Zeichen, wenn Syntax überhaupt auswendiggelernt werden muss und nicht so natürlich und selbstverständlich wie ein Tanzschritt daherkommt, wie es bei Python der Fall ist.

Wir gehen jetzt im Folgenden davon aus, dass das *turtle*-Modul mit »from« importiert wurde.

Die Turtle ist dann noch nicht sichtbar. Geben Sie noch zusätzlich

```
>>>reset()
```

ein, so erscheint das in Abbildung 1.1 gezeigte Bild. Die Turtle wird durch

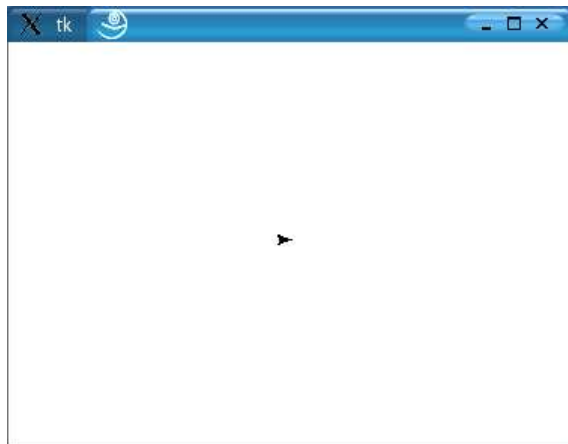


Abbildung 1.1: Turtle in Ausgangsposition

ein kleines Dreieck dargestellt. Sein Zeichenstift befindet sich im Gegensatz zu den Verhältnissen bei Comal an der Spitze. Führen Sie nun noch

```
>>>demo()
```

aus, um zu erkennen wie ein Zeichenvorgang aussieht.

## 1.1 Turtle-Beispiel

```
#File: turtle1.py
from turtle import *
def neck(eckenzahl, laenge, breite, col):
    winkel=360/eckenzahl
    width(breite)
```

```

color(col)
for i in range (eckenzahl):
    forward (laenge)
    left(winkel)
def rosette(eckenzahl, laenge, breite, col, drehwinkel):
    for i in range(360/drehwinkel):
        neck(eckenzahl, laenge, breite, col)
        left(drehwinkel)

```

Mit den Funktionen dieses Moduls können schöne Rosetten gezeichnet werden, sofern die Eckenzahl ein Teile von  $360^\circ$  ist. Denken Sie daran, nach `import turtle1` auch `from turtle1 import *` folgen zu lassen. Die Eingabe von

```

>>>rosette(10,40, 1, 'blue', 36)
>>>rosette(5, 80, 1, 'red', 36)

```

erzeugt das Bild Nr. 1.2.

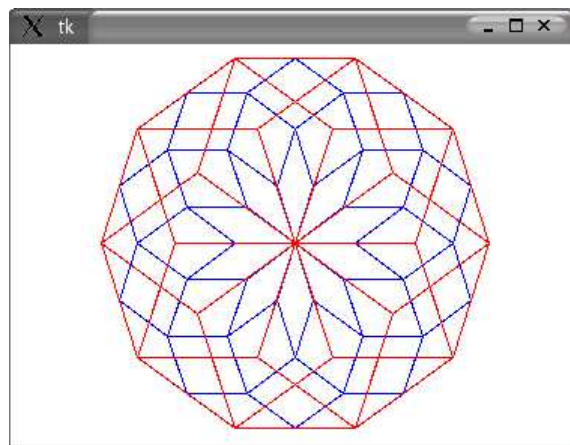


Abbildung 1.2: Zeichnen von Rosetten mit der *Turtle*

## 1.2 Das Füllen von geschlossenen Figuren

Für das Erzeugen eines Kreises dient in der Regel die Funktion `circle`, für die man nur einen Radius angeben muss. Die Funktion `fill(flag)` sorgt nun

für das Füllen mit einer inneren Farbe, wobei es mir noch nicht gelungen ist, herauszufinden, ob auch noch ein Füllmuster ausgewählt werden kann. Probieren Sie bitte das Folgende einmal aus:

```
#File: turtle3.py
from turtle import *
from turtle1 import *
up()
goto(10,10)
down()
color('red')
width(2)
fill(1)
circle(50)
color('maroon')
goto(20,20)
fill(0)
#Zweiter Kreis
up()
goto(-70,-60)
down()
fill(1)
neck(360,0.5,2,'blue')
color('green')
fill(0)
tracer(0)
```

Das Ergebnis zeigt Bild 1.3

Im ersten Fall wird gezeigt, dass *fill(1)* vor der gezeichneten Figur eingegeben werden muss. Es bedeutet: Achtung, die nächste Figur soll gefüllt werden. Dann *muss* der Zeichenstift irgendwie ins Innere der Figur bewegt werden, bevor mit *fill(0)* der Füllvorgang tatsächlich ausgeführt wird.

Wer dieses »ins Innere führen« des Zeichenstiftes nicht so gerne hat, kann natürlich auch ein *360-Eck* mit Hilfe der in *turtle1* geschriebenen Funktion *neck* durchführen, wobei allerdings die Schwierigkeit besteht, den Radius im Griff zu haben, wobei sich der natürlich auch (ab 10. Klasse) mit Hilfe von *sin()* bestimmen lässt. Aber es ist vielleicht gut, zu wissen, dass für die Seitenlänge des Ecks auch gebrochene Zahlen angegeben werden dürfen. Beim Winkel sieht das etwas anders aus. Vermeiden Sie immer gebrochene

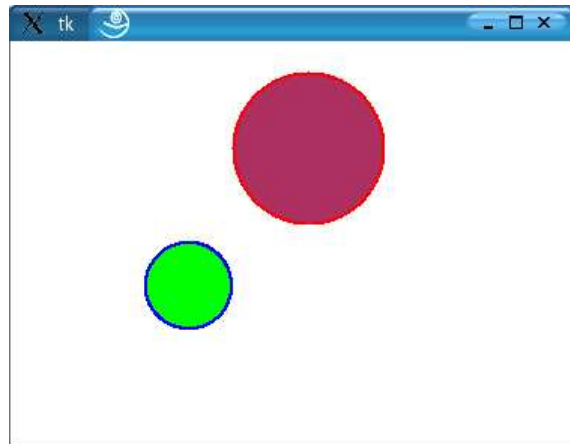


Abbildung 1.3: Füllen von Kreisen

Winkel für *neck*! Ein 99-Eck ergibt mit dieser Funktion *keinen* vollständigen Kreis!

## 1.3 Turtle-Funktionen

Die Funktionen des Turtle-Moduls sind ohne Ausnahmen Prozeduren, d.h. sie geben kein Objekt zurück. In den folgenden Abschnitten sind diese Prozeduren ihrer Aufgabe nach, nicht der alphabetischen Ordnung nach aufgeführt. Im Wesentlichen kann die Turtle Bewegungen mit oder ohne abgesenkten Zeichenstift ausführen. Darüberhinaus kann natürlich die Zeichenfarbe verändert werden. Eine besonders einfache Methode dient dazu, Text auf das Zeichenbrett zu bringen, die der Funktion *print()* des Standard-Python entspricht.

### 1.3.1 Geradlinige Bewegung

**forward(*distance*):** Prozedur, die die Turtle um *distance*-Pixel nach vorne bewegt.

**backward(*distance*):** Prozedur, die die Turtle um *distance*-Pixel nach hinten bewegt.

**goto(x,y):** Prozedur, die die Turtle an die Stelle  $(x,y)$  bewegt. Dabei sind  $x$  und  $y$  absolute Koordinaten. Wie auch bei der Farbeinstellung *color* kann  $(x,y)$  auch als ganzes Tupel an *goto* übergeben werden, also *goto((x,y))*.

### 1.3.2 Drehung, Winkel, Kreis

**left(*angle*):** Dreht die Turtle um den Wert *angle* nach links. Voreingestellt ist, dass *angle* in Altgrad interpretiert wird.

**right(*angle*):** Dreht die Turtle um den Wert *angle* nach rechts.

**degrees():** Prozedur, welche dafür sorgt, dass Winkelangaben in den folgenden Anweisungen als Altgrad interpretiert werden. Das ist voreingestellt.

**radians():** Prozedur, welche dafür sorgt, dass Winkelangaben in den folgenden Anweisungen als Radiant interpretiert werden.

**circle(*radius*):** Prozedur, die einen Kreis mit vorgegebener Farbe zeichnet. Wenn der Kreis nicht vollständig sein soll, kann durch einen zweiten Parameter *extent* in Grad vorgegeben werden, wie gross der Bogen sein soll. Beispielsweise zeichnet *circle(50,90)* einen Viertelkreis mit dem Radius 50.

### 1.3.3 Erscheinungsbild und Konfiguration

**tracer(*flag*):** Prozedur, die den »Tracer« ein und ausschaltet. *Tracing* bedeutet »Verfolgung«, d.h. die Linien werden langsamer gezogen als üblich und während des Zeichnens ist die Turtle sichtbar. Das *flag* kann die Werte 1 (für »wahr«) und 0 (für »falsch«) annehmen. Es bedeutet *tracer(1)* sozusagen ein *show\_turtle* und *tracer(0)* ein *hide\_turtle*-Befehl.

**up():** Prozedur, die den Schreibstift hochstellt. Jede Bewegung erfolgt also ohne Zeichnung.

**down():** Prozedur, die den Schreibstift absenkt. Die folgenden Bewegungen werden aufgezeichnet.

**width(*width*):** Prozedur, die die Strichdicke auf *width* Pixel festlegt.

**color(*s*):** Prozedur, die die Zeichenfarbe festlegt. *s* muss dabei ein Name als String sein, zum Beispiel *color('red')*.

**color(*r,g,b*):** Prozedur, die die Zeichenfarbe festlegt. Es müssen drei Parameter übergeben werden, die allesamt zwischen 0 und 1 liegen. Es ergibt zum Beispiel *color(0.2,0.2,0.1)* einen braunen Farbton. Es kann alternativ auch *(r,g,b)* als ganzes Tupel übergeben werden, wenn das erforderlich ist. So ist zum Beispiel *color((0.2,0.2,0.1))* auch möglich.

**fill(*flag*):** Prozedur, die einen folgenden geschlossenen Pfad mit der voreingestellten Farbe füllt. Üblicherweise ruft man vor der zu erstellenden Figur zunächst *fill(1)* auf, zeichnet sie dann und fügt am Ende *fill(0)* hinzu. Danach sollte die Figur gefüllt sein. Hier muss noch genauer nachgeforscht werden, wie eine Figur mit andersfarbigem Rand und bestimmtem Muster gezeichnet werden kann. Kreise können zunächst einmal auch noch nicht gefüllt werden.

**write(*text*):** Schreibt *text* an die aktuelle Cursorposition. Dabei muss *text* nicht unbedingt ein String sein. Andere Objekte sind auch möglich. *write* funktioniert ähnlich wie *print*. Optional kann als zweiter Parameter *move* eine 1 oder eine 0 gesetzt werden, je nachdem ob man wünscht, dass der Stift am Ende des Textes stehenbleiben soll oder nicht. Defaultmässig ist *move* auf »false«, also 0, eingestellt. Der Schreibstift kehrt also immer an den Anfang des Textes zurück.

### 1.3.4 Bildschirmprozeduren

**clear():** Prozedur, die den Bildschirm löscht.

**reset():** Prozedur, die den Bildschirm löscht, die Turtle wieder an den Anfang setzt und alle Variablen auf die defaultmässig voreingestellten Werte zurücksetzt.

### 1.3.5 Sonstiges

- Natürlich stehen *pi* und *e* als Konstante zur Verfügung.
- Die Prozedur *demo()* zeigt eine Demonstration der Turtle. Der Code dieser Funktion ist ebenfalls sehr aufschlussreich.

## 1.4 Turtle-Fenster sind *Canvas*

Nach diesen kleinen Demonstrationen bleibt die Frage, ob man hier nicht bereits alles vorfindet was das Herz begehrt. In der Tat lassen sich so auf einfache Weise vielfältige Zeichnungen erstellen. Nur: diesen Zeichnungen können keine Aufgaben zugewiesen werden. In den folgenden Abschnitten werden wir unterscheiden müssen zwischen:

- sogenannten *Items*, die zwar Objekte darstellen, aber nicht mit irgendwelchen ausführbaren Kommandos, sogenannten *Events*, versehen werden können,
- sogenannten *Widgets*<sup>2</sup>, die Grafiken darstellen, die mit einem Kommando verknüpft werden können und gegebenenfalls selbst wieder andere Widgets oder Items enthalten können.

Eine besondere Widgetklasse, die ein Fenster darstellt, in dem Zeichnungen erstellt werden können, nennt sich *Canvas*. Ein Canvas ist also ein Zeichenbrett, mit dem vor allen Dingen statische, grafische Elemente erstellt werden können. Die in Tkinter eingebettete Widgetklasse *Canvas* ist etwas flexibler als die Turtle. Die Turtle wiederum ist besonders einfach zu bedienen und kann präziser Zeichnungen erstellen

## 1.5 Turtle-Klassen

Nun besteht sicherlich mitunter der Wunsch, mehr als ein Canvas zur Verfügung zu haben. Deshalb lassen sich mit Hilfe der Klasse *Pen()* beliebig viele weitere Turtle generieren. Darüberhinaus gibt es noch die Klasse *RawPen(canvas)*, die es ermöglicht, innerhalb eines existierenden Canvas-Objektes ein Turtle-Canvas zu erstellen. Sämtliche Funktionen des Abschnitts 1.3 mit Ausnahme von *demo()* treten stehen den Instanzen als Methoden zur Verfügung.

Ein Beispiel:

```
#File: turtle2.py
```

---

<sup>2</sup>Dies ist ein Kunstwort aus Windows und Gadget, also ein Fenster, das irgendetwas machen kann.



```
from turtle import *
pen=Pen()
pen.up()
pen.color('red')
pen.width(2)
pen.goto(40,40)
pen.down()
pen.circle(40)
pencil=Pen()
pencil.up()
pencil.color('blue')
pencil.width(1)
pencil.goto(-60,-50)
pencil.down()
pencil.circle(40)
up()
color('green')
goto(40,-50)
down()
circle(40)
```

Die Ausführung dieses Programms liefert uns das Bild 1.4. Der dritte Stift ist

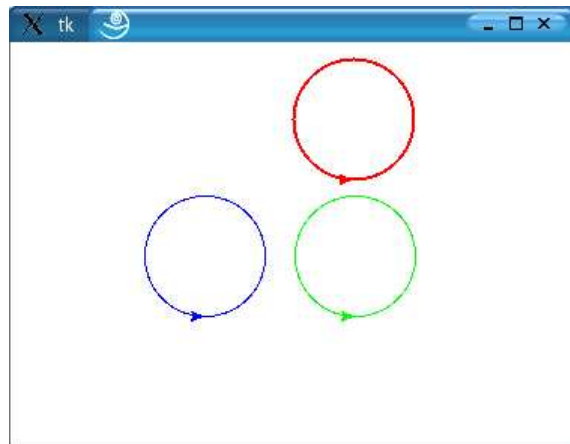


Abbildung 1.4: Turtle-Zeichnungen mit drei verschiedenen Stiften

der defaultmässig verfügbare, der zum Modul selbst gehört. Ein Beispiel für

*RawPen()* hat erst im Zusammenhang mit dem *Canvas*-Widget von Tkinter eine Bedeutung. Daher werden wir dieses Beispiel auf später verschieben.

## Kapitel 2

# Was ist Tkinter

Das *Tkinter* modul (»Tk interface«) ist das standardisierte Python interface zur grafischen Bibliothek Tk, die eigentlich zur Sprache Tcl (sprich: »tickl«) gehört.

Beide Bibliotheken, Tk und Tkinern, stehen sowohl auf den meisten Unix-Plattformen zur Verfügung, als auf Windows und Macintosh-Systemen.

Das Interface selbst befindet sich in ebenfalls in einem Modul mit dem Namen *\_tkinter*. In diesem Modul sind etliche Python-Module enthalten, aber die beiden wichtigsten sind das *Tkinter* Modul selbst und ein Modul, das sich *Tkconstants* nennt. Da das *Tkinter* Modul das *Tkconstants* Modul automatisch importiert, genügt in Python-Programmen im allgemeinen die Zeile

```
import Tkinter
```

oder noch besser:

```
from Tkinter import *
```

## Kapitel 3

# »Hallo, Tkinter«: Das erste Programm

Rufen Sie den *Editor* von IDLE auf und geben Sie in das Editorfenster den Programmtext

```
# File: hello1.py
from Tkinter import *
root=Tk()
w=Label(root, text='Hallo, Tkinter!')
w.pack()

root.mainloop()
```

### 3.1 Starten des ersten Programms

Um das Programm nun zu starten, importieren Sie die Datei »hello1« in gewohnter Weise, indem Sie entweder

```
python hello1.py
```

direkt am Shell-Prompt eingeben (für dieses Beispiel empfohlen), oder bei bereits laufendem Interpreter (z.B. unter IDLE) schreiben Sie einfach

```
import hello1
```

Abbildung 3.1: Das erste Programm *hello1.py*

Dann sollte das in der Abbildung 3.1 gezeigte Fenster sichtbar werden:

Wurde das Programm über den Shellprompt gestartet, lässt es sich dadurch beenden, dass das Kreuz oben rechts an dem Fenster angeklickt wird. Bei IDLE gibt es diesbezüglich manchmal Schwierigkeiten, da sich zwar das Fenster schliesst, aber IDLE selbst nicht wieder zum Interpreterprompt zurückkehrt. Unter Linux muss IDLE dann gewaltsam beendet werden. Empfehlung: Starten Sie unter Linux IDLE am besten über eine Konsole. Durch Eingabe von CTRL-C in dieser Konsole wird nicht etwa ganz IDLE beendet, sondern nur das gerade laufende Programm.

## 3.2 Details

**from Tkinter import \*:** Durch diese Anweisung wird das gesamte Containermodul *Tkinter* in das *main*-Modul des Interpreters ausgepackt. Es stehen dann alle darin enthaltenen Namen, die auf Klassen und ihre Methoden und Attribute verweisen, zur Verfügung.

**root=Tk():** Die Klasse mit dem Namen *Tk* ermöglicht die Generierung von reinen Fenstern (*windows*) als Instanzen. Durch *Tk()* wird der Konstruktor dieser Klasse aufgerufen und generiert eine solche Instanz. Dieser wird der Name *root* zugewiesen.

**w=Label(root text='Hallo, Tkinter'):** Ein *Label* ist nun mehr als ein Fenster. Es ist ein *widget*, was ein Kunstwort aus *window* und *gadget* bedeutet. Das Wort *gadget* ist Ihnen vielleicht nicht geläufig. Es bedeutet ein »Ding, das irgendwas machen kann.« Und hier liegt genau der Unterschied zum reinen Fenster. Ein *widget* ist ein Fenster mit einer ganz bestimmten Funktion, während ein *window* nur den Container für mehrere Fenster darstellt<sup>1</sup>. Das Label hat nun die Funktion, einen

---

<sup>1</sup>Wir werden später noch einen weiteren Containertyp kennenlernen, der sich *Frame* nennt, während *Tk* nur echte Windows generiert.

Text darzustellen<sup>2</sup>. `Label` ist selbst der Name einer Klasse, dessen Konstruktor hier aufgerufen wird, womit eine Instanz erzeugt wird, die den Namen *w* erhält. Der erste Parameter des `Label`-Konstruktors wird mit dem Namen desjenigen Fensters belegt, das das Label aufnehmen soll. Der zweite Parameter versteht sich auf den ersten Blick von selbst, enthält er doch die wesentliche Information. Technisch ist jedoch interessant, dass der Name *text* schon vom Konstruktor defaultmässig einem Objekt zugewiesen wurde<sup>3</sup>, das nun durch den String »Hallo, Tkinter« überschrieben wird. Wir erinnern uns daran, dass solche bereits vorbelegten Parameter nicht unbedingt angegeben werden müssen. In der Tat funktioniert das Programm auch dann, wenn nur allein *root* als Name übergeben wird und eine explizite Angabe des Parameters *text* fehlt. Probieren Sie es aus. Das Programm läuft, aber es fehlt der Text. `Label` hat natürlich noch etliche weitere Parameter, die aber bereits defaultmässig vorbelegt sind.

**w.pack():** Der Schreibweise können Sie sofort entnehmen, dass der Packer *pack* eine Methode aller Instanzen der Klasse *Label* sein muss. Wir werden diese Methode noch genauer untersuchen. Zunächst nur soviel: der Packer gibt an, *wie* die Instanz in dem Containerfenster erscheinen soll. Da keine Parameter angegeben werden, ist die Erscheinung so wie standardmässig vorgegeben.

**root.mainloop():** Die Methode *mainloop()* gehört allen Instanzen vom Typ *Tk* an. Es handelt sich dabei um eine *Event-Schleife*, die auf Mausklicks oder Tastendrücke reagiert. Ohne diese Schleife lässt sich das Programm nicht steuern. Daher *muss* eine solche Schleife immer vorhanden sein. Was sie genau macht, sehen wir ebenfalls später. In diesem Fall wird das Programm beendet, wenn das Fenster geschlossen wird.

---

<sup>2</sup>Es kann allerdings alternativ auch ein *Icon* oder eine *Abbildung* darstellen.

<sup>3</sup>Nämlich `NONE`.

## Kapitel 4

# Das zweite Programm

Wir haben in unserem ersten Programm die grafischen Routinen einfach hintereinander weg geschrieben. Das ist kein guter Stil. Wie wir sahen, haben wir es mit Klassen zu tun, und wir können an dieser Stelle nicht ausgiebig genug betonen:

Schreibe so viel wie möglich eigene Klassen!

Hier ist das zweite Programm:

```
#File: hello2.py
from Tkinter import *
class App:

    def __init__(self, master):
        frame=Frame(master)
        frame.pack()
        self.button=Button(frame, text='Quit', fg='red', command=frame.quit)
        self.button.pack(side=LEFT)
        self.hi_there=Button(frame, text='Hello', command=self.say_hi)
        self.hi_there.pack(side=LEFT)

    def say_hi(self):
        print 'Hi there, everyone!'
root=Tk()
app=App(root)
root.mainloop()
```

## 4.1 Starten des Programms

Wenn Sie dieses Programm starten (siehe Abbildung 4.1 ), so schreibt jeder



Abbildung 4.1: Programm mit Buttons

Klick auf den 'Hello'-Button den Satz 'hi there, everyone!' auf die Konsole. Ein Klick auf den linken Button beendet das Programm, jetzt aber vernünftig, auch bei Verwendung von IDLE.

## 4.2 Details

Fast das gesamte Programm wurde hier als Klasse ausgeführt. Wir fangen mit der Besprechung des Programms etwas unorthodox bei denjenigen Zeilen an, die *nicht* in die Klasse eingebunden sind.

**root=Tk():** Das kennen wir schon. Es wird ein Fenster mit dem Namen *root* eingerichtet.

**app=App(root):** Hier wird es interessant. Durch *App(root)* wird eine Instanz der Klasse *App* erzeugt, die offenbar die gesamte »Applikation« dieses Programms<sup>1</sup> darstellt. Da der Name *root* übergeben wird, *muss* in der Klasse *App* zwangsläufig ein Konstruktor der Form `__init__` existieren. Fehlt eine explizite Angabe dieses Konstruktors, kann nur ein Aufruf der Form *name=App()* erfolgen. Die erzeugte Instanz heisst *app*. Dieser Name wird allerdings im weiteren Verlauf nicht benötigt. Unterscheiden Sie deutlich zwischen dem *Klassennamen* *App* und dem Instanznamen *app*. Schauen wir uns nun den Inhalt der Klasse etwas genauer an:

**class App:** Diese Zeile markiert nicht etwa nur den Anfang der Klassenfestlegung, sondern ist selbst eine Anweisung. Der Operator *class* generiert eine Klassenobjekt, dem der Name *App* zugewiesen wird. Welche Namen sind nun lokal in der Klasse verfügbar?

---

<sup>1</sup>Applikation=Anwendung



1. *self*: Wie in jeder Klasse üblich, ist *self* die Schnittstelle der Klasse zu den erzeugten Instanzen. Wird eine Instanz - in diesem Falle also *app* - erzeugt, so weist der *self*-Parameter auf den Namen *app*.
2. `__init__`: Dies ist wie gesagt eine Funktion, die den Konstruktor *App()* neu definiert. Der Name `__init__` ist also immer mit dem Klassennamen - in diesem Falle *App* - identisch.
3. Der dritte Name gehört einer Methode<sup>2</sup>, die *say\_hi* heisst.
4. Immer etwas versteckt sind die Attribute einer Klasse. Es handelt sich dabei um diejenigen Datenobjekte, die über einen Namen der Gestalt *self.name* definiert werden. Wir sehen hier zwei Namen, *self.button* und *self.hi\_there*, die auf Instanzen der Klasse *Button* verweisen

**def `__init__`(self, master):** Noch einmal zurück zum Konstruktor.

**frame=Frame(master):** Mit dieser Anweisung generiert der Konstruktor eine Instanz der *Tk*-Klasse *Frame*, die wir im ersten Beispiel schon kurz erwähnten. Ein *Frame* ist ein *widget*, das beliebig viele weitere Widgets enthalten kann. Eine *Frame*-Instanz muss aber immer in einem Fenster (*window*) eingebettet sein. Daher *muss* dem Konstruktor von *Frame* der Name des Hauptfensters (hier *master* genannt) übergeben werden.

**self.button=Button(...):** Objekte der Klasse *Button* unterscheiden sich von Objekten der Klasse *Label*, wie wir sie im ersten Beispiel betrachtet haben, im wesentlichen dadurch, dass der Button noch ein Kommando ausführen kann. Durch *command=frame.quit* zeigt der Name *command*, der zu der Klasse *Button* gehört nicht mehr länger auf das voreingestellte Objekt *NONE*, sondern auf den Namen einer Methode der *Frame*-Instanz *frame*. Diese Systemmethode ist in der *Frame*-Klasse als *self.quit* festgelegt und hat keine weitere Aufgabe als das Instanzobjekt zu zerstören<sup>3</sup>. Der Parameter *fg* (foreground) dient der

---

<sup>2</sup>Eine Methode ist eine Funktion, die zu einer Klasse gehört.

<sup>3</sup>Die eigentliche Zerstörung erfolgt durch Pythons Garbage-Collector. Die Methode *self.quit* ist quasi ein Destruktor.

Konfiguration des Buttons und existiert auch für *Label*-Instanzen.

**self.button.pack(side=LEFT):** Hier sehen wir zum ersten Mal, dass dem Packer eine Information über die Lage der Instanz in dem umgebenden Fenster übergeben wird: der Name *side* des Packers zeigt jetzt auf das Objekt *LEFT*, das ein im Packer definierte Methode ist, die das Objekt an die linke Position des umgebenden Fensters setzt.

**self.hi\_there=Button(...):** Diese Anweisung zeigt, wie man eine selbst geschriebene Methode durch Buttonklick ausführen kann. Dabei wird dem Namen lediglich der Name dieser Methode übergeben. Wir sehen, dass die Methode *say\_hi* später in der Klasse definiert wird<sup>4</sup>. Natürlich lautet der Name für den Aufruf innerhalb der Klasse *self.say\_hi* und nicht einfach nur *say\_hi*, was ein Name wäre, der nur innerhalb des Konstruktors seine Gültigkeit hätte.

**self.hi\_there.pack(side=LEFT):** Etwas merkwürdig ist es, dass nun auch der Button *self.hi\_there* nach links gepackt werden soll, wo sich doch bereits dort der Button *self.button* befindet. Richtig, der Packer bezieht sich in der Positionsangabe immer auf den *freien* Platz des umgebenden Fensters.

**root.mainloop():** Hier wird wieder die Event-Schleife des Hauptfensters aufgerufen (siehe Beispiel 1).

### 4.3 Radiobutton

---

<sup>4</sup>Früher ist aber auch zulässig!

## Kapitel 5

# Tk\_Intern - die Übersicht

Mit Tk\_Intern wird in diesem Tutorial eine Übersicht bezeichnet, die möglichst alles das was zusammengehört auch zusammen beschreibt. Manche Dinge kommen in späteren Kapiteln noch einmal aus anderer Sicht betrachtet vor.

Wie wir schon sahen, wird das oberste, das *root*-Widget durch die Klasse *Tk* festgelegt. In Tkinter sind nun insgesamt nur noch 15 weitere Widgetklassen eingebaut, was die ganze Sache sehr übersichtlich macht. Tabelle 5.1 zeigt diese Widgetklassen in alphabetischer Reihenfolge.

Die nächste Gruppe bilden die drei Geometrie-Manager, die einzelne Widgets innerhalb ihres umgebenden Widgets anordnen. Diese Manager sind in der Tabelle 5.2 gezeigt.

Zusätzlich enthält Tkinter noch separate Klassen, die keine Widgets darstellen, wie die Tabelle 5.3 zeigt.

Als Ergänzung zu Tkinter stehen noch weitere Module zur Verfügung, die in der Tabelle 5.4 angegeben sind, und separat geladen werden müssen.

Widgetklasse	Beschreibung
<i>Button</i>	Ein anklickbarer Knopf, der ein Kommando oder eine Operation durchführt.
<i>Canvas</i>	Eine sehr beliebte Klasse, die beliebige grafische Elemente auf dem Bildschirm generieren kann (siehe 6).
<i>Checkbutton</i>	Dieser Button ist mit einem Objekt verknüpft, das zwei Werte enthalten kann: entweder ist der Button angeklickt oder nicht.
<i>Entry</i>	Ein Textfeld zum Eingeben von Texten.
<i>Frame</i>	Ein Container-Widget mit Border und Background zur Aufnahme weiterer Widgets.
<i>Label</i>	Gibt einen Text auf dem Bildschirm aus.
<i>Listbox</i>	Zeigt eine Liste mit Auswahlpunkten, die über Checkbuttons oder Radiobuttons erfolgen kann.
<i>Menu</i>	Eine Klasse, welche die Generierung von Pulldown und Popup Menüs ermöglicht.
<i>Menubutton</i>	Diese Klasse wird zur Implementierung von Pulldown-Menüs benötigt.
<i>Message</i>	Dieses Objekt gibt einen Text aus, ähnlich wie Label, bricht den Text aber auch automatisch um, je nach vorgegebenen Randbedingungen.
<i>Radiobutton</i>	Radiobuttons sind alle an ein und dasselbe Objekt gebunden, während Checkbuttons jeweils an ein eigenes Objekt gebunden sind. Wenn ein Radiobutton angeklickt wurde, so enthält das Objekt die Information des angeklickten Inhalts, was bedeutet, dass der vorher angeklickte Radiobutton wieder zurückgesetzt wird. Radiobuttons sind also im Gegensatz zu Checkbuttons voneinander abhängig.
<i>Scale</i>	Ermöglicht das Übergeben eines numerischen Objektes, indem ein umrandetes Objekt durch Ziehen mit der Maus vergrößert und verkleinert wird.
<i>Scrollbar</i>	Generiert Rollbalken für Instanzen der Klassen <i>Canvas</i> , <i>Entry</i> , <i>Listbox</i> und <i>Text</i> .
<i>Text</i>	Ermöglicht die Ausgabe von formatiertem Text.
<i>Toplevel</i>	Toplevel Widgets sind den Frame Widgets sehr verwandt, werden jedoch in einem gesonderten Fenster dargestellt.

Tabelle 5.1: Übersicht der Tkinter Widgetklassen

Manager	Beschreibung
<i>Grid</i>	Ein Geometrie-Manager, der Widgets in einer zwei-dimensionalen Tabelle einordnet. Das Master-Widget wird dabei in eine Anzahl von Reihen und Spalten eingeteilt und jede so entstehende Zelle kann ein Widget enthalten.
<i>Place</i>	Der einfachste und älteste der drei Manager. Er setzt Widgets innerhalb eines Eltern-Widget an eine Position, die durch absolute oder relative Koordinaten angegeben wird. In der Regel ist <i>Pack</i> vorzuziehen!
<i>Pack</i>	Der Packer setzt einzelne Widgets in Reihen oder Spalten des Eltern-Widgets und ist <i>Grid</i> verwandter als <i>Place</i> .

Tabelle 5.2: Übersicht der Geometrie-Manager

Klasse	Beschreibung
<i>BitImage</i>	Die Klasse wird benötigt, um vorgefertigte Bilder für Tkinter verfügbar zu machen.
<i>DoubleVar</i>	Klasse für Float-Variablen
<i>Font</i>	Wird zur Generierung von Schriftsätzen benötigt.
<i>IntVar</i>	Klasse für Integer-Variablen
<i>PhotoImage</i>	Die Klasse wird benötigt, um Fotos für Tkinter verfügbar zu machen, die für Labels, Canvases, Buttons und Text Widgets verwendet werden.
<i>StringVar</i>	Klasse für String-Variablen

Tabelle 5.3: Übersicht der Tkinter-Klassen, die keine Widgets darstellen

Modul	Beschreibung
<i>ScrolledText</i>	Text-Widget mit einer vertikalen, eingebauten Scrollbar (siehe <i>Tk_intern</i> ).
<i>tkColorChooser</i>	Dialog, um eine Farbe zu wählen (siehe Kapitel 10).
<i>tkCommonDialog</i>	Basisklasse für alle Dialoge in den Dialogmodulen (siehe Kapitel 10).
<i>tkFileDialog</i>	Dialoge für das Öffnen und Speichern von Dateien (siehe Kapitel 10).
<i>tkFont</i>	Utilities für das Arbeiten mit Fonts.
<i>tkMessageBox</i>	Schnittstelle zu den Standard Message-Boxen von Tk (siehe Kapitel 10).
<i>tkSimpleDialog</i>	Schnittstelle zu Standard-Eingabeboxen von Tk (siehe Kapitel 10).
<i>Tkdnd</i>	Drag and Drop Unterstützung für Tkinter (noch experimentell, siehe <i>Tk_intern</i> ).
<i>turtle</i>	Turtle-Grafik (siehe <i>Tk_intern</i> ).

Tabelle 5.4: Übersicht zusätzlicher Module für Tkinter

## Kapitel 6

# Tk\_Intern: *Canvas*

Wir hatten in den vorangegangenen Beispielen echte Widgets untersucht. Um den Gedankengang, den wir mit der Turtle begannen, weiterzuführen, wollen wir uns aber als erster Widgetklasse mit dem Canvas-Widget genauer befassen und die Elemente nach und nach einbauen. Führen Sie die einzelnen Schritte am besten gleich am Interpreterprompt aus.

Wir erinnern uns daran, dass *canvas* ein Zeichenbrett bedeutet. Richten wir also eins ein<sup>1</sup>:

```
canvas=Canvas(width=640, height=480, bg='white')
```

erstellt ein solches Zeichenbrett mit den Abmessungen 640x480 und weissem Hintergrund. Es erscheint allerdings noch *nicht* das Canvas, wenn Sie diese Zeile eingeben, sondern das Hauptwidget, welches das Canvas aufnehmen soll. Sie werden zugeben müssen, dass der Hintergrund in der Tat grau und nicht weiss aussieht. Mit dem Schritt

```
canvas.pack(expand=YES, fill=BOTH)
```

erscheint das Canvas tatsächlich und füllt das ganze Fenster nicht nur in beiden Richtungen aus, sondern erweitert es auch so, dass es tatsächlich seine richtige Grösse hat, obwohl es noch gar nichts enthält. Verwechseln Sie bitte nicht *canvas* mit *Canvas*. Das ist ein wesentlicher Unterschied:

---

<sup>1</sup> *import tkinter* oder *from tkinter import \** wird jetzt nicht immer mitgeschrieben!

- *Canvas* ist der nicht frei wählbare der Klasse der Canvas-Widgets, so wie sie in Tkinter implementiert ist. Der Aufruf *Canvas(optionen...)* ist bereits eine Methode der Klasse und zwar der *Konstruktor*<sup>2</sup>, der eine *Instanz*, d.h. ein *Objekt* der Klasse mit allen Attributen und Methoden generiert.
- *canvas* ist der von uns frei gewählte Name dieser Instanz, die nun in dem Sinne ein Abbild der Klasse *Canvas* ist, wie das Ergebnis eines Stempeldrucks das Abbild des Stempels darstellt.

In den folgenden Abschnitten werden wir uns nun die einzelnen Methoden einer Canvas-Instanz ansehen. Die nun folgenden Objekte müssen sich an eine Eigenheit halten, die bei der Turtle anders war: die Koordinaten für alle Zeichenobjekte laufen von dem linken oberen Eckpunkt (0|0) zu rechten unteren (639|479). Der Mittelpunkt dieses Canvas lautet demnach (320|240) und nicht (0|0) wie bei der Turtle.

## 6.1 Zeichnen von Ellipsen und Kreisen

Versuchen wir also jetzt einen gelben Kreis mit Radius 50 und rotem Rand genau in die Mitte des Canvas zu plazieren:

```
kreis=canvas_create.oval(320-25,240-25,320+25,240+25,
width=2, outline='red', fill='yellow') #muss in eine Zeile!
```

Natürlich hätten wir auch statt 320−25 einfach 299 schreiben können, aber es soll hier deutlich werden, wie ein Kreis mit vorgegebenem Radius so gesetzt werden kann, dass sein Mittelpunkt genau an der Position steht, wo man ihn haben will.

Wir wollen nun eine aufrechte Ellipse zeichnen, deren grosse Halbachse genauso gross wie die des Kreises und deren kleine Halbachse genau halb so gross sein soll. Die Position sei der Punkt (100,100).

```
ellipse=canvas_create.oval(100-12.5,240-25,320+12.5,240+25,
width=2, fill='green') #muss in einer Zeile stehen
```

---

<sup>2</sup>Innerhalb der Klassendefinition ist dieser Konstruktor durch `__init__` vorgegeben.



## 6.2 Arbeiten mit den gezeichneten Objekten

Wir haben beiden Objekten Namen gegeben und wollen nun einmal sehen, welcher *Art* diese Objekte sind. Dazu drucken wir beiden Namen aus:

```
print kreis, ellipse
```

Das Ergebnis ist etwas ernüchternd. Es werden nur die Nummern 1 und 2 ausgedruckt. Tatsächlich sind die gezeichneten Figuren, wie wir auch schon bei der Turtle feststellten, *keine* Widgets, mit denen man irgendetwas machen kann. Auch wenn es in den folgenden Schritten so aussieht. Solche Objekte nennen wir *Items*. Alle Items werden je nach dem Zeitpunkt der Erstellung mit 1 beginnend durchnummeriert, so dass statt durch einen Namen wie hier *kreis* und *ellipse* die Items auch durch die Nummer angesprochen werden können.

### 6.2.1 Konfigurieren eines Items

Wir wollen die Ellipse nun genauso einfärben wie den Kreis. Dazu bedienen wir uns der Methode *itemconfig*.

```
canvas.itemconfig(ellipse, outline='red', fill='yellow')
```

Wollen wir dem Canvas selbst einen anderen Hintergrund, zum Beispiel 'grau' geben, so schreiben wir:

```
canvas.config(bg='gray')
```

### 6.2.2 Löschen eines Items

Wir lassen nun die Ellipse verschwinden, indem wir

```
canvas.delete(ellipse)
```

eingeben. Soll auch noch der Kreis verschwinden, geben wir Entsprechendes ein, können aber natürlich auch

```
canvas.delete(1)
```

absetzen, da der Name *kreis* ja auf die 1 zeigt. Tatsächlich sind beide Items zerstört - genauer gesagt:

- sie wurden mit der Hintergrundfarbe übermalt und
- es wurden alle Informationen über sie gelöscht

Es gibt keine Methode von *canvas*, die so etwas rückgängig macht. Zwar zeigen die Namen *kreis* und *ellipse* noch auf die Nummern, aber diese sind keinem Item mehr zugeordnet. Eine generelle *create*-Methode der Art *create(ellipse)* gibt es nicht. Es muss das Item neu erstellt werden. Tun Sie dies bitte jetzt wieder so wie vorhin. Wichtig ist, dass die Ellipse jetzt wieder grün aussieht!

### 6.2.3 Bewegen und Verändern eines Items mit Hilfe von Tags

Mit

```
canvas.move(ellipse,320-100,240-100)
```

bewegen wir die Ellipse in den Kreis hinein. Für die Bewegung werden als Koordinaten keine absoluten, sondern immer relative Koordinaten verwendet.

Was nun, wenn jetzt beide Items *gemeinsam* an die alte Position der Ellipse gesetzt werden sollen? Natürlich wären zwei entsprechende Aufrufe nacheinander für *kreis* und *ellipse* die Lösung. Aber es geht auch sehr viel eleganter: mit Hilfe von *Tags*.

Ein *Tag* ist eine Marke, die an ein Item gebunden wird, gleich einem Fähnchen. Die eigentlichen Namen der Items sind ja, wie wir sahen, Nummern, auch *ObjectIDs* genannt. Die Fähnchen haben die Gestalt von Strings, und können entweder gleich bei der Einrichtung mit der *tag=* Option im Konstruktor oder nachträglich mit der Methode *addtag\_withtag(string,objectId)* eingerichtet werden:

```
canvas.addtag_withtag('egg',kreis)
canvas.addtag_withtag('egg',ellipse)
```

Statt einer *objectId* kann als zweiter Parameter auch ein Tag angegeben werden. Dann werden alle Items mit demselben Tag durch das neue Tag markiert. Tags dienen also dazu, Items zusammenzufassen.

Ein Aufruf der Form

```
canvas.move('egg',-240,-140)
```

bewegt jetzt also *beide* items wieder an den alten Platz der Ellipse zurück.

### 6.2.4 Überdecken und Verstecken von Items

Zeichnen wir nun einen neuen Kreis

```
gkreis=canvas_create.oval(100-30,100-30,100+30,100+30,
width=1, outline='red', fill='maroon', tag='gegg')
```

Dieser überdeckt die alten Objekte. Wollen wir, dass *nur* die Ellipse oben liegt, schreiben wir

```
canvas.tkraise(ellipse)
```

Wollen wir sie wieder verschwinden lassen, schreiben wir

```
canvas.lower(ellipse)
```

Der Grund dafür, dass das Überdecken nicht mit *raise*, sondern *tkraise* erfolgen muss, hängt damit zusammen, dass *raise* ein Schlüsselwort für Standard-Python darstellt<sup>3</sup>. Wollen wir beide Objekte hervorholen, geschieht dies mit Hilfe des Tags in der Form

```
canvas.tkraise('egg')
```

---

<sup>3</sup>...und zwar bei dem Abfangen von Ausnahmen

### 6.2.5 Das *all* und das *current*-Tag

Geben Sie einmal ein

```
canvas.move('all',100,100)
```

Das Tag *'all'* ist ein reserviertes Tag, dass *alle* Items tragen. Lassen Sie jetzt den Mauszeiger genau in der grünen Ellipse stehen und geben Sie ein

```
canvas.move('current',100,100)
```

Das Ergebnis ist verblüffend. In der Tat ist das Bewegen mit Hilfe des *current*-Tags der erste Schritt zur Maussteuerung von Items. Nur dasjenige Item, das sich direkt unter dem Mauscursor befindet, trägt das *'current'*-Tag. Das *'all'*-Tag ist permanent, das *'current'*-Tag dagegen ist temporär.

## 6.3 Weitere Canvas-Items

Auf dieselbe Art und Weise können Sie mit den noch fehlenden Items vorgehen. Grundsätzlich haben alle die folgenden Methoden den Aufruf

```
Canvasinstanz.methode(fromX,fromY,toX,toY,optionen(optional))
```

Methoden, die nach diesem Prinzip arbeiten sind:

- *create\_rectangle*, für das Zeichnen eines Rechtecks
- *create\_line*, wobei hier darauf zu achten ist, dass die Option *outline* nicht funktioniert, sondern mit *fill* gearbeitet werden muss
- *create\_arc*, wobei hier ein Kreisbogen vom Anfangspunkt bis zum Endpunkt in der Weise gezeichnet wird, dass diese einander gegenüberliegende Ecken eines Rechteks sind. Es handelt sich also *nicht* um Kreisbögen, sondern um ein Viertel von Ellipsenbögen.
- *create\_polygon*, wobei hier statt »fromX,fromY,toX,toY« mindestens noch zwei weitere Koordinaten für einen dritten Punkt angegeben werden müssen. Es sind beliebig viele Punkte möglich, aber mindestens drei *müssen* es sein.

Canvases, die irgendetwas setzen und in der Form

```
Canvasinstanz.methode(X,Y, optionen)
```

aufgerufen werden, sind

- *create\_image*, zum Einbringen eines Photos,
- *create\_window*, zum Einsetzen eines Widgets beliebiger Art und
- *create\_text*, zum Einbringen von Text.

Wenn ein Photo in das Canvas eingebracht werden soll, erzeugen Sie sich zuerst eine Instanz der Klasse *PhotoImage* in der Form

```
photo=PhotoImage(file='filename')
```

wobei mit *filename* der vollständige Pfad als Name und als String angegeben werden muss. Das Setzen des Photos erfolgt nun mit der Methode *create\_image* in der Form

```
Canvasinstanz.create_image(X,Y, image=photo, anchor=NW)
```

Die Option *image* ist natürlich notwendig, um die eben erzeugte Photoinstanz an die Methode zu binden. Der *anchor* ist eine Angabe in »Himmelsrichtungen«. Der Wert NW sagt, dass die obere linke Ecke (Nordwesten) an die Position (*X*, *Y*) gesetzt werden soll.

Die anderen beiden Canvases arbeiten genauso. Nur, statt *image* wird bei *create\_window* die Option *window* benötigt und bei *create\_text* die Option *text*=*'irgendwas'*, wobei »irgendetwas« natürlich der Text sein soll, der dargestellt wird.

Statt diese Items nun noch einzeln durchzugehen und festzustellen, dass wir ohnehin nicht viel Neues dabei lernen, hier ein Beispielprogramm:

```

#file: canvas1.py
from Tkinter import *
canvas=Canvas(width=400, height=400, bg='white')
canvas.pack()
item_a=canvas.create_line(200,200,300,300, width=3, fill='blue')
print item_a
for i in range (50,100,2):
    canvas.create_line(10,i,50,i)
item_b=canvas.create_oval(50,50,200,200, width=2, fill='red')
print item_b
item_c=canvas.create_oval(70,170,150,250, width=2, fill='yellow')
print item_c
canvas.create_rectangle(300,300,350,350, width=4, outline='magenta',
fill='blue') # in eine Zeile!
canvas.create_arc(10,300,70,360, fill='green')
photo=PhotoImage(file='guido.gif')
canvas.create_image(200,0, image=photo, anchor=NW)
widget=Label(canvas,text='I am a lumberjack', fg='white', bg='black')
widget.pack()
canvas.create_window(120,100, window=widget)
canvas.create_text(300,50, text='I am OK')
canvas.create_polygon(200,200,300,200,300,250, fill='blue',
stipple='gray25') # in eine Zeile!

```

Das Ergebnis dieses Programms zeigt die Abbildung 6.1 . Mit der Option *stipple* können Patterns eingespielt werden. Dazu später mehr.

## 6.4 Canvas und Turtle

Im Abschnitt 1.5, in dem wir Einiges über Turtle-Klassen gesagt haben, wiesen wir daraufhin, dass es eine Klasse mit dem Namen *RawPen* gibt, die es uns erlaubt, in regulären Canvases eine Turtle einzubauen. Wie das geht, zeigt das folgende Beispiel:

```

file: canvas2.py
from Tkinter import *
from turtle import *
canvas=Canvas(width=320, height=240, bg='gray')

```

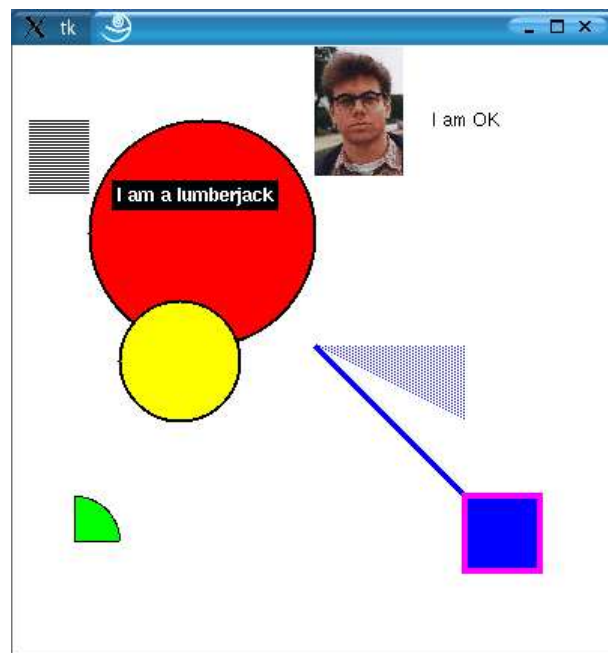


Abbildung 6.1: Verschiedene Canvases

```

canvas.pack(expand=YES)
widget=Canvas(canvas, width=200, height=200, bg='white')
widget.pack(padx=200,pady=40)
def neck(canvasname, eckenzahl, laenge, breite, col, tpen):
    winkel=360/eckenzahl
    tpen.width(breite)
    tpen.color(col)
    for i in range (eckenzahl):
        tpen.forward (laenge)
        tpen.left(winkel)
def rosette(canvasname, eckenzahl, laenge, breite, col, drehwinkel,tpen):
    for i in range(360/drehwinkel):
        neck(canvasname, eckenzahl, laenge, breite, col, tpen)
        tpen.left(drehwinkel)

turtlepen=RawPen(canvas)
turtlepen.up()
turtlepen.goto(-80,-40)
turtlepen.down()
turtlepen.tracer(0)
rosette(canvas,10,20,1,'red',36,turtlepen)
turtle=RawPen(widget)
turtle.up()
turtle.down()
turtle.tracer(0)
rosette(widget,10,20,1,'blue',36,turtle)

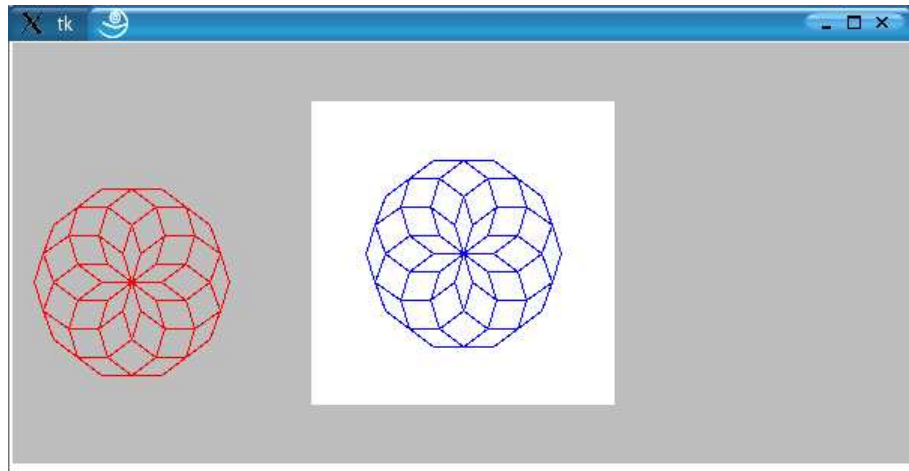
```

Es ist möglich, mit einem RawPen genauso umzugehen wie mit einer ganz gewöhnlichen Turtle der Klasse Pen(), nur dass die Turtle-Prozeduren jetzt als Methoden in Erscheinung treten und bei der Erstellung dasjenige Canvas angegeben werden muss, in dem die Turtle arbeiten soll. Das Ergebnis zeigt die Abbildung 6.2.

## 6.5 Canvas und Scrollbars

Um eine Scrollbar in Y-Richtung anzudocken, müssen die folgenden Schritte durchgeführt werden:



Abbildung 6.2: Turtle vom Typ *RawPen()* in Aktion

1. Es muss im Canvas mit der Option *scrollregion* der scrollbare Bereich des Canvas festgelegt werden. Der Option *scrollregion* wird dabei die linke obere Ecke des Canvas und die rechte untere Ecke als 4-Tupel angegeben, also *scrollregion=(fromX, fromY, toX, toY)*.
2. Mit dem Konstruktor *Scrollbar()* wird eine Scrollbar erzeugt.
3. Das Canvas hat ein Attribut mit dem Namen *yview*. Dieses muss der Scrollbar als *command*-Option übergeben werden, um die Scrollbar mit dem Canvas zu verbinden. Der Befehl lautet also: *scrollbarInstanz.config(command=canvasInstanz.yview)*
4. Alle Befehle, die von der Scrollbar kommen, müssen natürlich vom Canvas ausgewertet werden. Die Scrollbar weiss durch den vorhergehenden Schritt, an welches Canvas sie die Befehle absetzen soll, aber das Canvas weiss natürlich noch nicht, von welcher Scrollbar es die Befehle auswerten soll. Das Attribut *set* der Scrollbarinstanz muss dem Canvas als Option *yscrollcommand* übergeben werden und zwar in der Form *canvasInstanz.config(yscrollcommand=scrollbarInstanz.set)*.
5. Zum Schluss müssen sowohl die Scrollbar als auch das Canvas gepackt werden.

Ein Beispiel:



Abbildung 6.3: Canvas mit Scrollbars

```
#file: canvas3.py
from Tkinter import *
canv=Canvas(width=300, height=200, bg='blue', relief=SUNKEN)
#Gibt an, von wo bis wo der Scrollbalken laufen soll
canv.config(scrollregion=(0,0,300,1000))
canv.config(highlightthickness=0)
sbary=Scrollbar()
sbary.config(command=canv.yview)
canv.config(yscrollcommand=sbary.set)
sbary.pack(side=RIGHT, fill=Y)
canv.pack(side=LEFT, expand=YES, fill=BOTH)
canv.create_text(150,150,text='I am a lumberjack ...', fill='beige')
widget=Label(canv, text='and I am OK', fg='beige', bg='brown')
widget.pack()
canv.create_window(150,250, window=widget)
```

Das Ergebnis ist in dem Bild 6.3 zu sehen. Wie Scrollbars in X-Richtung eingerichtet werden können, wird an dieser Stelle noch nicht beschrieben.

## 6.6 Canvas und Events

Über Events werden wir erst im Kapitel 8 detaillierter sprechen. Da das Problem, wie ein Item mit der Maus bewegt werden kann zu den interessantesten Fragen gehört, gebe ich hier eines der schönsten Beispiele aus dem Buch von

Mark LUTZ, Programming Python, O'Reilly Verlag, das ich nur wärmstens als erweiterte Lektüre empfehlen kann:

```
#file: canvas5.py
#Beispiel von Mark Lutz aus dem Buch Programming Python, O' Reilly

from Tkinter import *

trace = 0

class CanvasEventsDemo:

    def __init__(self, parent=None):
        canvas = Canvas(width=300, height=300, bg='beige')
        canvas.pack()
        canvas.bind('<ButtonPress-1>', self.onStart) # click
        canvas.bind('<B1-Motion>', self.onGrow) # and drag
        canvas.bind('<Double-1>', self.onClear) # delete all
        canvas.bind('<ButtonPress-3>', self.onMove) # move latest
        self.canvas = canvas
        self.drawn = None
        self.kinds = [canvas.create_oval, canvas.create_rectangle]

    def onStart(self, event):
        self.shape = self.kinds[0]
        self.kinds = self.kinds[1:] + self.kinds[:1] # start dragout
        self.start = event
        self.drawn = None

    def onGrow(self, event): # delete and redraw
        canvas = event.widget
        if self.drawn: canvas.delete(self.drawn)
        objectId = self.shape(self.start.x, self.start.y, event.x, event.y)
        if trace: print objectId
        self.drawn = objectId

    def onClear(self, event):
        event.widget.delete('all') # use tag all

    def onMove(self, event):
```

```
        if self.drawn: # move to click spot
        if trace: print self.drawn
        canvas = event.widget
        diffX, diffY = (event.x - self.start.x), (event.y - self.start.y)
        canvas.move(self.drawn, diffX, diffY)
        self.start = event

CanvasEventsDemo()
mainloop()
```

Dieses Beispiel kann Ellipsen und Rechtecke zeichnen und zwar wie folgt:

- Durch Ziehen der Maus bei niedergedrückter linker Taste (Button-1) wird auf dem Canvas eine der beiden Figuren gezeichnet, und zwar so, dass Sie mit der Maus die Figur vergrössern können.
- Wenn Sie zwischen Ellipse und Rechteck wechseln möchten, klicken Sie mit der mittleren Maustaste einmal ins Canvas.
- Durch Anklicken eines Ortes mit der rechten Maustaste positionieren Sie das letzte gezeichnete Item. Wichtig ist, dass Sie dabei nicht irgendwo vorher mit der linken Maustaste einen Ort angeklickt haben, denn dann ist dieser Ort als Punkt das zuletzt gezeichnete Item.

## Kapitel 7

# Allgemeine Konfiguration von Widgets

Um das Aussehen eines Widgets zu kontrollieren, verändert man in der Regel die *Optionen* des Widgets.

Um ein Widget zu erstellen, wird der Konstruktor der Widgetklasse in der Form

```
widgetklassenname(master,option=wert,)
```

aufgerufen. Wenn zum Beispiel ein Button erstellt werden soll, geschieht dies durch einen Aufruf der Form

```
b=Button(root,text='Quit', command=root.quit)
```

Das Masterfenster, das hier *root* heisst, muss natürlich vorher eingerichtet worden sein. Alle Optionen eines Widgets haben vordefinierte Werte, so dass Sie im einfachsten Fall nur den Namen des Masterfensters übergeben müssen. Selbst das ist aber auch nicht wirklich notwendig. Wenn der Master fehlt, nimmt Tkinter das zuletzt eingerichtete Window als Master.

Um den gerade eingestellten Wert einer Option wie zum Beispiel *text* zu erfahren, benutzen Sie die Methode *cget* der Instanz. Zum Beispiel ergibt

```
>>>b.cget('text')
```

für die gerade eingerichtete Button-Instanz den Wert 'Quit'. Achten Sie bitte darauf, dass Sie auch die Option als einen String eingeben. Der Rückgabewert ist ebenfalls ein String.

Eine andere Möglichkeit, den aktuellen Wert von Optionen abzufragen ist *widgetname[option (als String!)]*. So liefert zum Beispiel

```
>>>b['fg']
```

die aktuell eingestellte Farbe.

Um bei eingerichteter Instanz später noch Optionen zu verändern, verwenden Sie die Methode *configure* oder *config*. Die Optionen müssen dabei als Schlüsselworte angegeben werden. Zum Beispiel:

```
>>>b.config(fg='red')
```

färbt die Schrift auf dem Button fortan rot.

Alternativ können natürlich auch durch *widgetname[option (als String!)]=wert (als String!)* Optionen gesetzt werden, also zum Beispiel:

```
>>>b['fg']='blue'
```

Sämtliche Optionen eines Widgets werden mit der (Dictionary-)Methode *keys* als Liste zurückgegeben. Probieren Sie aus:

```
>>>b.keys()
```

Alle Standardwidgets von Tkinter besitzen einen ähnlichen Satz von Optionen, über welche die Farbe, die Schriftart oder Ähnliches festgelegt werden kann.

## 7.1 Farbe

Mit den Optionen *background* und *foreground* können die Widgetfarbe und die Textfarbe festgelegt werden. Als Farbnamen dürfen vordefinierte Wörter verwendet werden, oder Sie legen die Farbe über einen RGB-Code fest. In

jedem Falle sind die Farbnamen *Red*, *Green*, *Blue*, *Yellow* und *LightBlue* verfügbar.

Um einen RGB-Code für eine *beliebige* Farbe festzulegen, müssen Sie die Sättigungswerte für Rot, Grün und Blau als Folge von drei zweistelligen Hexadezimalzahlen in der Form *#RRGGBB* eingeben. Da  $16 \cdot 16$  die Zahl 256 ergibt dürfen diese Sättigungswerte also von 0 bis 255 laufen.

Beispiel: Wollen wir für Rot eine Sättigung vom Wert 128 haben, was im Hexagesimalsystem den Wert 80 ergibt, für Grün eine Sättigung von 192, was im Hexagesimalsystem den Wert c0 ergibt, und für Blau die Sättigung 200, was im Hexagesimalsystem c8 bedeutet, so wandelt

```
>>>b.config(fg='#80c0c8')
```

die Buttonfarbe entsprechend in das Ergebnis um, was ein schwaches Türkis ergibt. Wenn Ihnen die Rechnerei zu lästig erscheint, können Sie auch das Tupel (128, 192, 200) in der folgenden Form in einen String verwandeln:

```
>>>rgb='#%02x%02x%02x' % (128,192,200)
```

Die Bezeichnung *%02* ist ein Platzhalter, der den zugeordneten Wert in eine Hexadezimalzahl verwandelt.

Umgekehrt können Sie mit der Methode *winfo\_rgb* einen solchen String oder ein Farbwort wieder in ein Tupel zurückverwandeln. Allerdings erfolgt die Angabe durch 16-Bit-Ketten, die demnach von 0 bis 65535. Sie müssen also alle Werte des Tupels noch nachträglich durch 256 teilen.

Probieren Sie nun bitte aus:

```
>>>rgbtupel=b.winfo_rgb('#80c0c8')
>>>rgbtupel[0]/256, rgbtupel[1]/256, rgbtupel[2]/256
```

und

```
>>>rgbtupel=b.winfo_rgb('red')
>>>rgbtupel[0]/256, rgbtupel[1]/256, rgbtupel[2]/256
```

## 7.2 Schriftsatz

Um Schriftsätze (*fonts*) festzulegen, übergeben Sie an die *text*-Option eines Widgets entweder

- ein Tripel der Form *(‘Helvetica’,10,’bold’)* oder, falls der Fontname keine Leerzeichen enthält
- einen dreistelligen String *‘Helvetica 10 bold italic’*

Zuerst kommt immer der Name der Schriftfamilie, zum Beispiel: Arial, Helvetica, Times New Roman (hier *muss* die Tupelschreibweise verwendet werden!). Danach wird die Grösse<sup>1</sup> angegeben und zum Schluss optionale Stilbezeichnungen wie *bold* für ‘Fettdruck’ oder *bold italic* für kursiven Fettdruck. Probieren Sie aus:

```
>>>b.config(font='Arial 10 bold italic')
>>>b.config(font=('Times New Roman', 12, 'bold'))
```

Stilbezeichnungen sind: bold, italic, normal, overstrike, roman, underline.

### 7.2.1 Eigene Fonts als aufrufbare Instanzen

Tkinter ermöglicht die Erstellung eigener Fonts wenn das Modul *tkFont* geladen wurde. In diesem Modul existiert die Klasse *Font*, deren Konstruktor einen neuen Font als Instanz festlegt. Der Konstruktor enthält die Optionen:

**family:** Die Schriftsatzfamilie (String), zum Beispiel ‘Arial’

**size:** Die Grösse des Fonts als Integerzahl.

**weight:** Dicke der Schrift, die entweder die Konstante NORMAL (Voreinstellung) oder die Konstante BOLD annehmen kann.

**slant:** Schrägheit des Fonts, der die Konstante NORMAL (Voreinstellung) oder die Konstante ITALIC annehmen kann.

---

<sup>1</sup>In der Tupelschreibweise nicht als String!



**underline:** Hierbei handelt es sich um ein Bit<sup>2</sup>, das gesetzt werden kann (Wert 1) oder auch nicht (Wert 0, Voreinstellung).

**overline:** ist das Entsprechende zu *underline*.

Ein Beispiel:

```
>>>from tkFont import *
>>>myfont=Font(family='Arial',size=10,weight=BOLD,underline=1)
>>>b.config(font=myfont)
```

### 7.3 Textformatierung

Obwohl Labels und Buttons normalerweise nur eine einzige Zeile Text enthalten, unterstützt Tkinter auch mehrere Zeilen. Wie bei einem gewöhnlichen String wird durch das Newline-Zeichen `\n` ein Zeilenumbruch festgelegt.

Eine andere Möglichkeit, Text umzubrechen, ist die Festlegung der *wrapline*-Option auf eine bestimmte Anzahl von Zeichen. Überschreitet die Zeilenlänge diese Grenze, wird automatisch umgebrochen.

Soll der Text nicht zentriert erscheinen, muss die *justify*-Option auf LEFT oder RIGHT gesetzt werden.

### 7.4 Rahmen

Alle Widgets sind durch *Borders* begrenzt, auch wenn diese nicht immer sichtbar sind. Die folgenden Optionen sind für das Zeichnen eines Rahmens möglich:

**borderwidth:** Diese Option kann auch durch *bd* abgekürzt werden. Sie setzt die Bordergrenzen in Pixeln, wobei ein oder zwei Pixel das Normale sind.

**relief:** gibt an, wie ein 3D-Border gezeichnet werden soll und kann die Werte SUNKEN, RIDGE und FLAT annehmen.

---

<sup>2</sup>Ein sogenanntes *Flag*.

Schliesslich gibt es noch drei Optionen, die steuern wie der Rand aussehen soll, wenn ein Fenster für Eingaben über die Tastatur aktiv oder passiv ist. Ein aktives Fenster wurde mit der Maus angeklickt und ermöglicht nun, irgendwelche Texte einzugeben. Alle anderen Fenster sind dann *passiv*.

**highlightcolor:** gibt die Farbe des Randes an, wenn das Fenster *aktiv* ist.

**highlightbackground:** gibt die Randfarbe an, wenn das Fenster *passiv* ist.

**highlightthickness:** gibt die Breite der Border an, wenn das Fenster aktiv ist.

## 7.5 Aussehen des Cursors

Durch die Option *cursor* kann in einem Widget das Aussehen des Maus-cursors verändert werden, wenn die Maus sich in dem Widget befindet. Als Objekte für diese Option stehen eine Unmenge an Bildern zur Verfügung, die am Ende dieses Abschnitts namentlich aufgeführt sind.

Probieren Sie das Verändern des Cursors durch die Zeile

```
>>>b.config(cursor=gobby)
```

einmal aus.

Solange sich der Cursor nicht direkt über dem Button befindet, behält er seine alte Gestalt.

Wenn die *cursor*-Option nicht angegeben wird, so benutzt Tkinter den Cursor des übergeordneten Fensters.

Und hier ist die Liste:

**A:** arrow

**B:** based\_arrow\_down, based\_arrow\_up, boat, bogosity, bottom\_left\_corner, bottom\_right\_corner, bottom\_side, bottom\_tee, box\_spiral

**C:** center\_ptr, circle, clock, coffee\_mug, cross, cross\_reverse, crosshair

**D:** diamond\_cross, dot, dotbox, double\_arrow, draft\_large, draft\_small, draped\_box

**E:** exchange

**F:** fleur

**G:** gobbler, gumby

**H:** hand1, hand2, heart

**I:** icon, iron\_cross

**L:** left\_ptr, left\_side, left\_tee, leftbutton, ll\_angle, lr\_angle

**M:** man, middlebutton, mouse

**P:** pencil, pirate, plus

**Q:** question\_arrow

**R:** right\_ptr, right\_side, right\_tee, rightbutton, rtl\_logo

**S:** sailboat, sb\_down\_arrow, sb\_h\_double\_arrow, sb\_left\_arrow, sb\_right\_arrow,  
sb\_up\_arrow, sb\_v\_double\_arrow, shuttle, sizing, spider, spraycan,  
star

**T:** target, tcross, top\_left\_arrow, top\_left\_corner, top\_right\_corner, top\_side,  
top\_tee, trek

**U:** ul\_angle, umbrella, ur\_angle

**W:** watch

**X:** xterm

## 7.6 Referenz zum Styling von Widgets

In diesem Abschnitt werden die Optionen zusammengestellt, die bei einem Aufruf der Form

*widgetInstanz.config()*

oder im Konstruktor der Klasse gesetzt werden können. Dabei bedeuten die Zeichen:

- ∈: Angabe der Art des Objektes, das als Wert zugewiesen werden kann
- ∃: Angabe derjenigen Widgets, die diese Option verarbeiten können; alle Widgets, die nicht aufgeführt sind, können diese Option nicht verarbeiten
- ∀: Angabe derjenigen Widgets, die diese Option *nicht* verarbeiten können; alle nicht aufgeführten Widgets können diese Option verarbeiten

### 7.6.1 Häufige Optionen, die von fast allen Widgets unterstützt werden

- background (bg):** legt die Hintergrundfarbe fest; ∈: color, z.B. '#FF5500' oder 'red'; die Angaben müssen als String erfolgen;
- borderwidth (bd):** legt die Randbreite für einen 3D-Effekt fest; ∈: pixel als Integerzahl, z.B. 3
- cursor:** legt die Art des Mauscursors fest, wenn er sich in dem Widget befindet; ∈: cursor, z.B. gumby
- font:** legt den Schriftsatz fest; ∈: font, z. B. 'helvetica' oder 'Arial, 8';  
∀: Canvas, Frame, Scrollbars und Toplevel
- foreground (fg):** legt die Vordergrundfarbe fest; ∈: wie *background*; ∀: Canvas, Frame, Scrollbars und Toplevel
- highlightbackground:** legt die Farbe fest, wenn sich das Fenster im Hintergrund befindet, d.h. der Mauscursor zur Eingabe befindet sich in einem anderen Fenster; ∈: wie *background*; ∀: Menu
- highlightcolor:** wie *highlightbackground* für das Fenster, das gerade für eine Eingabe aktiv ist
- highlightthickness:** legt die Dicke der Umrandung für das aktive Fenster fest; ∈: wie *borderwidth*; ∀: Menu
- relief:** legt den 3D-Effekt für ein Widget fest; ∈: Konstanten: RAISED, SUNKEN, FLAT, RIDGE, SOLID, GROOVE
- takefocus:** legt fest, inwieweit ein Fenster den Eingabefokus bekommen soll; ∈: boolean, konstante Werte 0 (Fenster ist passiv), 1 (Fenster bekommt

den Eingabefokus so lange er über dem Widget sichtbar ist. Wird *takefocus* nicht gesetzt, wird automatisch entschieden, wann das Fenster aktiv oder passiv ist.

**width:** legt die Breite eines Widgets fest;  $\in$ : integer, zum Beispiel 3;  $\forall$ : Menu

### 7.6.2 Seltenere Optionen, die nicht von allen Widgets unterstützt werden

**activebackground:** legt die Hintergrundfarbe fest, wenn die Maus aktiv mit dem Fenster arbeitet, z.B. es an einen anderen Ort zieht;  $\in$ : color, 'red', '#FA5090';  $\exists$ : Button, Checkbutton, Menu, Menubutton, Radiobutton, Scale, Scrollbar

**activeforeground:** Analogon zu *activebackground*

**anchor:** legt nach Himmelsrichtungen fest, wie eine Information in einem Widget angegeben werden soll;  $\in$ : Konstanten: N, NE, E, SE, S, SW, W, NW, CENTER;  $\exists$ : Button, Checkbutton, Label, Menubutton, Message, Radiobutton

**bitmap:** legt eine Bitmap-Zeichnung fest, die in einem Widget erscheinen soll;  $\in$ : Bitmap;  $\exists$ : Button, Checkbutton, Label, Menubutton, Radiobutton

**command:** bindet ein Kommando an das Widget;  $\in$ : command, z.B. on-Click, wenn diese Methode oder Funktion definiert wurde;  $\exists$ : Button, Checkbutton, Radiobutton, Scale, Scrollbar

**disabledforeground:** Analogon zu *activeforeground*

**height:** legt die Höhe eines Widgets fest;  $\in$ : integer;  $\exists$ : Button, Canvas, Checkbutton, Frame, Label, Listbox, Menubutton, Radiobutton, Text, Toplevel

**image:** legt ein Bild fest, das auf dem Widget gezeigt werden soll; dieses muss mit der Methode *create* erzeugt worden sein;  $\in$ : image;  $\exists$ : Button, Checkbutton, Label, Menubutton, Radiobutton

**justify:** legt die Ausrichtung eines Textes aus mehreren Zeilen auf dem Widget fest;  $\in$ : Konstante: RIGHT, LEFT, CENTER;  $\exists$ : Button, Checkbutton, Entry, Label, Menubutton, Radiobutton

**padx:** legt fest, wieviel zusätzlicher Freiraum für das zu platzierende Widget in der X-Richtung erforderlich ist;  $\in$ : pixel, z.B. 2m oder 10;  $\exists$ : Button, Checkbutton, Label, Menubutton, Message, Radiobutton, Text

**pady:** Analogon zu *padx*

**selectbackground:** legt die Hintergrundfarbe für ausgewählte Items fest;  $\in$ : siehe *activebackground*;  $\exists$ : Canvas, Entry, Listbox, Text

**selectborderwidth:** legt die Strichdicke des Randes von einem ausgewählten Item fest;  $\in$ : color, siehe *activebackground*;  $\exists$ : wie *selectbackground*

**selectforeground:** Analogon zu *selectbackground*

**state:** spezifiziert einen bestimmten Status;  $\in$ : Konstante, ACTIVE, DISABLED, NORMAL;  $\exists$ : Button, Checkbutton, Entry, Menubutton, Radiobutton, Scale, Text

**text:** legt einen String zur Ausgabe fest;  $\in$ : string,  $\exists$ : Button, Checkbutton, Label, Menubutton, Message, Radiobutton

**textvariable:** legt den Namen einer Textvariablen fest  $M \in$ : variable, z.B. widgetContent;  $\exists$ : Button, Checkbutton, Entry, Label, Menubutton, Message, Radiobutton

**underline:** legt fest, welcher Buchstabe des Textes unterstrichen werden soll (für Short-Cuts);  $\in$ : integer, der angeibt, welcher Buchstabe des Strings benutzt werden soll, z.B. 2 für den dritten Buchstaben (zum Zählen wird bei 0 angefangen);  $\exists$ : Button, Checkbutton, Label, Menubutton, Radiobutton

**wraplength:** legt die Maximallänge eines Textes fest, bevor ein Umbruch erfolgt;  $\in$ : pixel, z.B. 5, 3i;  $\exists$ : Button, Checkbutton, Label, Menubutton, Radiobutton

**xscrollcommand:** legt ein Kommando fest, um mit der horizontalen Scrollbar zu kommunizieren;  $\in$ : Funktion, zum Beispiel *xscrollcommand=scrollbarInstanz.set*;  $\exists$ : Canvas, Entry, Listbox, Text

**yscrollcommand:** Analogon zu *xscrollcommand*.

## Kapitel 8

# Ereignisse (Events) und Bindungen

Wie wir bereits vorher erwähnten, verbringt eine Tkinter-Applikation die meiste Zeit innerhalb einer Ereignisschleife (event loop), die durch eine *main-loop*-Methode initiiert wird. Ereignisse können durch verschiedene Aktionen ausgelöst werden, zum Beispiel:

- Tastendrücke
- Mausklicks
- oder zurückgegebene Ereignisse des Windows-Managers

Tkinter stellt einen sehr mächtigen Mechanismus zur Verfügung, der es Ihnen erlaubt, Ereignisse selbst zu kontrollieren. Für jedes Widget kann mit der *bind*-Methode eine Python-Funktion oder eine Methode an ein Ereignis gebunden werden.

Hier ein einfaches Beispiel:

```
#File: bind1.py
from Tkinter import *
root=Tk()
def callback(event):
    print 'clicked at', event.x, event.y
frame=Frame(root, width=100, height=100)
```

```
frame.bind('<Button-1>', callback)
frame.pack()
root.mainloop()
```

Lassen Sie dieses Programm laufen und klicken Sie in das auftretende Fenster. Jedesmal wenn Sie klicken, erscheint eine Message der Form 'clicked at 44 63' in dem Konsolenfenster.

Das Ereignis, das mit der Funktion *callback* verbunden wird, heisst hier `<Button-1>`. Solche Ereignisse sind als Strings zu übergeben und müssen eine bestimmte Syntax einhalten, die im wesentlichen aus den drei Bestandteilen *modifier*, *type* und *detail* besteht. Die Syntax lautet

`<modifier-type-detail>`

Das *type*-Feld ist das Wichtigste am Ganzen, denn es bezeichnet die Art des Events, das wir binden wollen. Welche Arten von Events es gibt, sehen wir gleich.

Die Felder *modifier* und *detail* geben zusätzliche Informationen und sind in vielen Fällen optional.

Die Bindung eines solchen Events an eine Methode, die als *Handler* bezeichnet wird, geschieht über den Aufruf

`widgetname.bind(event, handler)`

In unserem Beispiel wird der String '`<Button-1>`' an den Namen *event* der Funktion *callback* übergeben. Die Attribute des Events `<Button-1>` heissen *self.x* und *self.y* und werden von der *callback*-Funktion ausgegeben.

## 8.1 Arten von Events

**<Button-1>**: Der linke Mausbutton wurde über dem Widget ausgelöst. Tkinter fragt die Position des Mauszeigers automatisch ab und Mausereignisse werden so lange an das Widget zurückgegeben wie der Mausbutton gehalten wird. Die aktuelle Position wird in die Attribute `<Button-1>.x` und `<Button-1>.y` gespeichert. Statt `<Button-1>` sind auch die Schreibweisen `<ButtonPress-1>` und `<1>` erlaubt.



- <**Button-2**>: Analoges Ereignis für die mittlere Maustaste.
- <**Button-3**>: Analoges Ereignis für die rechte Maustaste.
- <**B1-Motion**>: Die Maus wird über das Widget mit gedrücktem Button-1 bewegt. Positionen werden wieder in den Attributen *x* und *y* gespeichert. Analoge Ereignisse für die mittlere und rechte Maustaste.
- <**ButtonRelease-1**>: Button-1 wurde losgelassen. Weiteres siehe vorherige Beschreibungen.
- <**Double-Button-1**>: Der Button-1 wurde doppelt geklickt.
- <**Triple-Button-1**>: Der Button-1 wurde dreimal geklickt.
- <**Enter**>: Der Mauszeiger ist in das Widget hineingelaufen. Dieses Ereignis bedeutet *nicht*, dass die ENTER-Taste gedrückt wurde!
- <**Leave**>: Der Mauszeiger hat das Widget verlassen.
- <**Return**>: Der User hat die Entertaste gedrückt. Welche Tasten sonst noch als Ereignisse gebunden werden dürfen, zeigt der Abschnitt 8.2.
- <**Key**>: Der User hat irgendeine Taste gedrückt. Welche Taste gedrückt wurde, wird in dem *char*-Attribut des Ereignisses gespeichert.
- a:** Der User hat ein 'a' gedrückt. Fast alle druckbaren Zeichen können auf diese Weise verwendet werden. Achtung: 1 ist eine Keybindung, <1> dagegen ist eine Bindung des Mausbuttons. Um die Leertaste zu binden, ist <space> einzugeben. Um '<' zu binden, muss <less> benutzt werden.
- <**Shift-Up**>: Der User hat den Aufwärtspfeil und gleichzeitig die SHIFT-Taste gedrückt. Als Prefixe können neben SHIFT auch ALT und CONTROL verwendet werden.
- <**Configure**>: Die Abmessungen des Widgets haben sich geändert. Die neuen Abmessungen werden in den Attributen *width* und *height* gespeichert.

## 8.2 Tastenbindungen

Wie wir im vorigen Abschnitt sahen, wird durch `<Return>` die Enter-Taste gebunden, während `<Enter>` *keine* Keybinding, sondern eine Mausbindung darstellt. Welche weiteren Tastenbindungen gibt es sonst noch?

`<Cancel>`: die Pause-Taste

`<BackSpace>`: die Rückwärts-Löschtaste

`<Tab>`: der Tabulator

`<Shift_L>`: irgendeine Shift-Taste

`<Control_L>`: irgendeine Control-Taste

`<Alt_L>`: irgendeine ALT-Taste

`<Prior>`: Seite aufwärts

`<Next>`: Seite abwärts

und ferner die Tasten `<Pause>`, `<Caps_Lock>`, `<Escape>`, `<End>`, `<Home>`, `<Left>`, `<Up>`, `<Right>`, `<Down>`, `<Print>`, `<Insert>`, `<Delete>`, `<F1>`, ..., `<F12>`, `<Num_Lock>` und `<Scroll_Lock>`.

## 8.3 Das Event-Objekt und seine Attribute

Das Standard-Python Objekt *event* hat die folgenden Attribute:

**widget:** enthält die Information über das Widget, das dieses Ereignis erzeugte. Die Information ist *kein* Name, sondern eine Tkinter widget-Instanz! Alle Events setzen dieses Attribut.

**x, y:** Die aktuelle Mausposition in Pixeln.

**x\_root, y\_root:** Die aktuelle Mausposition relativ zur oberen linken Ecke des Bildschirms in Pixeln.

**char:** Falls es sich um ein Keyboard-Event handelt, wird der Key als String in diesem Attribut gespeichert.

**keysym:** Das Symbol des Keys.

**keycode:** Der Code des Keys.

**num:** Die Nummer des Mausbuttons.

**width, height:** Die neuen Abmessungen des Widgets in Pixeln (nur <Configure>-Events).

**type:** Die Art des Events.

## 8.4 Instanzen- und Klassenbindungen

Die *bind*-Methode, die wir oben benutzten, erzeugte eine Instanzenbindung. Das bedeutet: die Bindung wird nur auf ein einziges Widget angewendet. Wenn Sie neue Frames erzeugen, werden die Bindungen *nicht* mitvererbt. Doch es gibt eine Möglichkeit, Bindungen auf vier verschiedenen Levels vorzunehmen:

1. Bindungen an die *widget*-Instanz mit Hilfe der Methode *bind*.
2. Bindungen an das übergeordnete Fenster, das das widget enthält (Toplevel oder root). Hierzu dient ebenfalls *bind*.
3. Bindungen an die gesamte Widget-Klasse mit Hilfe von *bind\_class* und schliesslich
4. die Bindung an die gesamte Anwendung mit Hilfe von *bind\_all*.

Denken Sie zum Beispiel einmal an Programme, bei denen über die **F1**-Taste ein Hilfefenster geöffnet wird. In diesen Programmen ist mit *bind\_all* die **F1**-Taste an die *gesamte* Applikation gebunden worden.

Wenn Sie mehrfache Bindungen derselben Taste vornehmen wollen, so bedenken Sie, dass Tkinter immer die naheliegendste Bindung dann auch tatsächlich ausführt. Eine Bindung der **Enter**-Taste durch <Return> und <Key> führt dazu, dass die erste Bindung aufgerufen wird, sobald die **Enter**-Taste gedrückt wurde.

Wenn eine <Return>-Bindung sowohl an ein Toplevelwidget vorgenommen wird und eine <Return>-Bindung auf der Instanzebene, werden beide Bindungen aufgerufen und zwar genau in der oben angegebenen Reihenfolge.

Manchmal gibt es dabei Verwirrungen, wenn Sie versuchen, das Standardverhalten eines Widgets abzuändern. Denken Sie zum Beispiel an Texteingabefenster. Sie möchten verhindern, dass User in ein Textfenster mehr als eine Zeile eingeben und Sie überlegen sich einen Weg, wie die RETURN-Taste ignoriert werden kann. Erst schreiben Sie sich eine Funktion

```
def ignore(event)
    pass
```

und nehmen die Bindung

```
text.bind('<Return>', ignore)
```

vor. Was passiert? Gar nichts. Der User kann die RETURN-Taste problemlos weiterhin mit Erfolg einsetzen. Was ist los? Die von Ihnen vorgenommene Bindung existierte nur auf der untersten Ebene, der Instanzebene. Das Standardverhalten ist aber an die Klassenebene gebunden! Sie können nun die Methode *bind\_class* anstelle von *bind* einsetzen. Aber das würde das Verhalten *aller* Widgets dieser Klasse in der Anwendung verändern! Ein einfacherer Weg besteht darin, den eben geschriebenen Code zu verwenden und anstelle von *pass* die Funktion *ignore* tatsächlich etwas ausführen zu lassen, zum Beispiel mit *return 'break'* ein Stringobjekt zurückzugeben.

Wenn Sie tatsächlich eine Klassenbindung vornehmen wollen, schreiben Sie

```
top.bind_class('Text', '<Return>', ignore)
```

in Ihren Code. Doch vermeiden Sie dies nach Möglichkeit. Schreiben Sie dann lieber eine eigene, von den Textwidgets abgeleitete Klasse, in der Sie diese Bindung vornehmen.

## 8.5 Protokolle

Zum Schluss noch ein paar Worte zum Thema Protokolle. Es gibt bei Tkinter die Möglichkeit, neben *event handlers* auch *protocol handlers* einzurichten. Protokolle beziehen sich auf die Interaktion zwischen der Anwendung und dem Windowmanager. An dieser Stelle wird nur ein einziges Beispiel gegeben.

Das am häufigsten eingesetzte Protokoll heisst `WM_DELETE_WINDOW`, das definiert, was passieren soll wenn der User ein Fenster schliesst.

Benutzen Sie für die Installation des Protokoll-Handlers die Methode *protocol* des Widgets, das ein root- oder Toplevel-Window sein muss. Durch den Aufruf

```
widgetname.protocol('WM_DELETE_WINDOW', handler)
```

wird Tkinter das widget nicht mehr automatisch schliessen, sondern durch den Handler zum Beispiel eine Nachricht ausgeben. Um den Handler dann das Window schliessen zu lassen, muss die *destroy*-Methode des Fensters aufgerufen werden.

Beispiel:

```
#File: protocol1.py
from Tkinter import *
import tkMessageBox
def callback():
    if tkMessageBox.askokcancel('Quit', 'Wirklich aufhören?'):
        root.destroy()
root=Tk()
root.protocol('WM_DELETE_WINDOW', callback)
root.mainloop()
```

Window-Manager-Protokolle waren ursprünglich ein Teil des X-Window Systems. Sie werden in einem Dokument mit dem Namen *Inter-Client Communication Conventions Manual* oder *ICCCM* beschrieben. Weitere Protokolle sind zum Beispiel `WM_TAKE_FOCUS` und `WM_SAVE_YOURSELF`.

## Kapitel 9

# Anwendungsfenster

Jetzt wird es wieder sehr praktisch. In diesem Kapitel untersuchen wir

- Base Windows
- Menüs
- Werkzeugleisten und
- Statuszeilen

### 9.1 Base Windows

Dieser Abschnitt bringt nicht allzuviel Neues, denn das Meiste über Basisfenster haben wir schon gesagt. Wenn der Konstruktor der Klasse *Tk* aufgerufen wird, erzeugt Tkinter ein Basisfenster, dem wir meistens den Namen *root* geben:

```
from Tkinter import *  
root=Tk()  
root.mainloop()
```

ist wohl das kleinste, funktionierende Tkinter-Programm. *root* ist tatsächlich das »Wurzelfenster«. Alle anderen Fenster sind »Kinder« dieses Hauptfensters.

Wenn ein zweites Fenster mit den Aufgaben des Wurzelfensters versehen werden soll, so ist der Konstruktor der Klasse *Toplevel* aufzurufen. Probieren Sie bitte das folgende Programm aus:

```
from Tkinter import *
root=Tk()
top=Toplevel()
root.mainloop()
```

Es gibt keine Notwendigkeit, das Toplevel-Fenster zu packen, zumal es ja automatisch auf dem Bildschirm generiert wird. Ja, es gäbe sogar eine Fehlermeldung.

## 9.2 Menüs

Wahrscheinlich haben Sie darauf schon gewartet? ;-) Nun kommt es endlich! Für Menüs gibt es – wer hätte das gedacht – eine spezielle Widget-Klasse mit dem Namen *menu*.

Wesentliche Methoden sind:

- *add\_command(label=string, command=callback)*, durch die ein gewöhnlicher Menüeintrag erfolgt
- *add\_separator()* fügt lediglich eine Trennlinie ein, um Menüeinträge zu gruppieren
- *add\_cascade(label=string, menuinstance=menuinstance)* fügt ein Untermenü ein. Dies kann sowohl ein Pull-Down- als auch ein Fold-Out-Menü sein, je nachdem wie das Elternmenü aussieht.

Ein kleines Beispiel:

```
#File: menu1.py
from Tkinter import *
def callback():
    print 'called the callback'
root=Tk()
#Erstellen des Menüs
```



Abbildung 9.1: Einfaches Menü

```
menuobj=Menu(root) #Menükonstruktor
root.config(menu=menuobj)
filemenu=Menu(menuobj)
menuobj.add_cascade(label='File', menu=filemenu)
filemenu.add_command(label='New', command=callback)
filemenu.add_command(label='Open', command=callback)
filemenu.add_separator()
filemenu.add_command(label='Exit', command=callback)
helpmenu=Menu(menuobj)
menuobj.add_cascade(label, menu=helpmenu)
helpmenu.add_command(label='About', command=callback)
mainloop()
```

Das Ergebnis sehen Sie in dem folgenden Bild 9.1:

*menuobj* ist eine Menüinstanz, die zu Beginn des Programms durch den Konstruktor *Menu(root)* erzeugt wird, wobei der Name des Widgets übergeben wird, in dem das Menü entstehen soll. Damit werden auf die Menüinstanz bestimmte Eigenschaften übertragen, zum Beispiel die Frage, ob sie ein Popup-Menü sein soll oder nicht. Dann wird durch die *config*-Methode von *root* das Menü an das Rootfenster gebunden, d.h. es wird eine Menüzelle in *root* erstellt. Das Menü muss nicht gepackt werden!

Schliesslich wird eine zweite Menüinstanz erzeugt, für die das eben erzeugte *menuobj* das Container-Widget darstellt. Dann werden die entsprechenden



Einträge vorgenommen und auf ähnliche Art und Weise ein Hilfemenü erstellt.

### 9.3 Werkzeugleisten (Toolbars)

Viele Anwendungen haben unterhalb der Menüzeile eine Werkzeugleiste, die typischerweise eine Anzahl von Buttons für gewöhnliche Funktionen wie 'open', 'print' oder 'undo' aufweisen. Um solche Buttons aufzunehmen, sollten wir ein *Frame*-Widget verwenden, da in diesem Fenster mehrere Widgets untergebracht werden sollen.

```
#File: toolbar1.py
from Tkinter import *
root=Tk()
def callback():
    print 'called the callback'
toolbar=Frame(root)
b=Button(toolbar, text='new', width=6, command=callback)
b.pack(side=LEFT, padx=2, pady=2)
b=Button(toolbar, text='open', width=6, command=callback)
b.pack(side=LEFT, padx=2, pady=2)
toolbar.pack(side=TOP, fill=X)
```

Das Programm dürfte selbsterklärend sein. Lediglich ein paar Worte wollen wir zum Packen verlieren. Die Buttons werden von links aneinander gesetzt und die ganze Werkzeugleiste selbst wird ganz nach oben an das Fenster gepackt.

Um die Dinge einfach zu halten, verwendet dieses Beispiel nur Text als Aufschriften. Wenn Sie ein Icon sehen wollen, muss der Photo-Image Konstruktor geladen werden, um ein kleines Icon von der Platte zu holen. Mit der *image* Option der Buttons wird dann das Icon dargestellt. Die folgende Abänderung des Programms

```
#File: toolbar1.py
from Tkinter import *
root=Tk()
def callback():
```



Abbildung 9.2: Menü mit Icons

```

    print 'called the callback'
    toolbar=Frame(root)
    image=PhotoImage(file='new.gif')
    b=Button(toolbar, image=image,command=callback)
    b.pack(side=LEFT, fill=BOTH)
    image2=PhotoImage(file='open.gif')
    b=Button(toolbar, image=image2,command=callback)
    b.pack(side=LEFT, fill=BOTH)
    toolbar.pack(side=TOP, fill=X)
    mainloop()

```

liefert das Bild 9.2:

Achten Sie bitte darauf, dass die Image-Formate von *PhotoImage* noch auf PPM, GIF und PGM (grayscale) beschränkt sind.

## 9.4 Statuszeilen

Schliesslich haben die meisten Anwendungen eine Statuszeile am Fuss jedes Anwendungsfensters. So eine Statuszeile zu implementieren ist wirklich trivial: Sie müssen lediglich ein entsprechend konfiguriertes Label-Widget verwenden und die entsprechende *text*-Option rekonfigurieren. Es ist sehr anzuraten, für diese Vorgänge eine eigene Klasse zu definieren, die über entsprechende *set* und *clear*-Methoden verfügt:

```

#File: StatusBar.py
class StatusBar(Frame):
    def __init__(self, master):
        Frame.__init__(self, master)
        self.label=Label(self, bd=1, relief=SUNKEN, anchor=W)
        self.label.pack(fill=X)

```

```
def set(self, format):
    self.label.config(text=format)
    self.label.update_idletasks()
def clear(self):
    self.label.config(text='')
    self.label.update_idletasks()
```

Der Anker beim 'label'-Widget ist eine Option, die auf Himmelsrichtungen reagiert. Der Text soll also im Westen beginnen. Die Methode *update\_idletasks()* führt die eben durchgeführte Konfiguration sofort aus. Der eigentliche Trick besteht darin, dass wir vom *Frame* Widget eine Vererbung vorgenommen haben und eine neue Klasse entwickelten, die eben *StatusBar* heisst. Jetzt können wir mit

```
status=StatusBar(root)
status.pack(side=BOTTOM, fill=X)
status.set('Achtung')
```

eine beliebige Statuszeile einrichten. Dies ist die eleganteste Methode, die sehr viel besser ist, als einfach nur ein Label ans Ende zu setzen.

## Kapitel 10

# Standard Dialoge und Message-Boxen

Bevor wir uns ansehen, wie wir ein Applikationsfenster füllen, werfen wir einen Blick auf einen anderen, sehr wichtigen Teil der GUI-Programmierung: das Darstellen von Dateneingaben und Nachrichten-Fenstern (Message Boxes). Beide Arten sind sogenannte Dialogfenster, da sie mit dem Anwender in Interaktion treten.

Bei der Dateneingabe wird der Anwender aufgefordert, irgendwelche Zahlen oder Textzeilen in ein vorgefertigtes Fenster einzugeben. Eine andere, sehr wichtige Art der Dateneingabe ist die Auswahl von Farben.

Message-Boxes dagegen sind die Antworten oder Hinweise und Warnungen der Anwendung an den User.

Beide Fenstertypen sind sozusagen eine Art Kommunikationsheft, das der Anwender benötigt, um mit der Anwendung zu kommunizieren.

### 10.1 Message-Boxes

Die Messageboxen zerfallen in zwei Gruppen: den eigentlichen Nachrichten (*showboxes*) und den Fragen (*askboxes*).

Die Dialoge, welche die einzelnen Boxtypen aufrufen, sind in dem Modul *tkMessageBox* enthalten. Der Aufruf der Funktionen, die wir gleich im Einzelnen besprechen werden, ist immer gleich:

- `tkMessageBox.function(title, message, options1)`, falls das Modul `tkMessageBox` mit `import` importiert wurde und
- `function(title, message, options2)`, falls das Modul `tkMessageBox` mit `from` importiert wurde.

Das Argument *title* wird in der Titelzeile des Fensters angezeigt und die *message* im zentralen Feld des Fensters. Mit Hilfe des Newline-Zeichens `'\n'` kann sich die Message über mehrere Zeilen erstrecken. Optionen können das Aussehen des Fensters verändern, sind aber optional und werden später im Abschnitt 10.1.5 besprochen.

### 10.1.1 showinfo

... liefert eine einfache Message-Box, die nichts weiter als eine Mitteilung macht.

Beispiel:

```
from tkMessageBox import *
def zeige(zahl):
    if zahl%2==0:
        showinfo('Modulotester','Die Zahl ist gerade')
    else:
        showinfo('Modulotester','Die Zahl ist ungerade.')
```

liefert nach Eingabe von

```
>>>zeige(4)
```

das Fenster 10.1.

---

<sup>1</sup>*optional*

<sup>2</sup>*optional*

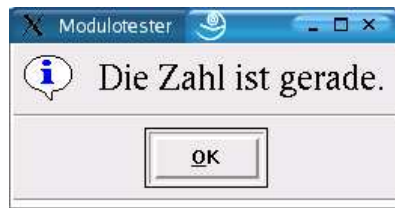


Abbildung 10.1: MessageBox

### 10.1.2 showwarning

Warnboxen eignen sich besonders gut für Ausnahmehandlungen mit Hilfe von *try* und *except*. Betrachten wir ein Programm, das eine Datei öffnen soll und in dem Augenblick, wo es feststellt, dass diese Datei nicht vorhanden ist, eine Warnung ausgibt:

```
from tkinter import messagebox
def oeffne(filename):
    try:
        fp=open(filename,'r')
    except:
        showwarning('Open file', 'Kann die Datei\n%s\nnicht öffnen.'%filename)
    return
```

Das Ergebnis, nachdem am Eingabeprompt ein Aufruf wie

```
>>>oeffne('babsi')
```

erfolgt ist<sup>3</sup>, und *babsi* *nicht* existiert, ist ein Fenster wie das in Abbildung 10.2 gezeigte:

%s ist natürlich ein Platzhalter für das Stringobjekt, das *filename* zugeordnet ist.

---

<sup>3</sup>Bedenke, dass die Filenamen *immer* als Strings eingegeben werden müssen.



Abbildung 10.2: Warnungsbox

### 10.1.3 showerror

Error-Boxen haben eine ganz ähnliche Funktion wie Warnungen. Während eine Warnung den Programmablauf jedoch nicht unterbricht, verwendet man Error-Boxen in der Regel stets dann, wenn das Programm nicht weiter ausgeführt werden kann.

Beispiel:

```
from tkinter import *
def teile(zahl1,zahl2):
    try:
        return (zahl1/zahl2)
    except:
        if zahl2==0:
            showerror('Overflow','Division by Zero')
        return
```

Wenn am Eingabeprompt

```
>>>teile(4,0)
```

einggegeben wird, muss die Box in Abbildung 10.3

erscheinen.



Abbildung 10.3: Errorbox

#### 10.1.4 Question-Boxes

Die *askquestion*-Box liefert die Antworten *1* für 'ja' und *0* für 'nein' zurück. Let's try:

```
from tkMessageBox import *
askyesno('Print', 'Print this report?')
```

Wir verzichten hier auf eine Abbildung, da diese fast genauso aussieht wie die *showinfo*-Box mit der Ausnahme, dass zwei Buttons mit den Aufschriften 'Ja' und 'Nein' auftauchen.

Die *askokcancel*-Box unterscheidet sich von der *askquestion*-Box nur dadurch, dass anstelle 'Ja' die Bezeichnung 'OK' und anstelle 'Nein' die Bezeichnung 'Abbruch' auftaucht.

Die dritte Box ist die *askyesno*-Box. Zwischen der *askquestion*- und der *askyesno*-Box gibt es keinen Unterschied.

Die vierte Box ist die *askretryignore*-Box, die ebenfalls eine alternative Antwort erwartet und vom Desing einer Warnungsbox ähnelt.

#### 10.1.5 Message-Box Optionen

Das Aussehen der Boxen kann in Grenzen durch Optionen, die als dritter Parameter eingegeben werden, verändert werden. Noch einmal: zwingend notwendig sind die Informationen

1. Titel der Box (1. Parameter)
2. Message der Box (2. Parameter)



Option	Datentyp	Beschreibung	Werte
<i>default</i>	constant	Gibt an, welcher Button der Standard sein soll.	ABORT, RETRY, IGNORE, OK, CANCEL, YES, NO
<i>icon</i>	constant	Gibt an, welches Icon geladen werden soll.	ERROR, INFO, QUESTION und WARNING
<i>message</i>	string	Die Nachricht, die dargestellt werden soll (zweites Argument).	keine
<i>parent</i>	widget	Benennt das Window, in dem die Box stehen soll.	keine
<i>title</i>	string	Die Kopfzeile der Box (erstes Argument).	keine
<i>type</i>	constant	Die Art der Box.	ABORTRETRYIGNORE, OK, OKCANCEL, RETRYCANCEL, YESNO oder YESNOCANCEL

## 10.2 Dateneingabe

Sehr viel interessanter sind die Dialoge, die es ermöglichen, Daten einzugeben. Diese Dialoge sind in dem Modul *tkSimpleDialog* gespeichert. Es gibt die Möglichkeit, *strings*, *numeric values*, *filenames* und *colors* auszuwählen.

### 10.2.1 *strings* mit *askstring*

Die *askstring*-Box benötigt zwei zwingende Informationen:

1. Welchen Titel soll die Box haben?
2. Welche Aufforderung soll an den User gegeben werden?

Optional können noch zwei Optionen angegeben werden:



Abbildung 10.4: Fragebox

**initialvalue:** vom Typ *string*. Gibt einen vorgefertigten String vor, falls das gewünscht wird. Die Standardvoreinstellung ist ein leerer String.

**parent:** vom Typ *widget*. Gibt an, in welchem Fenster die Box platziert werden soll.

Option	Type	Beschreibung
<i>initialvalue</i>	string	Gibt einen voreingestellten String vor. Defaultmässig ist ein leerer String eingestellt.
<i>parent</i>	widget	Gibt an, in welchem Fenster die Box auftauchen soll.

Beispiel:

```
from tkSimpleDialog import *
answer=askstring('Palins Message', 'And now to something',
initialvalue='Insert here')
print answer
```

Das entsprechende Fenster sieht so aus wie in der Abbildung 10.4 .

Wenn Sie auf *Cancel* klicken, wird das Objekt `NONE` zurückgegeben, ansonsten der String, der natürlich hier heissen muss 'completely different'.

### 10.2.2 *numerical values* und *askinteger* und *askfloat*

Mit *askinteger* und *askfloat* geht man genauso um. Neben den üblichen Optionen *initialvalue* und *parent* gibt es aber noch zwei weitere Optionen, die

sich *minvalue* und *maxvalue* nennen. Werden die Grenzen überschritten, taucht ein Warnfenster auf, das den Dialog nicht schliesst, sondern angibt, wie sich der User zu verhalten hat. Da die Fenster optisch nichts Neues liefern, verzichten wir an dieser Stelle auf eine Abbildung.

### 10.2.3 *filenames*

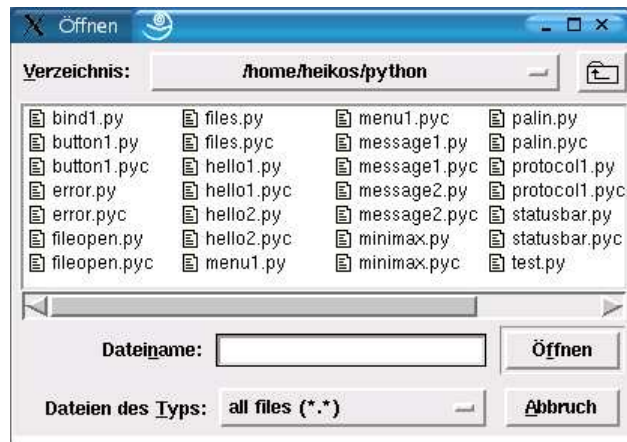
Um Dateien zu öffnen oder zu speichern gibt es die Funktionen *askopenfilename* und *asksavefilename*, die zu dem Modul *tkFileDialog* gehören. Die Boxen lassen sich denkbar einfach aufrufen, denn alle Parameter sind optional. Diese Optionen sind allerdings *sehr* nützlich:

Option	Type	Beschreibung
<i>defaulttextextension</i>	string	Gibt die Dateinamenserweiterung vor, sofern der User nicht eine andere wünscht. Der String sollte den führenden Punkt enthalten.
<i>filetypes</i>	list	Sequenz von (label, pattern) Tupeln. Dasselbe Label darf mit verschiedenen Patterns auftauchen.
<i>initialdir</i>	string	Gibt das Startverzeichnis an.
<i>initialfile</i>	string	Gibt eine vorgeschlagene Datei an. Wird von <i>askopenfilename</i> ignoriert.
<i>parent</i>	widget	Gibt wie üblich das Fenster an, an das die Box gebunden wird.
<i>title</i>	string	Gibt wieder den Titel der Box an.

Ein Beispiel:

```
from tkFileDialog import *
answer=askopenfilename(defaulttextextension='*.py',
filetypes[('all files','*.*')])
print answer
```

Dies liefert die sehr eindrucksvolle Box in der Abbildung 10.5. Natürlich wird die ausgewählte Datei als String zurückgegeben und zwar als *absoluter* Pfadname. Entsprechend funktioniert die Funktion *asksavefilename*.

Abbildung 10.5: *askopenfilename* und die zugehörige Box

#### 10.2.4 colors

Das letzte der Standarddialogmodule ist *tkColorChooser* mit der naheliegenden Funktion *askcolor*. Der Rückgabewert dieser Funktion ist ein 2-Tupel, dessen erste Komponente ein RGB-Triplett und dessen zweite ein Tk-Color String ist. Schauen wir uns dies am besten wieder an einem Beispiel an:

```
from tkColorChooser import *
answer=askcolor()
print answer
```

Die einzigen Optionen, die *askcolor* kennt, sind *initialcolor* vom Typ *color* und wiederum *parent* und *title*, wie auch bei den anderen Boxen. *initialcolor* ist entweder ein RGB-Triplett oder ein Tk-String. Bei der in Abbildung 10.6 dargestellten Box wird der Wert ((212,12,105),#d90c6c) zurückgegeben. Wird keine Farbe ausgewählt, lautet der Rückgabewert (NONE, NONE).

Abbildung 10.6: Farbauswahl mit *askcolor*

# Index

*askboxes*, 50

*background*, 28

*bd*, 31

*borderwidth*, 31

*Canvas*, XV

*cget*, 27

*config*, 28

*configure*, 28

*cursor*, 32

*detail*, 38

*Event-Schleife*, 4

*Events*, XV

*fonts*, 30

*foreground*, 28

*Frame*, 7

*gadget*, 3

*Handler*, 38

*highlightbackground*, 32

*highlightcolor*, 32

*highlightthickness*, 32

*Items*, XV, 15

*justify*, 31

*Label*, 3

*modifier*, 38

*Optionen*, 27

*pack*, 4

*relief*, 31

*showboxes*, 50

*Statuszeilen*, 48

*Textfarbe*, 28

*Tkconstants*, 1

*tkFont*, 30

*Tkinter*, 1

*tkinter*, 1

*tkMessageBox*, 50

*tkSimpleDialog*, 55

*Toolbars*, 47

*type*, 38

*widget*, 3

*Widgetfarbe*, 28

*Widgets*, XV

*windows*, 3

*wrapline*, 31

