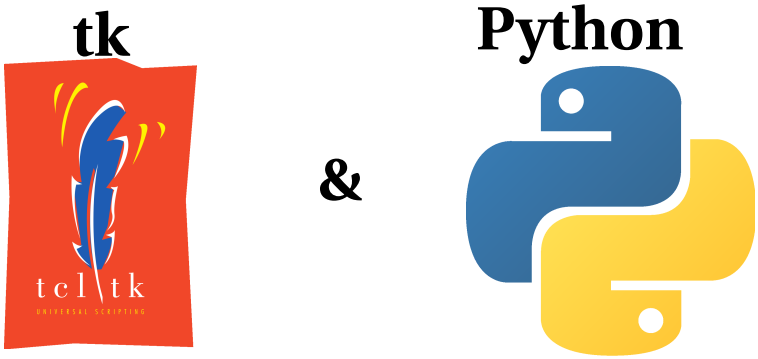
FSST - 4.Klasse

Tkinter - GUIs mit



Version 0.2

November 2020

Basierend auf mehreren online Tutorials und dem Buch `Python 3` von Johannes Ernesti und Peter Kaiser

Für die vierte Schulstufe an einer HTL

November 2020
HTL Anichstraße

Inhaltsverzeichnis

1 Change-Log

1.1 Version 02 (Nov.2020)

- Ein zusätzliches Beispiel (TicTacToe) auf Seite 33 hinzugefügt.
- Auf Anregung von Walter Müller ein zusätzliches Kapitel 8.0.1 zur Erklärung von tkinter Steuervariablen auf Seite 13 eingefügt. Danke Walter!

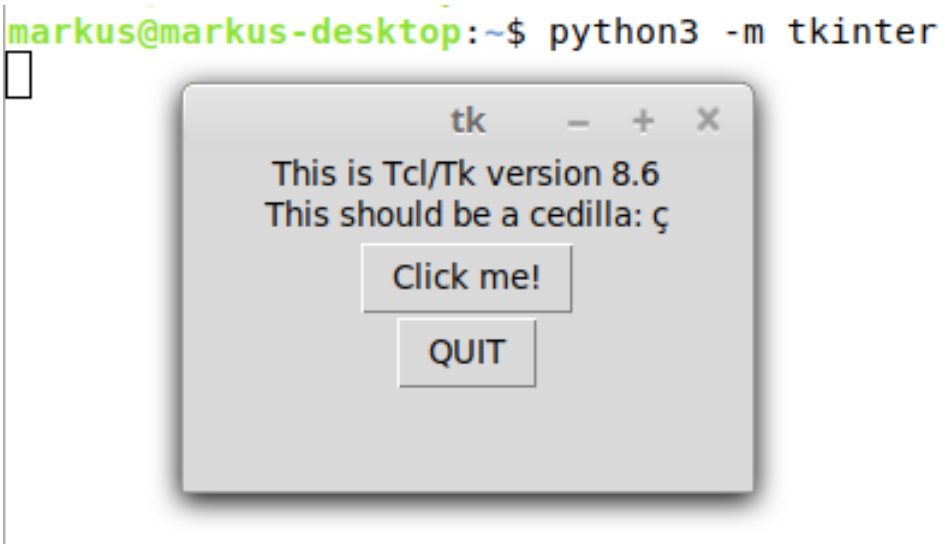
2 Vorausstellende Info

Dies ist ein *kurzes* Tutorial für eine erste Einführung in tkinter. Es deckt bei weitem nicht alle Möglichkeiten von tkinter ab und ist nur als Einstieg in die GUI-Programmierung gedacht.
Für eine detailreiche Einführung oder als Nachschlagewerk empfehle ich:
https://web.archive.org/web/20190524140835/https://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html

Und nun, viel Spaß mit den ersten Python GUIs ;-)

3 Hintergrund

Tk ist ein freies, plattformübergreifendes GUI-Toolkit zur Programmierung von grafischen Benutzeroberflächen. Tk wurde ursprünglich für die Scriptsprache Tcl entwickelt, es wurden aber aufgrund der einfachen Erlernbarkeit von vielen Programmiersprachen als einfache GUI Bibliothek importiert.
Tkinter ist eine Sprachanbindung für Tk für die Programmiersprache Python. Der Name steht als Abkürzung für Tk interface. Tkinter war das erste GUI-Toolkit für Python, weshalb es inzwischen auf Mac OS, Linux und Windows zum standard Lieferumfang von Python gehört.
Natürlich gibt es noch andere Möglichkeiten mit Python GUIs zu programmieren z.B. mit PyGame oder dem Qt Framework, aber für Einsteiger ist tkinter leichter zu erlernen als z.B. Qt.
Tkinter sollte mit python installiert worden sein und kann mit folgendem Programm getestet werden:
`python -m tkinter  ODER  python3 -m tkinter`
Sollte folgenden Output liefern:



4 Hello World als GUI

Wir beginnen ganz klassich mit Hello World ;-)
Dazu verwenden wir unser erstes **widget** - ein Label! Ein widget ist ein Kunstwort das aus Wi(ndow) und (Ga)dget abgeleitet wurde und bezeichnet in den meisten GUI Bibliotheken ein vordefiniertes Element einer GUI.
Für unser erstes Programm verwenden wir nun ein Label. Ein Label ist eine Art Display-Box in der man Text schreiben und auch updaten kann.

Code (quick and dirty):

```
from tkinter import *

root_window = Tk()

w = Label(root_window,
        text="Hello World!")
w.pack()

root.mainloop()
```

Code (nice and object oriented):

```
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.create_widgets()

    def create_widgets(self):
        self.hi_there = tk.Label(self)
        self.hi_there["text"] = "Hello World"
        self.hi_there.pack()

root_window = tk.Tk()
app = Application(master=root_window)
app.mainloop()
```

Output von beiden Varianten:

5 Einführung mittels Hello World

5.1 Hello World erklärt

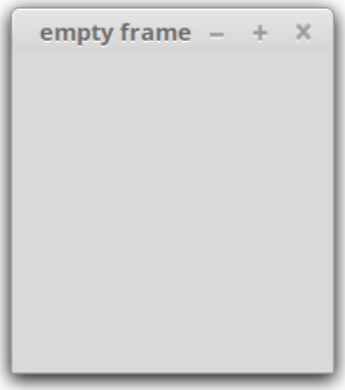
Wir beginnen unseren Einstig in `tkinter` mit einer Analyse unseres Hello World Beispiels:

```
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.create_widgets()

    def create_widgets(self):
        self.hi_there = tk.Label(self)
        self.hi_there["text"] = "Hello World"
        self.hi_there.pack()

root_window = tk.Tk()
app = Application(master=root_window)
app.mainloop()
```

5.1.1 Frame vs Window und Sichtbar-Machen

Zunächst wird das Modul `tkinter` eingebunden und eine Klasse erstellt, die von `tkinter.Frame` erbt. Ein `Frame` ist ein Steuerelemt welches unsichtbar ist und uns als Container für weitere Steuerelemente dient.
Code für einen leeren Frame der Größe 100x100 Pixel:

```
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.height = 100
        self.width = 100

root_window = tk.Tk()
root_window.title("empty frame")
app = Application(master=root_window)
app.mainloop()
```



Betrachten wir noch die vier Code-Zeilen am Ende:
Zuerst wird eine Instanz (ein Objekt) der Klasse `tkinter.Tk` erzeugt. Es representiert das klassische *Fenster* oder *Window* welches wir von GUIs gewohnt sind. Es hat die typischen Steuerelemente wie minimieren (_), maximieren (+) und schließen (x) sowie eine dynamische Größenveränderung mittels drag and drop und das *look and feel* des verwendeten Betriebssystems.
Wir können dem Window mit der Methode `.title("empty frame")` auch einen Titel geben.
Dieses Window ist das erste *(root oder master)* Element unsere GUI.
In der vorletzten Code Zeile erzeugen wir ein Objekt unserer `Application` Klasse und übergeben ihr in der `__init__()` Methode unser `root_window` als `master`.
In der letzen Code-Zeile wird die GUI durch die Methode `.mainloop()` gestartet (d.h. sichtbar gemacht und an das Eventhandling übergeben. Eventhandling -> die GUI reagiert auf Events z.B. Left Mouse Click auf das x im Window).

5.1.2 Constructor und pack()

Im Konstruktor (der `__init__()` Methode) von `Application` wird der mit Hilfe von `super()` der Konstruktor der Basisklasse aufgerufen und `root_window` übergeben.
Der Aufruf der Methode `.pack()` meldet das Widget, für das die Methode aufgerufen wurde, beim *Packer* an. Der

Packer ist ein Layout-Manager, der die angemeldeten Steuerelemente (Labels, Buttons, usw.), nach Vorgaben des Programmieres, automatisch auf der GUI anordnet. Dies ist einer manuellen Anordnung (mit fixen Positionen in Pixel) vorzuziehen, weil es sonst beim re-sizing von Fenstern aufwändig wird die neuen Positionen bzw. Skalierungen selbst zu rechnen.
Mehr zu Packer dann in Kapitel 6.1 auf Seite 4.
Als letztes wir im Konstruktor noch die Methode `create_widgets()` aufgerufen. In dieser Methode erzeugen wir weitere Steuerelemente (in diesem Beispiel nur ein simples Label) und melden dies ebenfalls beim Packer an. Mehr zu verschiedenen Steuerelementen dann in Kapitel 8 auf Seite 13.

6 Layout- oder Geometry-Manager

Layout-Manager haben (in allen GUI-Bibliotheken von Programmiersprachen) die Aufgabe die Anordnung von Widgets in der GUI für den Programmierer zu vereinfachen.
Tkinter bietet drei verschiedene Manager an. Den Pack-Manager haben wir schon kennengelernt. Zusätzlich gibt es noch den Grid- und den Place-Geometry-Manager.
Im Folgenden werden kurz auf die drei Manager eingehen.

6.1 Der Packer

Der pack-Manager (bzw. Packer) packt Widgets prinzipiel in Zeilen oder Spalten unter Berücksichtigung von Vorgaben im Programm. Wichtig ist, dass die Anordnung hierarchisch ist! Das bedeutet, dass jedes Steuerelement über ein übergeordnetes Vater-Element verfügt und es beliebig viele Kind-Elemente haben darf (master-slave). Die Hierarchie der Steuerelemente ist wichtig, um die Arbeitsweise des Packers zu verstehen. Dieser ordnet nämlich die Kindelemente innerhalb ihres Vaterelementes an und dann das Vaterelement mitsamt den Kindelementen in dessen Vaterelemente an usw.
Es ist also sinnvoll, Steuerelemente zu kapseln (meist mit dem `frame` element), damit sie gemeinsam vom Packer angeordnet werden können.
Dem Packer können verschiedene Vorgaben gemacht werden und einige der vielen verschiedenen Gestaltungsmöglichkeiten mit dem pack-Manager sollen hier anhand einfacher GUIs aufgezeigt werden.

6.1.1 Parameter für den packer

Parameter die der `.pack()` Methode übergeben werden können:

Parameter	**Mögliche Werte**	**Bedeutung**
after	Widget	Das Steuerelement soll nach dem angegebe- nen Widget gepackt werden.
anchor	"n", "ne", “e", “se", “s", “sw", "w", "nw","center"	Wenn der dem Widget zugeteilte Bereich größer ist als das Widget, kann über anchor die Ausrichtung des Widgets innerhalb dieses Bereichs festgelegt werden. Bei den möglichen Werten handelt es sich um die Himmelsrichtungen sowie »zentriert«.
before	Widget	Das Steuerelement soll vor dem angegebenen Widget gepackt werden.
expand	bool	Legt fest, ob die Position des Widgets bei Vergrößerung des Master-Widgets angepasst werden soll
fill	"x", "y", "both", "none"	Die Größe des Widgets wird bei Vergrößerung des Master-Widgets angepasst. Die Größe kann dabei horizontal, vertikal, vollständig oder gar nicht angepasst werden.
in	Widget	Fügt das Steuerelement in das angegebene Master-Widget ein.
ipadx ipady	int	Pixelgröße für das horizontale bzw. vertikale innere Padding
padx pady	int	Pixelgröße für das horizontale bzw. vertikale äußere Padding
side	"left", "right","top", "bottom"	Die Seite des Arbeitsbereichs, an der das Wid- get eingefügt wird. Ein auf der linken bzw. rechten Seite platziertes Widget beansprucht die gesamte Höhe und ein oben bzw. unten platziertes Widget die gesamte Breite des Arbeitsbereichs. Differenziertere Layouts erreichen Sie mithilfe eines Frame - Widgets.

6.1.2 Pack Beispiele

Vorausstellend: Um den Beispielcode in diesem Kapitel kurz zu halten, wird auf ein OOP Ausführung verzichtet!

6.1.2.1 Packer-Beispiel 1

Es wird ein Tk-Fenster erzeugt und direkt darauf (ohne `frame`) werden vier `label` gelegt. Es werden Text, Hintergrund- und Vordergrundfarbe der einzelnen Labels geändert. Dem Packer werden KEINE explizieten Anweisungen übergeben.

```
from tkinter import *
tkFenster = Tk()
tkFenster.title("pack example 01")

label1 = Label(master=tkFenster, text="eins", bg="red", fg="white")
label2 = Label(master=tkFenster, text="zwei", bg="green", fg="black")
label3 = Label(master=tkFenster, text="drei", bg="blue", fg="white")
label4 = Label(master=tkFenster, text="vier", bg="yellow", fg="black")
label1.pack()
label2.pack()
label4.pack()
label3.pack()

tkFenster.mainloop()
```

Die Anordnung erfolgt vertikal in der Reihenfolge in denen die Elemente dem Packer übergeben wurden (vier vor drei). Die horizontale Breite wird durch das längste Element (zwei) bestimmt.

6.1.2.2 Packer-Beispiel 2

Es wird ein Tk-Fenster erzeugt und direkt darauf (ohne `frame`) werden vier `label` gelegt. Es werden Text, Hintergrund- und Vordergrundfarbe der einzelnen Labels geändert. Dem Packer werden expliziete Anweisungen zum layout (side) und zum padding übergeben.

```
from tkinter import *
tkFenster = Tk()
tkFenster.title("pack example 02")

label1 = Label(master=tkFenster, text="eins",
        bg="red", fg="white")
label2 = Label(master=tkFenster, text="zwei",
        bg="green", fg="black")
label3 = Label(master=tkFenster, text="drei",
        bg="blue", fg="white")
label4 = Label(master=tkFenster, text="vier",
        bg="yellow", fg="black")
label1.pack(side = "right", ipadx = 5, ipady = 5,
        padx = 5, pady = 5)
label2.pack(side = "right", ipadx = 20, ipady = 20,
        padx = 20, pady = 20)
label4.pack(side = "bottom", ipadx = 5, ipady = 5,
        padx = 5, pady = 5)
label3.pack(side = "bottom")
tkFenster.mainloop()
```

Die Anordnung erfolgt diesmal in der Reihenfolge in denen die Elemente dem Packer übergeben wurden und unter Berücksichtigung der Steuerparameter.

6.1.2.3 Packer-Beispiel 3

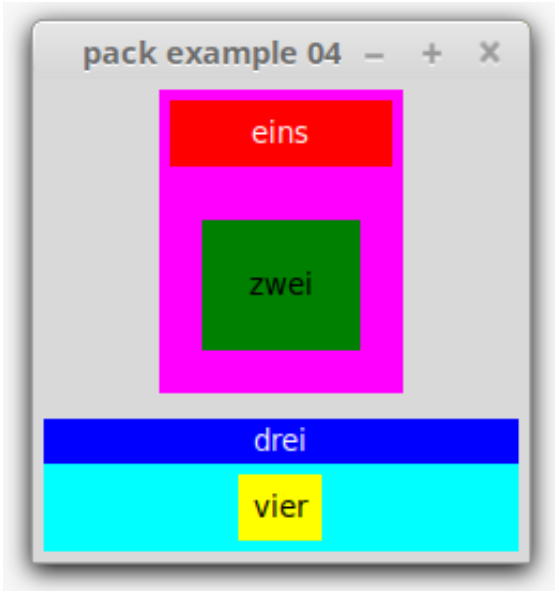
Wie Beispiel 2 oben aber nun noch mit der fill Anweisung.

```
from tkinter import *
tkFenster = Tk()
tkFenster.title("pack example 02")

label1 = Label(master=tkFenster, text="eins",
        bg="red", fg="white")
label2 = Label(master=tkFenster, text="zwei",
        bg="green", fg="black")
label3 = Label(master=tkFenster, text="drei",
        bg="blue", fg="white")
label4 = Label(master=tkFenster, text="vier",
        bg="yellow", fg="black")
label1.pack(fill = "x", ipadx = 5, ipady = 5,
        padx = 5, pady = 5)
label2.pack(side = "right", ipadx = 20, ipady = 20,
        padx = 20, pady = 20)
label4.pack(side = "bottom", ipadx = 5, ipady = 5,
        padx = 5, pady = 5)
label3.pack(fill = "both")
tkFenster.mainloop()
```

6.1.2.4 Packer-Beispiel 4

Nun verwenden wir zwei zusätzliche Frames um mehr Kontrolle über das Layout zu erlangen.

```
from tkinter import *
tkFenster = Tk()
tkFenster.title("pack example 04")

rahmen1 = Frame(master=tkFenster, bg='magenta')
rahmen1.pack(side='top', padx='5', pady='5')

rahmen2 = Frame(master=tkFenster, bg='cyan')
rahmen2.pack(side='bottom', fill = "x", padx='5',
        pady='5')

label1 = Label(master=rahmen1, text="eins",
        bg="red", fg="white")
label2 = Label(master=rahmen1, text="zwei",
        bg="green", fg="black")
label3 = Label(master=rahmen2, text="drei",
        bg="blue", fg="white")
label4 = Label(master=rahmen2, text="vier",
        bg="yellow", fg="black")
label1.pack(fill = "x", ipadx = 5, ipady = 5,
        padx = 5, pady = 5)
label2.pack(side = "right", ipadx = 20, ipady = 20,
        padx = 20, pady = 20)
label4.pack(side = "bottom", ipadx = 5, ipady = 5,
        padx = 5, pady = 5)
label3.pack(fill = "both")

tkFenster.mainloop()
```



Bitte den hierarchischen Haufbau beachten! Die `frames` haben als *master* das Fenster und die `labels` haben als *master* die frames.

6.2 Der Grid-Geometry-Manager

Der Grid Manager arbeitet mit einer zwei dimensionalen Tabelle. Das Master Widget wird in eine Nummer von Zeilen und Spalten unterteilt und jede Zelle in dieser Tabelle kann ein eigenes Widget beinhalten.
Die Verwendung des Grid-Managers ist sehr einfach: Man erstellt Widgets und ruft die `.grid()` Methode auf. Als Parameter kann man dann Zeile und Spalte angeben in der das Widget plaziert werden soll. Zusätzlich kann noch wie beim Pack-Manager ein Padding eingestellt werden (mit `padx` und `pady`) und mit dem Parameter `sticky` kann die Ausrichtung des Widgets in der Zelle bestimmt werden. Die Parameter Werte für `sticky` sind die Himmelsrichtungen als String abgekürzt, also z.B. 'n', 'e', 's', 'w' oder 'nw', 'se' usw.

Automatische Größenanpassung:
Wenn man möchte, dass das Widget im einer grid-Spalte-Zeile bei dynamisch die Größe ändert wenn die Fenstergröße verändert wird gilt es ein paar Dinge zu beachten:

- Auf dem `master-widget`, auf dem das `grid`-Layout liegt, muss für jede Spalte und Zeile ein sogenanntes `weight` eingestellt werden. Dies geschieht mir der `.columnconfigure(nr, weight=1)` und der `rowconfigure(nr, weight=1)` Methode. Der `weight` Parameter gewichtet wie stark das grid wachsen kann wenn Platz zur Verfügung steht. z.B. Wenn in einem Fenster zwei Frames in zwei grid-Spalten plaziert sind, und beide `weight=1` haben, wachsen sie im gleichen Maße: Sind 10 Pixes frei, bekommt jeder 5.
  Haben sie aber unterschiedliche Gewichtungen z.B. einer hat `weight=1` und der andere `weight=2`, dann wächst der Zweite bei z.B. 6 freien Pixeln vier Pixel und der erste nur zwei Pixel.
  Die `rowconfigure()` und `columnconfigure()` Methoden werden am besten in einer Schleife aufgerufen weil man ja jede Zeile und Spalte seperat konfigurieren muss.
- Zusätzlich muss jedes Widget welches in das grid plaziert wird und mit dem grid expandieren soll den Parameter `sticky=N+S+E+W` verwendet. Damit hängt es quasi an allen Seiten seiner Gridzelle fest und wird mit expandiert, wenn sich die Zellengröße ändert.

Für ein Beispiel zur automatischen Größenänderung siehe Kapitel 6.2.1.3 auf Seite 9.

6.2.1 Grid-Manager Beispiele

Wie schon bei den Beispielen zum Pack-Manager werden wir auch hier auf OOP Code verzichten, um die Beispiel mit wenigen Code-Zeilen realisieren zu können.

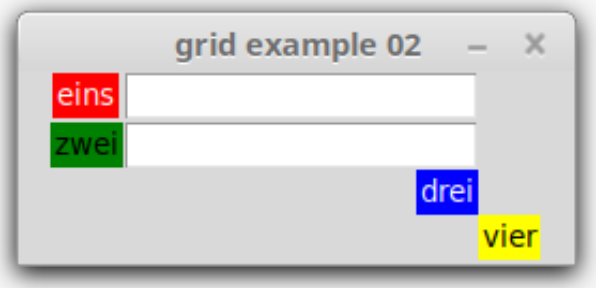
6.2.1.1 Grid-Beispiel 1

```
from tkinter import *
tkFenster = Tk()
tkFenster.title("grid example 01")
frame1 = Frame(master=tkFenster)
frame1.pack()

label1 = Label(master=frame1,
    text="eins", bg="red", fg="white")
label2 = Label(master=frame1,
    text="zwei", bg="green", fg="black")
label3 = Label(master=frame1,
    text="drei", bg="blue", fg="white")
label4 = Label(master=frame1,
    text="vier", bg="yellow", fg="black")
label1.grid(column=0)
label2.grid(row=0, column=1)
label3.grid(row=1, column=1)
label4.grid(row=1, column=2)
tkFenster.mainloop()
```

Bitte den hierarchischen Haufbau beachten und die gemischte Verwendung von `pack` und `grid`!

6.2.1.2 Grid-Beispiel 2

```
from tkinter import *
tkFenster = Tk()
tkFenster.title("grid example 02")
tkFenster.geometry("260x90")
tkFenster.resizable(False,False)
frame1 = Frame(master=tkFenster)
frame1.pack()

label1 = Label(master=frame1,
    text="eins", bg="red", fg="white")
label2 = Label(master=frame1,
    text="zwei", bg="green", fg="black")
label3 = Label(master=frame1,
    text="drei", bg="blue", fg="white")
label4 = Label(master=frame1,
    text="vier", bg="yellow", fg="black")
entry1 = Entry(master=frame1)
entry2 = Entry(master=frame1)

#Layout
label1.grid(row=0, column=0)
entry1.grid(row=0, column=1)
label2.grid(row=1, column=0)
entry2.grid(row=1, column=1)
label3.grid(row=2, column=1, sticky='e')
label4.grid(row=5, column=5)

tkFenster.mainloop()
```

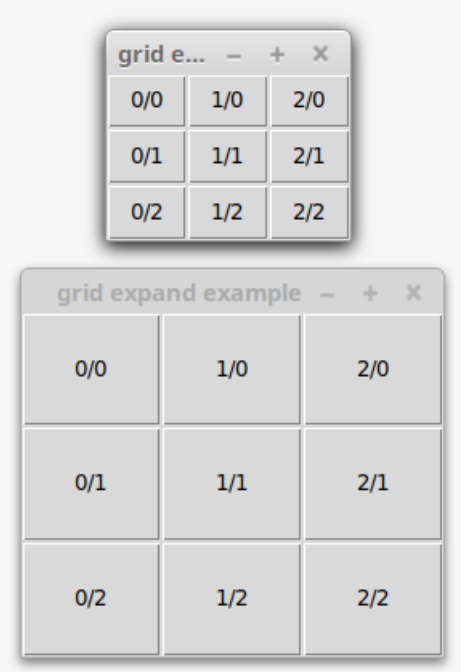


Bitte den hierarchischen Haufbau beachten und die gemischte Verwendung von `pack` und `grid`! Ausserdem wird die Fenster Geometrie vorgegeben und eine Größenänderung mit `.resizeable(False, False)` unterbunden.

6.2.1.3 Grid-Beispiel 3 - auto resizing

In diesem Beispiel soll die Größe eines grid-Layouts und der darin angeordneten Buttons automatisch an die Fenstergröße angepasst werden.

```
from tkinter import *

class MyApp(Frame):
    def __init__(self, master):
        super().__init__(master)
        #self.pack(fill=BOTH, expand=True)
        master.geometry("200x200")
        #frame
        self.f1 = Frame(master=master)
        self.f1.pack(fill=BOTH, expand=True)
        #parameter fuer die grid groesse
        self.grid_length = 3
        self.create_buttons()

        #make the grid layout expand
        for x in range(self.grid_length):
            self.f1.columnconfigure(x, weight=1)
            self.f1.rowconfigure(x, weight=1)


    def create_buttons(self):
        for x in range(self.grid_length):
            for y in range(self.grid_length):
                b = Button(master=self.f1,
                    text="{}/{}".format(x,y))
                b.grid(row=y, column=x,
                    sticky=N+S+E+W)



tk_window = Tk()
tk_window.title("grid expand example")
app = MyApp(tk_window)
app.mainloop()
```



6.3 Der Place-Geometry-Manager

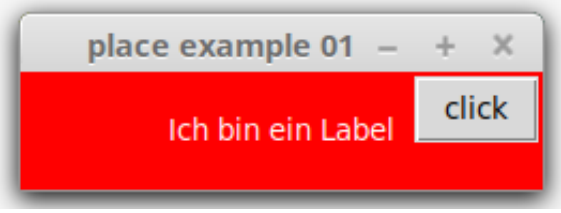
Place ist der einfachste der drei Layout-Manager. Er erlaubt es explizite Positionen von Widgets, entweder als absolute Position auf der GUI oder relative zu einem anderen Widget, zu setzen. Für dynamische GUIs verwendet man place eher nicht, da es zu aufwändig wird für jedes Widget Pixel-Positionen anzugeben. Aber für Spezialanordnungen ist er gut verwendbar.

6.3.1 Place-Manager-Beispiele

6.3.1.1 Place-Beispiel-01

In diesem Beispiel wird ein Button relative zu einem Label plaziert. Der Ankerpunkt des Buttons ist NW, die Ausrichtung soll von ganz rechts des Labels erfolgen (`relx=1`) und er soll um je zwei Pixel (x und y) nach innen verschoben sein.

```
from tkinter import *
tkFenster = Tk()
tkFenster.title("place example 01")
frame1 = Frame(master=tkFenster)
frame1.pack()

label1 = Label(master=frame1,
    text="Ich bin ein Label",
    bg="red", fg="white",
    height=3, width=30)
b = Button(master=frame1, text="click")

#Layout
label1.pack()
b.place(relx=1, x=-2, y=2, anchor=NE)

tkFenster.mainloop()
```



Bitte den hierarchischen Haufbau beachten und die gemischte Verwendung von `pack` und `grid`! Ausserdem wird die Fenster Geometrie vorgegeben und eine Größenänderung mit `.resizeable(False, False)` unterbunden.

7 Events

Beim Schreiben einer Tk-Anwendung wird nach dem Erstellen und Instanziieren der Applikationsklasse der Kontrollfluss durch Aufruf der Methode `mainloop()` an das Tk-Framework abgegeben. Es stellt sich die Frage, auf welchem Wege wir beispielsweise auf Eingaben des Benutzers reagieren können, wenn wir gar keine wirkliche Kontrolle über das Programm und die grafische Oberfläche haben. Aus diesem Grund ist in Tk eine Reihe von Events definiert. Ein Event ist beispielsweise ein Tastendruck oder Mausklick des Benutzers. Mithilfe der Methode `bind()` eines Widgets können wir eine selbst definierte Methode an ein Event binden. Eine an ein Event gebundene Methode wird vom Tk-Framework immer dann gerufen, wenn das entsprechende Event eintritt, der Benutzer also beispielsweise eine spezielle Taste gedrückt hat.

7.1 Event handling

Die Methode `bind(event, func, [add])` eines Steuerelements bindet die Funktion `func` an das Event `event` . Dabei muss für func das Funktionsobjekt einer Funktion übergeben werden, die genau einen Parameter, das sogenannte Event-Objekt, erwartet. Diese Funktion wird *Eventhandler* genannt. Wenn für den optionalen Parameter `add` der Wert `True` übergeben wird und es bereits andere Funktionen gibt, die an das Event gebunden sind, werden diese nicht gelöscht, sondern `func` wird nur in die Liste dieser Funktionen eingereiht. Standardmäßig werden vorherige Bindungen überschrieben.

Für den wichtigsten Parameter `event` muss ein String übergeben werden, der das Event spezifiziert, an das die Funktion `func` gebunden werden soll. Eine solche Event-Spezifikation hat die folgende Form:
`"<Modifier-Modifier-Type-Detail>"`
Die beiden Modifier-Einträge in der Event-Spezifikation sind optional und erlauben es beispielsweise, einen Mausklick bei gedrückter Shift-Taste und einen normalen Mausklick gesondert zu betrachten.
Mögliche Modifier sind: `Alt, Control, Shift, Lock, Double, Triple`

Der `Type` -Eintrag kennzeichnet den Event-Typ, und über den `Detail` -Eintrag kann eine nähere Spezifikation erfolgen, beispielsweise kann hier angegeben werden, welche Maustaste gedrückt werden muss, um das Event auszulösen.
Bsp:
`"<Strg-Shift-ButtonPress-1>"` oder einfach nur `"<ButtonPress-1>"`
Mögliche Event-Typen sind: `KeyPress, Key, KeyRelease, ButtonPress, ButtonRelease, Motion, Enter, Leave, FocusIn, FocusOut, Expose, Destroy`

Mittels `unbind` lassen sich solche Bindings auch wieder lösen, was z.B. bei dynamischen GUIs notwendig sein kann.

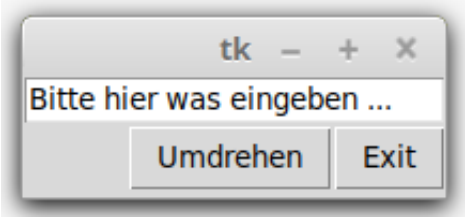
über event schreiben

7.2 Beispiele zum Eventhandling

7.2.1 Beispiel 1 - Textumdrehen

In diesem Beispiel soll eine einfache GUI erstellt werden, welche aus einem `Entry` Textfeld und zwei `Buttons` besteht. Wenn der erste Button gedrückt (von der linken Mousetaste) wird, soll sich der Text im Textfeld umdrehen. Wenn der zweite Button gedrückt wird soll sich die App schließen.
Da dies schon eine kleine Stand-Alone-Anwendung ist, werden wir wieder OOP-Code verwenden.

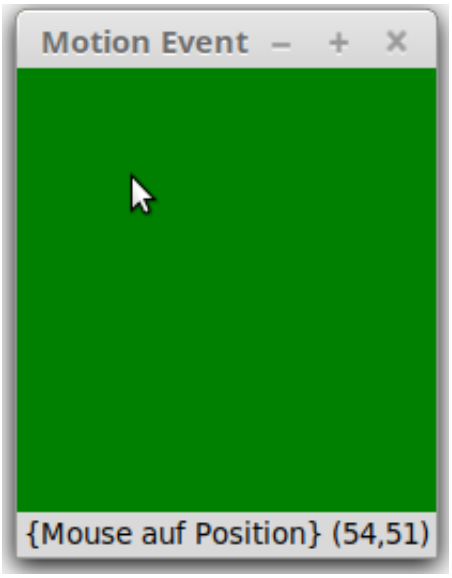
```
import tkinter

class MyFirstApp(tkinter.Frame):
    def __init__(self, master = None):
        super().__init__(master)
        self.pack(fill = "both")
        self.create_widgets()

    def create_widgets(self):
        #Textfeld erzeugen und default text einfügen
        self.text_entry = tkinter.Entry(self)
        self.text_entry.pack(fill = "x")
        self.some_text = tkinter.StringVar()
        self.some_text.set("Bitte hier was eingeben ...")
        self.text_entry["textvariable"] = self.some_text

        #Exit button erzeugen und command quit zuweisen
        self.exit_button = tkinter.Button(self)
        self.exit_button["text"] = "Exit"
        self.exit_button["command"] = self.quit
        self.exit_button.pack(side = "right")

        #Umdreh-Button erzeugen und and eine Funktion binden
        self.rev = tkinter.Button(self)
        self.rev["text"] = "Umdrehen"
        self.rev.bind("<ButtonPress-1>", self.reverse_func)
        self.rev.pack(side = "right")

    def reverse_func(self, event):
        the_text = self.some_text.get()
        reverse_text = the_text[::-1]
        self.some_text.set(reverse_text)

tk_window = tkinter.Tk()
app = MyFirstApp(tk_window)
app.mainloop()
```



7.2.2 Beispiel 2 - Mouseposition ausgeben

In diesem Beispiel verwenden zwei `Labels` um die Mouseposition auszugeben. Auf dem ersten Label wird kein Text sondern nur ein grüner Hintergrund angezeigt und es soll ein Motion Event-Listener auf das Label gebunden werden. In dem zweiten Label soll die x/y Position des Mouse-Zeigers im ersten Label angezeigt werden.

```
import tkinter

class MyFirstApp(tkinter.Frame):
    def __init__(self, master = None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.create_widgets()
        self.create_bindings()

    def create_widgets(self):
        self.motion_label = tkinter.Label(self,
            height = 12, width = 24, bg = "green")
        self.motion_label.pack()
        self.pos_label = tkinter.Label(self)
        self.pos_label["text"] = "Mouse auf Position ( , )"
        self.pos_label.pack(side = "bottom")


    def create_bindings(self):
        self.motion_label.bind("<Motion>", self.motion_event)

    def motion_event(self, event):
        self.pos_label["text"] = ("Mouse auf Position",
            "({},{})".format(event.x, event.y))


tk_window = tkinter.Tk()
tk_window.title("Motion Event")
app = MyFirstApp(tk_window)
app.mainloop()
```



8 Kurzer Überblick über einige ausgewählte Steuerelemente

Dies ist nur ein kurzer Überblick über einige verfügbaren Steuerelemente. Wir werden sie noch genauer in späteren Beispielen betrachten.

Widget	Bedeutung
`Widget`	die Basisklasse aller Steuerelemente
`Button`	eine Schaltfläche
`Canvas`	eine Zeichenfläche
`Checkbutton`	kann aktiv oder deaktiv sein
`Entry`	ein einzeiliges Eingabefeld
`Label`	für Beschriftungen
`LabelFrame`	für beschriftete Rahmen
`Listbox`	eine Liste von Einträgen
`Menu`	ein Kontextmenü
`Menubutton`	Button der ein Menü öffnet
`OptionMenu`	Schaltfläche die eine Auswahlliste anzeigt wenn sie angeklickt wird
`Radiobutton`	kann aktiv oder deaktiviert sein; Innerhalb einer Gruppe kann nur ein Radiobutton aktiv sein!
`Scrollbar`	eine Leiste die das Scrollen übergroßer Widgets ermöglicht
`Spinbox`	zum Einstellen eines Zahlenwertes
`Text`	ein mehrzeiliges Eingabefeld

Widgets können beim Instanziieren konfiguriert werden. Dazu können dem jeweiligen Konstruktor die gewünschten Einstellungen in Form von Schlüsselwortparametern übergeben werden z.B.:
`frame = tkinter.Frame(width=200, height=200)`
Alternativ können Widgets auch via Konfigurationsschlüssel verändert werden. Der Zugriff läuft wie bei einem Dictionary über das Schüsselwort z.B:
`frame["width"] = 200`
`frame["height"] = 100`
Viele Widgets verfügen über die Schlüssel `width` und `height` für die Breite bzw. Höhe des Widgets, `padx` und `pady` für das horizontale bzw. vertikale Padding, `state` für den Zustand des Widgets sowie `foreground` und `background` für Vordergrund- bzw. Hintergrundfarbe.
Welche Schlüssel für welches Widget zur Verfügung stehen geht über dieses Tutorial hinaus, bei bedarf bitte eine Internetrecherche starten.

8.0.1 Steuervariablen

Einige Steuerelemente (Widgets), wie beispielsweise das Eingabeelement (entry widget), Radio-Buttons und andere, können direkt unter Benutzung von speziellen Optionen mit Variablen der Anwendung verknüpft werden.
Die Optionen sind: variable, textvariable, onvalue, offvalue und value. Die Verknüpfung funktioniert in beide Richtungen: falls sich der Wert einer solchen Variablen aus welchen Gründen auch immer ändert, wird auch automatisch das mit der Variablen verknüpfte Widget auf den neuen Wert angepasst.
So entsteht mit einer gemeinsamen **Steuerelementvariable** das Verhalten eines Großelternradios: Wird eine Stationstaste z.B. für den Sender Ö1 gedrückt, springen automatisch (beim Radio noch mechanisch) die anderen Stationsknöpfe heraus.

Diese speziellen Tkinter-Kontrollvariablen werden prinzipiell wie normale Variablen benutzt. Man kann einem Widget oder genauer einer Option, die eine Tkinter-Variable erwartet, **keine normale Python-Variable übergeben**. Die einzigen Variablen, die zulässig sind, stellen die Variablen dar, die aus speziellen Klasse Variable hervorgehen.
Sie werden wie folgt deklariert:

```
x = StringVar() # Enthält eine Variable vom Typ string; default value ""
x = IntVar() # Enthält eine Variable vom Typ integer; default value 0
x = DoubleVar() # Enthält eine Variable vom Typ float; default value 0.0
x = BooleanVar() # Enthält eine Variable vom Typ boolean, returns 0 for  False and 1 for True
```

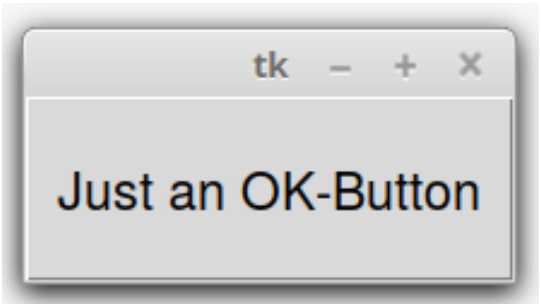
Um den aktuellen Wert einer solchen Variablen zu lesen, benutzt man die Methode `get()`. Die Zuweisung eines neuen Wertes erfolgt mittels der Methode `set()`.

Anwendungsbeispiele zu den **Steuerelementvariablen** findet man im Kapitel 8.2 auf Seite 15 und Kapitel 8.3 auf Seite 16.

8.1 Button

Was ein Button ist, muss an dieser Stelle hoffentlich nicht mehr erklärt werden ;-) Wichtig ist, nach dem Instanzieren der Button-Klasse die Optionen `text` und `command` zu setzen, über die die Beschriftung des Buttons und die Handler-Funktion festgelegt werden. Die Handler-Funktion wird vom Tk-Framework gerufen, wenn der Benutzer auf den Button ge- klickt hat. Da Buttons meist mit der linken Mousetaste gedrückt werden ist ein dezidiertes Eventhandling bei Buttons meist nicht notwendig.
Im folgenden Beispiel verwenden wir noch `font` um die Schriftart und Größe zu ändern:

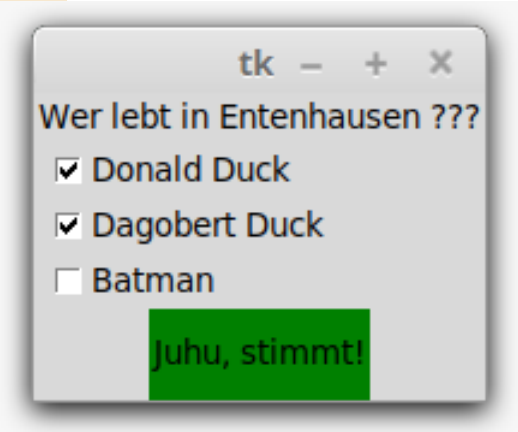
```
from tkinter import *
from tkinter import font as tkFont

class MyApp(Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()
        self.ok = Button(self)
        self.ok["text"] = "Just an OK-Button"
        self.ok["command"] = self.got_clicked
        self.ok["height"] = 2
        self.ok["font"] = tkFont.Font(size=16,
                        family='Helvetica')
        self.ok.pack()

    def got_clicked(self):
        print("I am okay!")

tk_window = Tk()
app = MyApp(tk_window)
app.mainloop()
```



8.2 Checkbox oder Checkbutton

Der Checkbutton, auch Checkbox genannt funktioniert ähnlich wie eine Button. Zusätzlich muss aber eine **Steuerelementvariable** (siehe Kapitel 8 auf Seite 13) für den aktuellen Status des Checkbuttons angelegt werden:

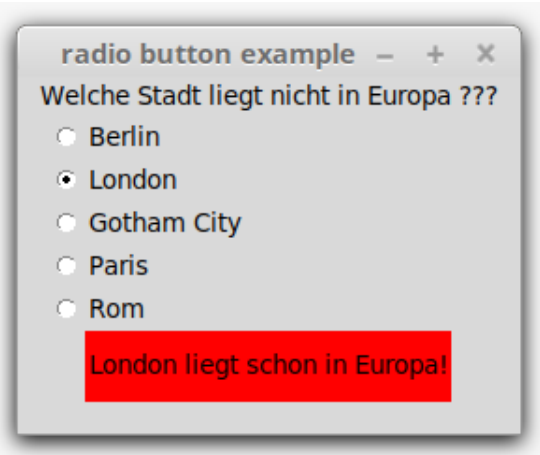
```
import tkinter

class MyApp(tkinter.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()
        self.l1 = tkinter.Label(self)
        self.l1["text"] = "Wer lebt in Entenhausen ???"
        self.l1.pack()
        self.names = ("Donald Duck", "Dagobert Duck", "Batman")
        self.checks = []
        self.vars = []
        for name in self.names:
            var = tkinter.BooleanVar()
            var.set(False)
            check = tkinter.Checkbutton(self)
            check["text"] = name
            check["variable"] = var
            check["command"] = self.handler
            check.pack(anchor="w")
            self.checks.append(check)
            self.vars.append(var)
        self.l2 = tkinter.Label(self)
        self.l2["text"] = "So stimmt es noch nicht!"
        self.l2["height"] = 2
        self.l2["background"] = "red"
        self.l2.pack()

    def handler(self):
        if (self.vars[0].get() == True and
           self.vars[1].get() == True and
           self.vars[2].get() == False):
            self.l2.config(text="Juhu, stimmt!")
            self.l2.config(background="green")
        else:
            self.l2.config(text="Stimmt so leider nicht!")
            self.l2.config(background="red")

tk_window = tkinter.Tk()
app = MyApp(tk_window)
app.mainloop()
```



8.3 Radiobutton

Ein *Radiobutton* ist wie ein Checkbutton ein Steuerelement, das durch einen Klick des Benutzers aktiviert oder deaktiviert werden kann. Das Besondere am Radiobutton ist, dass man eine Gruppe von Radiobuttons definieren kann, innerhalb derer immer genau einer der Radiobuttons aktiviert ist. Dabei bilden die Radiobuttons eine Gruppe, die sich dieselbe **Steuerelementvariable** (siehe Kapitel 8 auf Seite 13) teilt.

```
import tkinter

class MyApp(tkinter.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()
        self.l1 = tkinter.Label(self)
        self.l1["text"] = "Welche Stadt liegt nicht in Europa ???"
        self.l1.pack()
        self.auswahl = ["Berlin", "London", "Gotham City",
            "Paris", "Rom"]
        self.stadt = tkinter.StringVar()
        self.stadt.set("Berlin")
        for a in self.auswahl:
            b = tkinter.Radiobutton(self)
            b["text"] = a
            b["value"] = a
            b["variable"] = self.stadt
            b["command"] = self.handler
            b.pack(anchor="w")
        self.l2 = tkinter.Label(self)
        self.l2["text"] = "{} liegt schon in Europa!".format(self.stadt.get())
        self.l2["height"] = 2
        self.l2["background"] = "red"
        self.l2.pack()

    def handler(self):
        if self.stadt.get() == "Gotham City":
            self.l2["text"] = "Stimmt !!!"
            self.l2["background"] = "green"
        else:
            self.l2["text"] = "{} liegt schon in Europa!".format(self.stadt.get())
            self.l2["background"] = "red"

tk_window = tkinter.Tk()
tk_window.title("radio button example")
app = MyApp(tk_window)
app.mainloop()
```

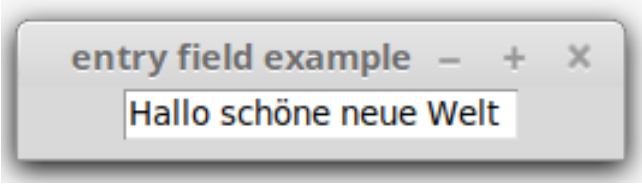


8.4 Entry

Bei einem Entry-Widget handelt es sich um ein einzeiliges Eingabefeld, in das der Benutzer beliebigen Text schreiben kann. Der folgende Beispiel-Code erzeugt ein Entry-Widget und schreibt einen Text hinein.
ACHTUNG: Diesmal wird das Event-Handling mit einer `.bind()` Methode selbst angelegt, weil das Entry-Widget kein `command`-Feld wie die Buttons hat.

```
import tkinter

class MyApp(tkinter.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()
        self.entryVar = tkinter.StringVar()
        self.entryVar.set("Hallo schöne neue Welt")
        self.entry = tkinter.Entry(self)
        self.entry["textvariable"] = self.entryVar
        self.entry.pack()
        self.entry.bind("<Return>", self.handler)

    def handler(self, event):
        print(self.entryVar.get())

tk_window = tkinter.Tk()
tk_window.title("entry field example")
app = MyApp(tk_window)
app.mainloop()
```

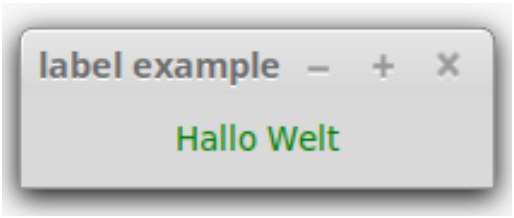
Es werden ein Entry -Widget und eine Steuerelementvariable instanziiert. Dann wird die Steuerelementvariable auf einen Wert gesetzt und mit dem Eingabefeld verbunden. Nachdem das Eingabefeld dem Packer übergeben wurde, verbinden wir noch das Event Return , das beim Drücken der Enter-Taste im Eingabefeld ausgelöst wird, mit einer Handler-Funktion, die den aktuellen Inhalt des Eingabefeldes ausgibt.



8.5 Label

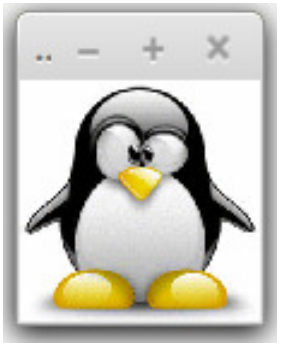
Ein Label-Widget ist ein sehr einfaches Widget, dessen einzige Aufgabe es ist, einen Text auf der grafischen Oberfläche anzuzeigen:

```
import tkinter

class MyApp(tkinter.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()
        self.label = tkinter.Label(self)
        self.label["text"] = "Hallo Welt"
        self.label["height"] = 2
        self.label["width"] = 24
        self.label["fg"] = "green"
        self.label.pack()

tk_window = tkinter.Tk()
tk_window.title("label example")
app = MyApp(tk_window)
app.mainloop()
```



Anstatt die Option text zu verwenden, wäre es an dieser Stelle auch möglich gewesen, über die Option textvariable eine Steuerelementvariable zu definieren und diese mit dem gewünschten Text zu beschreiben.
Achtung: `height` und `widht` beziehen sich bei der Anzeige von Text auf Text-Einheiten und bei der Anzeige von Bildern auf Pixel.
Noch ein Beispiel mit einem Bild:

```
from tkinter import *

class MyApp(Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()
        photo = PhotoImage(file="tux.gif")
        self.label = Label(self, image = photo)
        #keep a reference or problem with garbage
        #collector!
        self.label.image = photo
        self.label.pack()

tk_window = Tk()
tk_window.title("label example")
app = MyApp(tk_window)
app.mainloop()
```



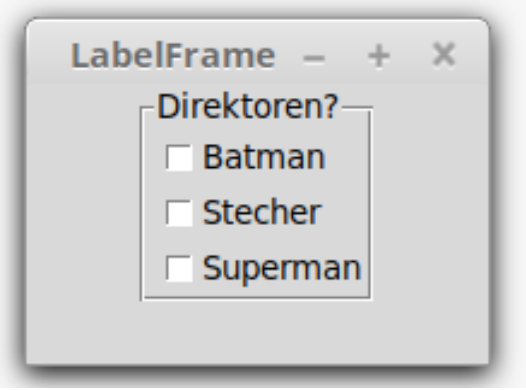
Zur Verwaltung einer Pixelgrafik verwendet man ein PhotoImage-Objekt. Ein bereits in einer Datei abgespeichertes Bild kann direkt bei der Erzeugung des PhotoImage-Objekts an dieses Objekt gebunden werden. Hierzu muss dem file-Attribut der Dateibeizeichner (evtl. mit zusätzlicher Pfadangabe) als Wert zugewiesen werden. Beachte, dass die Bilddatei im Format GIF, PPM oder PGM vorliegen muss.
Wenn man andere File-Formate verarbeiten möchte muss man auf die *Python Imaging Library (PIL)* zurückgreifen. Beispielcode-Segment dafür:

```
from PIL import Image, ImageTk

image = Image.open("tux.jpg")
photo = ImageTk.PhotoImage(image)
```

8.6 LabelFrame

Ein LabelFrame-Widget ist eine spezielle Form des Frame -Widgets und dient zur Gruppierung von Steuerelementen. Das LabelFrame -Widget zeichnet einen beschrifteten Rahmen um die ihm untergeordneten Widgets.

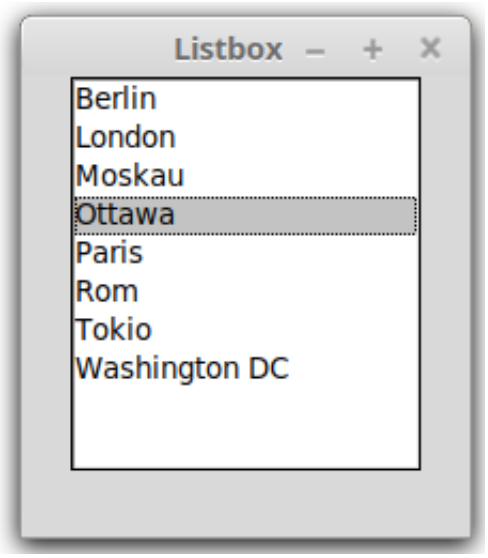
```
from tkinter import *

class MyApp(Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()
        self.names = ("Batman", "Stecher", "Superman")
        self.group = LabelFrame(self)
        self.group["text"] = "Direktoren?"
        self.group.pack()
        self.checks = []
        self.vars = []
        for name in self.names:
            var = BooleanVar()
            var.set(False)
            check = Checkbutton(self.group)
            check["text"] = name
            check["command"] = self.handler
            check["variable"] = var
            check.pack(anchor="w")
            self.checks.append(check)
            self.vars.append(var)

    def handler(self):
        if self.vars[1].get() == True:
            print("Direktor got selected!")

tk_window = Tk()
tk_window.title("LabelFrame")
app = MyApp(tk_window)
app.mainloop()
```



8.7 Listbox

Bei einer Listbox handelt es sich um ein Steuerelement, das eine Liste von Einträgen darstellt. Je nach Anwendung darf der Benutzer einen oder mehrere Einträge aus- wählen oder modifizieren. Im einfachsten Fall kann eine Listbox folgendermaßen erzeugt werden:

```
from tkinter import *

class MyApp(Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()
        self.eintraege = ["Berlin", "London", "Moskau", "
        "Paris", "Rom", "Tokio", "Washington DC"]
        self.lb = Listbox(master)
        self.lb.insert("end", *self.eintraege)
        self.lb.pack()

tk_window = Tk()
tk_window.title("Listbox")
app = MyApp(tk_window)
app.mainloop()
```



Zunächst legen wir die Liste self.eintraege an, die die Einträge enthält, die wir später in die Listbox schreiben möchten. Dann wird eine Instanz der Klasse tkinter.Listbox erzeugt und gepackt. Schließlich rufen wir für jeden gewünschten Eintrag die Methode insert der Listbox auf, die den jeweiligen Eintrag ans Ende der Listbox anhängt.

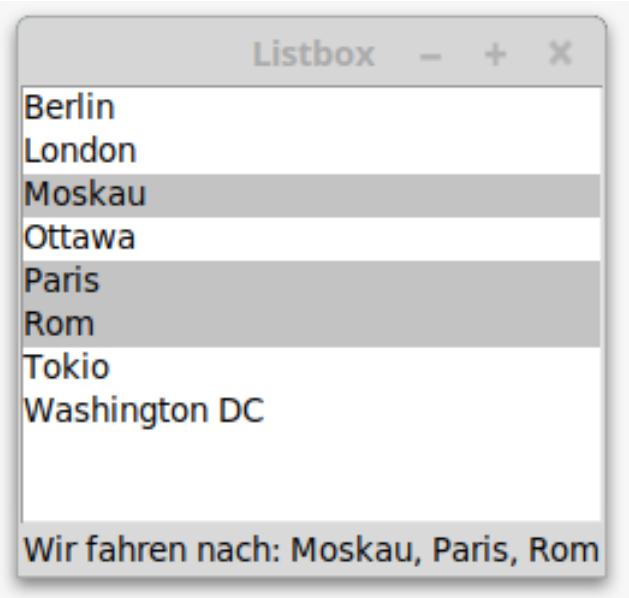
Die Einträge einer Listbox werden von 0 beginnend durchnummeriert. Über diesen Index kann auf die Einträge zugegriffen werden. Dazu definiert die Klasse Listbox eine Reihe von Methoden, die in der unten stehenden Tabelle zusammengefasst werden. Einige der Methoden bekommen dabei einen Index *first* und einen optionalen Index *last* übergeben. Wenn *last* angegeben wird, bezieht sich die Methode sinngemäß auf alle Einträge mit einem Index zwischen *first* und *last* . Wird *last* nicht angegeben, beziehen sie sich ausschließlich auf das Element mit dem Index *first*. Anstelle konkreter Indizes kann für *first* und *last* auch der String ënd"übergeben werden.

Methode	**Bedeutung**
`curselection()`	Gibt eine Liste mit den Indizes der aktuell ausgewählten Einträge zurück.
`delete(first, [last])`	Löscht einen bzw. mehrere Einträge.
`get(first, [last])`	Gibt einen bzw. mehrere Einträge als String zurück.
`insert(index, [*elements])`	Fügt die Elemente `elements` an der Position `index` in die Listbox ein.
`selection_clear(first, [last])`	Hebt eine eventuelle Auswahl eines bzw. mehrerer Einträge auf.
`selection_includes(index)`	Gibt an, ob ein Eintrag ausgewählt ist.
`selection_set(first, [last])`	Wählt ein bzw. mehrere Elemente aus.
`size()`	Gibt die Anzahl der Einträge zurück.

Das erste Beispielprogramm zur Listbox war statisch. Der Benutzer konnte zwar einen Eintrag der Listbox auswählen, doch passiert ist daraufhin nichts. Das folgende Beispielprogramm zeigt, wie man auf eine Änderung der Benutzerauswahl reagieren kann.

```
from tkinter import *

class MyApp(Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()
        self.eintraege = ["Berlin", "London", "Moskau", "Ottawa",
        "Paris", "Rom", "Tokio", "Washington DC"]
        self.lb = Listbox(self)
        self.lb.pack(fill="both", expand="true")
        self.lb["selectmode"] = "extended"
        self.lb.insert("end", *self.eintraege)
        self.lb.bind("<<ListboxSelect>>", self.selectionChanged)
        self.lb.selection_set(0)
        self.label = Label(self)
        self.label.pack()
        self.selectionChanged(None)

    def selectionChanged(self, event):
        self.label["text"] = "Wir fahren nach: " + ", ".join(
            (self.lb.get(i) for i in self.lb.curselection()))

tk_window = Tk()
tk_window.title("Listbox")
app = MyApp(tk_window)
app.mainloop()
```

Der Benutzer soll eine beliebige Menge von Städten auswählen können. Dieses Verhalten entspricht dem Wert `extended` des Konfigurationsschlüssels `selectmode`. Andere mögliche Werte sind single , browse und multiple.
Danach verbinden wir eine Handler-Methode mit dem sogenannten virtuellen Event `<<ListboxSelect>>`. Ein virtuelles Event ist ein spezielles Event, das nur mit einem be- stimmten Widget-Typ verwendet werden kann. Das `<<ListboxSelect>>`-Event wird immer dann gerufen, wenn der Benutzer die Auswahl in der Listbox verändert hat. Dann wird das erste Element der Listbox als einziges ausgewählt und ein Label-Widget erzeugt. Zum Schluss wird die Handler-Methode `selectionChanged` aufgerufen, um das Label-Widget mit einem sinnvollen Text zu versehen.



8.8 Menu

Bei einer komplexeren grafischen Benutzeroberfläche befindet sich direkt unter der Titelleiste eines Dialogs häufig eine Menüleiste, die mehrere Menüs enthält. Ein Menü ist eine Schaltfläche, über die der Benutzer eine Liste weiterer Kommandos erreichen kann.

```
from tkinter import *

class MyApp(Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()
        self.menuBar = Menu(master)
        master.config(menu=self.menuBar)
        self.fillMenuBar()

    def fillMenuBar(self):
        self.menuFile = Menu(self.menuBar, tearoff=False)
        self.menuFile.add_command(label="Öffnen", command=self.handler)
        self.menuFile.add_command(label="Speichern", command=self.handler)
        self.menuFile.add_command(label="Speichern unter",
        command=self.handler)
        self.menuFile.add_separator()
        self.menuFile.add_command(label="Beenden", command=self.quit)
        self.menuBar.add_cascade(label="Datei", menu=self.menuFile)

    def handler(self):
        print("Hallo Welt!")

tk_window = Tk()
tk_window.title("Menu")
tk_window.geometry("200x150")
app = MyApp(tk_window)
app.mainloop()
```

Die tkinter.Tk-Instanz, also das eigentlichen Window, besitzt eine Option `menu`, über die eine Menüleiste gesetzt werden kann. Das geschieht innerhalb des Konstruktors der Klasse MyApp, in dem über den Parameter master auf das Window zugegriffen werden kann. Dort wird zunächst die Menüleiste erzeugt und schließlich über die Option `menu` als Menüleiste eingetragen.

Die Methode `fillMenuBar`, die vom Konstruktor aus aufgerufen wird, hat die Aufgabe, die frisch erzeugte Menüleiste zu befüllen. Dazu wird zunächst ein Menü erzeugt, das fortan unter dem Namen `menuFile` verfügbar ist. Über den Parameter `tearoff` kann gesteuert werden, ob ein Ablösen des Menüs möglich sein soll. Dieses Verhalten ist bei den meisten Desktop-Umgebungen unpassend und wurde deshalb nicht zugelassen.

Danach werden dem Menü über die Methode `add_command` Menüpunkte hinzugefügt. Diese erhalten eine Beschriftung (`label`) und eine Handler-Funktion (`command`), die analog zur Handler-Funktion eines Buttons aufgerufen wird, wenn der Benutzer den jeweiligen Menüpunkt angewählt hat. In diesem Beispiel wird dann die Methode `handler` aufgerufen, die durch Ausgabe eines Textes demonstriert, dass das Beispielprogramm funktioniert. Einzig beim Menüpunkt *Beenden* wird `self.quit` als Handler-Methode eingetragen, um die Anwendung zu beenden.

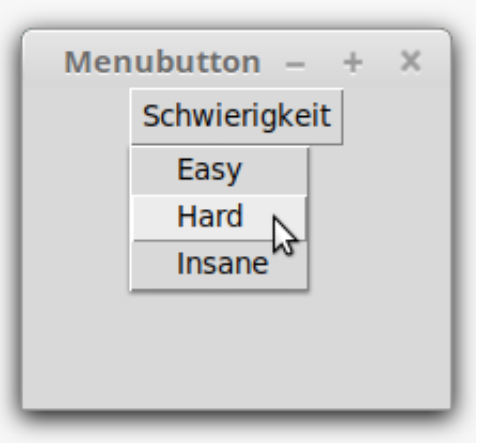
Über die Methode `add_separator` kann eine Trennlinie ins Menü eingefügt werden, um thematisch zusammengehörende Menüpunkte auch optisch zu gruppieren.
Schließlich wird über die Methode `add_cascade` der Menüleiste das neue Menü unter dem Titel *Datei* hinzugefügt.

Eine Menu-Instanz verfüght noch über die Methoden `add_check-button` und `add_radiobutton`. Diese beiden Methoden erlauben es, Radiobuttons und Checkbuttons in einem Menü zu verwenden. Die Optionen, die die Radio- bzw. Checkbuttons näher spezifizieren, werden den Methoden als Schlüsselwortparame- ter übergeben.

8.9 Menubutton

Bei einem Menubutton handelt es sich um eine Schaltfläche, die ein Menü anzeigt, wenn der Benutzer sie anklickt. Im folgenden Beispiel werden wir die oben erwähnten `radiobuttons` verwenden:

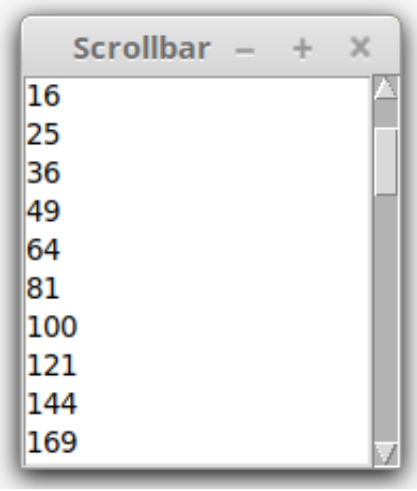
```
from tkinter import *

class MyApp(Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()
        self.mb = Menubutton(self, text="Schwierigkeit")
        self.menu = Menu(self.mb, tearoff=False)
        self.menu.add_radiobutton(label="Easy", command=self.handler)
        self.menu.add_radiobutton(label="Hard", command=self.handler)
        self.menu.add_radiobutton(label="Insane", command=self.handler)
        self.mb["menu"] = self.menu
        self.mb.pack()

    def handler(self):
        print("difficulty changed")

tk_window = Tk()
tk_window.title("Menubutton")
tk_window.geometry("200x150")
app = MyApp(tk_window)
app.mainloop()
```



8.10 Scrollbar

Es kommt häufig vor, dass der Inhalt eines Widgets, beispielsweise die Einträge einer Liste, mehr Platz benötigen, als das Widget bietet. Für solche Fälle erlauben es bestimmte Widget-Typen (Listbox-, Canvas-, Entry-, Spinbox- und Text-Widgets), eine sogenannte `Scrollbar` anzubinden. Das folgende Beispiel zeigt, wie Sie eine Scrollbar im Zusammenhang mit einer Listbox verwenden:

```
from tkinter import *

class MyApp(Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()
        self.lb = Listbox(self)
        self.lb.pack(side="left")
        self.sb = Scrollbar(self)
        self.sb.pack(fill="y", side="left")
        self.lb.insert("end", *[i*i for i in range(50)])
        self.lb["yscrollcommand"] = self.sb.set
        self.sb["command"] = self.lb.yview

tk_window = Tk()
tk_window.title("Scrollbar")
app = MyApp(tk_window)
app.mainloop()
```

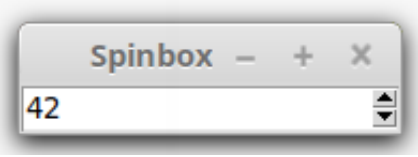


Zunächst werden eine Listbox und eine Scrollbar erzeugt und auf der Oberfläche so angeordnet, dass die Scrollbar rechts neben der Listbox steht. Danach wird die Listbox mit den Quadraten der Zahlen zwischen 0 und 50 gefüllt. Was jetzt noch fehlt, ist die Verbindung zwischen der Scrollbar und der Listbox, denn momentan haben wir nur zwei voneinander unabhängige Widgets erzeugt.
Um die Scrollbar an die Listbox zu binden, setzen wir zunächst die Option `yscrollcommand` der Listbox auf die Methode `set` der Scrollbar. Dies erlaubt ein automatisches Anpassen der Scrollbar, wenn die Einträge der Listbox über die Pfeiltasten oder das Mausrad gescrollt werden. Danach wird die `command`-Option der Scrollbar auf die Methode `yview` der Listbox gesetzt. Nun ist auch das Scrollen der Listbox mit der Scrollbar möglich.
Möchte man eine **horizontale Scrollbox** verwendet man einfach das `xscrollcommand` und die `xview`.

8.11 Spinbox

Bei einer Spinbox handelt es sich um ein Widget, in das der Benutzer eine ganze Zahl eintragen kann. Zusätzlich kann er die eingetragene Zahl über zwei Schaltflächen am Rand des Widgets nach oben oder unten korrigieren. Der folgende Code-Ausschnitt demonstriert die Verwendung einer Spinbox:

```
from tkinter import *

class MyApp(Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()
        s = Spinbox(master)
        s["from"] = 0
        s["to"] = 100
        s.pack()

tk_window = Tk()
tk_window.title("Spinbox")
app = MyApp(tk_window)
app.mainloop()
```



Die Spinbox wird instanziiert, danach werden über die Optionen `from` und `to` die Grenzen festgelegt, in denen sich die gespeicherte Zahl bewegen darf. In diesem Beispiel darf keine Zahl größer als 100 oder kleiner als 0 eingetragen werden.

Man könnte aber auch konkrete Werte angeben, welche in der Spinbox angezeigt werden sollen. Dies ist über die Option `values` möglich:

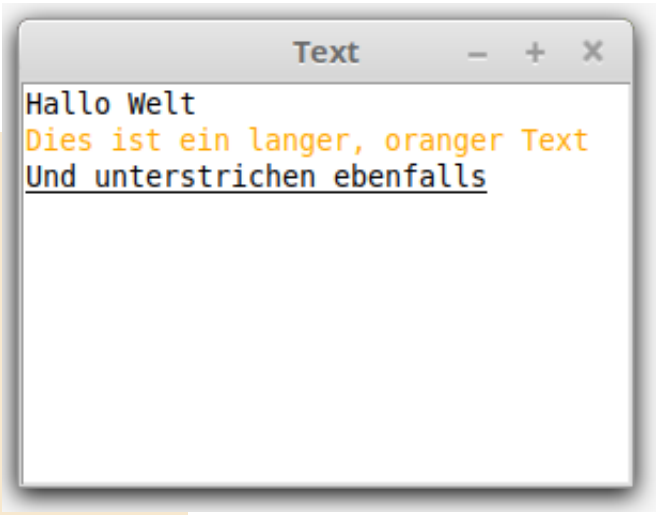
```
s = tkinter.Spinbox(master)
s["values"] = (2,3,5,7,11,13,19)
s.pack()
```

In diesem Fall kann der Benutzer eine der Primzahlen zwischen 2 und 19 in der Spinbox auswählen. Die Reihenfolge, in der die Zahlen in der Spinbox erscheinen, ist durch die Reihenfolge der Werte im Tupel gegeben.
Wenn die Werte, die die Spinbox annehmen kann, konkret angegeben werden, können dort auch andere Datentypen als `int` verwendet werden, z.B. `Strings`:
`s["values"] = ("A", "B", "C")`

8.12 Text

Bisher haben wir nur eine Möglichkeit kennengelernt, mithilfe des Entry-Widgets einzeilige Texteingaben vom Benutzer zu verlangen. Das `Text`-Widget erlaubt es, einen mehrzeiligen und formatierten Text anzuzeigen oder vom Benutzer eingeben zu lassen.

Das folgende Beispiel zeigt, wie Sie das Text -Widget dazu verwenden, formatierten Text anzuzeigen:

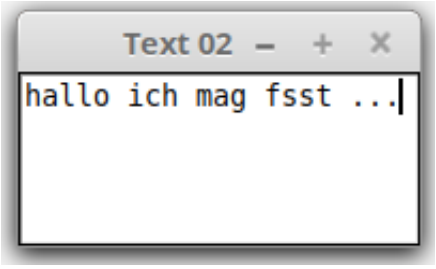



```
from tkinter import *

class MyApp(Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()
        self.text = Text(master)
        self.text.pack()
        self.text.tag_config("o",
            foreground="orange")
        self.text.tag_config("u",
            underline=True)
        self.text.insert("end",
            "Hallo Welt\n")
        self.text.insert("end",
            "Dies ist ein langer, oranger Text\n", "o")
        self.text.insert("end",
            "Und unterstrichen ebenfalls", "u")

tk_window = Tk()
tk_window.title("Text")
app = MyApp(tk_window)
app.mainloop()
```

Zunächst wird das `Text`-Widget instanziiert und gepackt. Danach definieren wir sogenannte Tags, die es uns später erlauben, den darzustellenden Text zu formatieren. In diesem Fall definieren wir das Tag `o` für orangefarbenen und das Tag `u` für unterstrichenen Text.
Danach fügen wir drei Textzeilen jeweils ans Ende des im Widget enthaltenen Textes an. Die erste Textzeile soll unformatiert, die zweite orangefarben und die dritte unterstrichen angezeigt werden.

Standardmäßig ist es dem Benutzer erlaubt, den im Text-Widget dargestellten Text zu verändern. Das folgende Beispiel zeigt, wie man auf eine Eingabe des Benutzers im Text-Widget reagiert und den eingegebenen Text ausliest:

```
from tkinter import *

class MyApp(Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()
        self.text = Text(master)
        self.text.pack()
        self.text.bind("<KeyRelease>", self.textChanged)
    def textChanged(self, event):
        print("Text:", self.text.get("1.0", "end"))

tk_window = Tk()
tk_window.title("Text 02")
app = MyApp(tk_window)
app.mainloop()
```



Anstelle einer Steuerelementvariablen bietet das Text-Widget die Methode get, über die man den im Widget dargestellten Text auslesen kann. Es handelt sich dabei um den reinen Text, jegliche Formatierungsanweisungen gehen beim Auslesen mittels get verloren.

Im Beispielprogramm wurde ein Eventhandler für das KeyRelease-Event eingerichtet. Dieses wird immer dann ausgelöst, wenn der Benutzer eine Taste loslässt, während das Text-Widget den Eingabefokus besitzt. Würden wir das KeyPress-Event verwenden, würde unser Eventhandler aufgerufen, bevor das vom Benutzer eingegebene Zeichen ins Text-Widget eingetragen wurde.

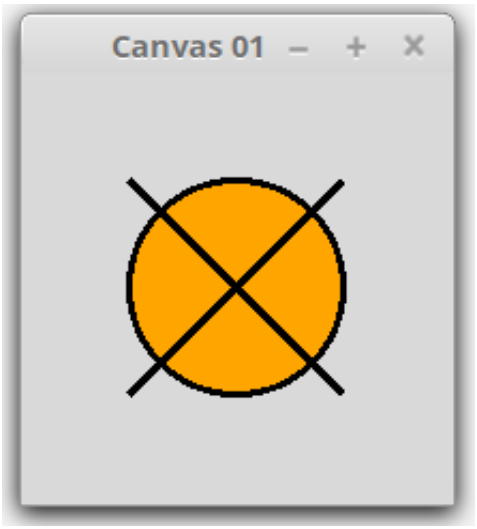
Im Eventhandler textChanged rufen wir die Methode get des Text-Widget auf. Diese Methode bekommt zwei Indizes übergeben, die angeben, welches Teilstück des Textes ausgelesen werden soll. In diesem Fall interessieren wir uns für den gesamten im Widget enthaltenen Text und geben die Indizes 1.0 und end an.
Der Index 1.0 liest sich als »erste Zeile, nulltes Zeichen«, wobei zu beachten ist, dass die Indizierung der Zeilen bei 1 und die der Spalten, also der Zeichen, bei 0 beginnt. Der Index 1.0 bezeichnet also das erste Zeichen des im Widget dargestellten Textes. Der Index end bezeichnet selbstverständlich das letzte Zeichen des im Widget enthaltenen Textes.

Es ist möglich, eine horizontale oder vertikale Scrollbar mit einem Text-Widget zu verbinden. Dies geschieht analog zum Listbox-Widget über die Optionen xscrollcommand und yscrollcommand.

8.13 Canvas - Zeichnungen

Das `Canvas`-Widget (dt. »Leinwand«) ist ein Widget, in dem beliebige Grafiken dargestellt werden können. Man kann das `Canvas`-Widget beispielsweise benutzen, um ein Diagramm zu zeichnen oder um ein Bild darzustellen. Im folgenden Beispiel-programm wird das `Canvas`-Widget verwendet, um einen Kreis und zwei Linien zu zeichnen.

```
from tkinter import *

class MyApp(Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()
        self.cv = Canvas(self,
            width=200, height=200)
        self.cv.pack()
        self.cv.create_oval(50, 50,
            150, 150, fill="orange", width=3)
        self.cv.create_line(50, 150,
            150, 50, width=3)
        self.cv.create_line(50, 50,
            150, 150, width=3)

tk_window = Tk()
tk_window.title("Canvas 01")
app = MyApp(tk_window)
app.mainloop()
```



Zunächst wird ein quadratisches `Canvas`-Widget mit einer Seitenlänge von 200 Pixel erzeugt. In dieser Zeichenfläche können wir nun über die `create`-Methoden des `Canvas`-Widgets grundlegende geometrische Formen zeichnen. In diesem Fall verwenden wir die Methoden `create_oval` und `create_line`, um den Kreis bzw. die beiden Linien zu zeichnen.

Die `create`-Methoden bekommen jeweils zwei Koordinatenpaare als erste Parameter übergeben. Diese spezifizieren die Position, an der die Form gezeichnet werden soll. Die Koordinatenangaben beziehen sich auf das lokale **Koordinatensystem** im `Canvas`-Widget, dessen **Ursprung** in der **oberen linken Ecke des Widgets liegt**. Die positive Y-Achse dieses Koordinatensystems zeigt nach unten. Das Koordinatenpaar (50, 100) bezeichnet also den Punkt, der 50 Pixel rechts und 100 Pixel unterhalb der oberen linken Ecke des `Canvas`-Widget liegt.

Die Methode `create_oval` bekommt die obere linke und die untere rechte Ecke des die Ellipse umgebenden Rechtecks übergeben. Dadurch sind Position und Form der Ellipse vollständig beschrieben. Die Methode `create_line` bekommt Start- und Ziel-punkt der Linie übergeben.
Zusätzlich können den `create`-Methoden Optionen in Form von Schlüsselwortparametern übergeben werden, die das Aussehen der gezeichneten Form spezifizieren. In diesem Fall werden die Optionen `fill` für die Füllfarbe und `width` für die Stiftdicke gesetzt.

Das `Canvas`-Widget bietet noch eine vielzahl von weiteren Methoden welche den Umfang dieses Tutorials sprengen würde. Aus diesem Grund werde an dieser Stelle nur die wichtigsten Methoden aufgezählt und auf das WWW für mehr Info (Link: Mehr Info zu Canvas) dazu verwiesen.

- `.create_arc()`: A slice out of an ellipse.
- `.create_bitmap()`: An image as a bitmap.
- `.create_image()`: A graphic image.
- `.create_line()`: One or more line segments.
- `.create_oval()`: An ellipse; use this also for drawing circles.
- `.create_polygon()`: A polygon.
- `.create_rectangle()`: A rectangle.
- `.create_text()`: Text annotation.
- `.create_window()`: A rectangular window.

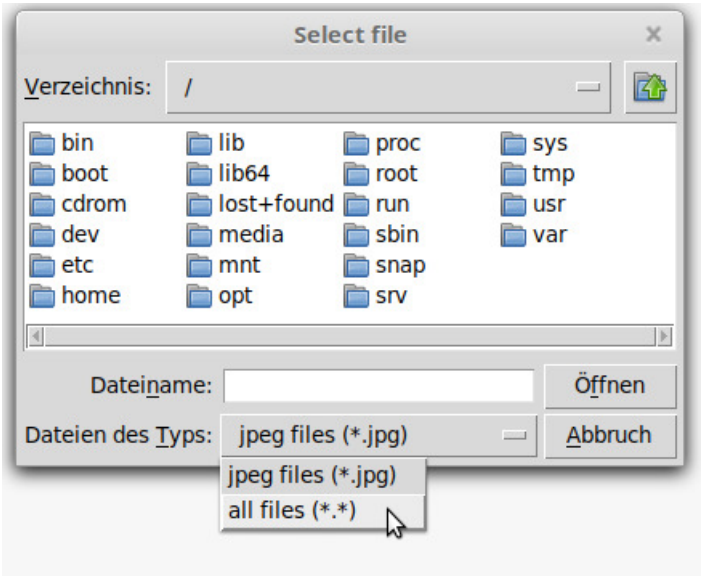
8.14 File-Dialoge

Bei der Programmierung grafischer Benutzeroberflächen gibt es Standarddialoge, die für bestimmte Fragen an den Benutzer gemacht sind, die immer wieder gestellt werden. Solche Standarddialoge haben für den Programmierer den Vorteil, dass er keinen eigenen kreieren muss. Für den Benutzer liegt der Vorteil darin, dass er sich nicht ständig mit verschiedenen grafischen Oberflächen für dieselbe Fragestellung konfrontiert sieht, sondern immer denselben vertrauten Dialog vorfindet. Auch im Tk-Framework ist es möglich, die Standarddialoge des Betriebssystems bzw. der Desktop-Umgebung zu nutzen.
Eine wichtige Klasse von Standarddialogen sind Dateidialoge, die den Benutzer dazu auffordern, Dateien oder Ordner von der Festplatte auszuwählen. Sei es, um sie in das Programm hineinzuladen oder Inhalte dorthin zu speichern. Dateidialoge werden ständig benötigt.
Das Modul `filedialog` des Pakets tkinter stellt vorgefertigte Dateidialoge bereit. In der Regel genügt ein Funktionsaufruf, um den Dialog in die eigene Anwendung zu integrieren. Im Folgenden besprechen wir die vom Modul filedialog bereitgestellten Funktionen.

8.14.1 Ein File öffnen - askopenfile()

Das Beispielprogramm öffnet einen File-Dialog und gibt den Dateinamen inklusive voller Dateipfad aus.

```
from tkinter import filedialog
from tkinter import *

root = Tk()
root.filename =  filedialog.askopenfilename(initialdir = "/",
    title = "Select file",
    filetypes = (("jpeg files","*.jpg"),("all files","*.*")))
print (root.filename)
```



8.14.2 Ein File speicher - asksaveasfile()

Identischer Code zu oben, nur verwenden wir jetzt die `asksaveasfilename()`-Methode.

```
from tkinter import filedialog
from tkinter import *

root = Tk()
root.filename =  filedialog.asksaveasfilename(initialdir = "/",
    title = "Select file",
    filetypes = (("jpeg files","*.jpg"),("all files","*.*")))
print (root.filename)
```

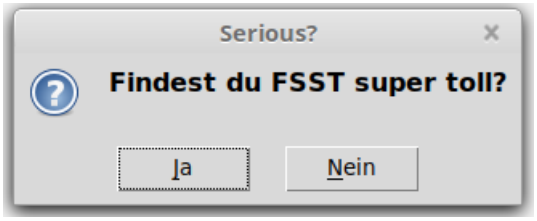
8.15 Messagebox - einfache Dialoge

Das Modul `messagebox` des Pakets tkinter ermöglicht es, durch einen einfachen Funktionsaufruf sogenannte Message Boxes anzuzeigen. Eine Message Box ist ein kleines Fenster mit einer Botschaft an den Benutzer. Sie kann dazu verwendet werden, den Benutzer über einen Fehler zu informieren oder ihm eine Frage zu stellen. Es gibt mehrere Typen von Message Boxes, beispielsweise einen, der zusätzlich zur Botschaft ein entsprechendes Icon für eine Fehlermeldung anzeigt, oder einen, der die beiden Buttons Ja und Nein anbietet, über die der Benutzer eine in der Botschaft gestellte Frage beantworten kann.
Das Modul `messagebox` stellt eine Reihe von Funktionen bereit, über die verschiedene Arten von Message Boxes erzeugt und angezeigt werden können. Diese Funktionen verfügen alle über dieselbe Schnittstelle und können wie im folgenden Beispiel verwendet werden:

```
import tkinter.messagebox

tkinter.messagebox.showwarning("Just a joke",
    "Die Installation von 'virus.exe' wird
     jetzt gestartet.")
```

Manche Dialoge lassen auch Antworten zu wie z.B. der `.askquestion()` Dialog. Er gibt je nach Userauswahl die Strings 'yes' oder 'no' zurück. Ein Beispiel:

```
import tkinter.messagebox

answer = tkinter.messagebox.askquestion("Serious?",
    "Findest du FSST super toll?")

if (answer == 'yes'):
    print("Yeah - Mitarbeitsplus!")
else:
    print("Was? Sofort Prüfungstermin ausmachen!")
```



Es werden die folgenden Dialoge bereitgestellt:

Funktion	Schaltflächen	Rückgabewert
`askokcancel`	OK/Abbrechen	True/False
`askquestion`	Ja/Nein	'yes'/'no'
`askretrycancel`	Wiederholen/Abbrechen	True/False
`askyesno`	Ja/Nein	True/False
`askyesnocancel`	Ja/Nein/Abbrechen	True/False/None
`showerror`	OK	'ok'
`showinfo`	OK	'ok'
`showwarning`	OK	'ok'

9 Mögliche Übungsbeispiele

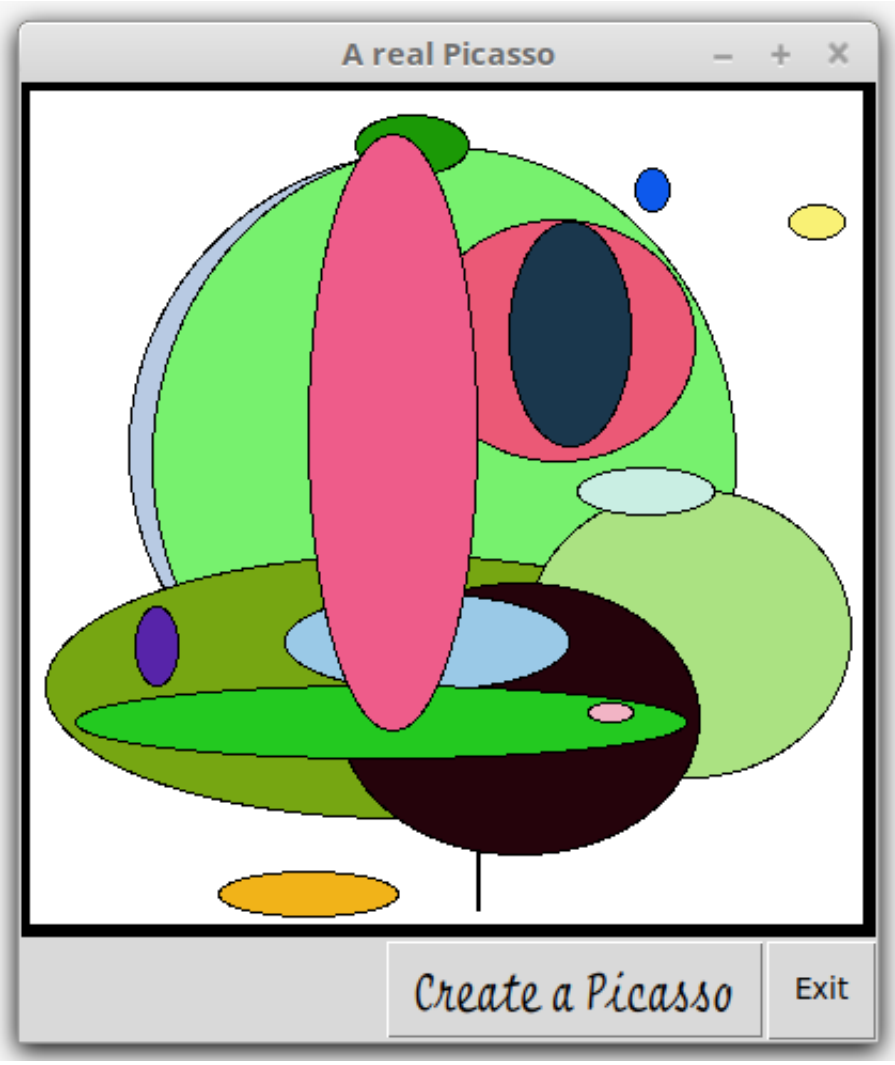
Die wichtigsten (aber bei weitem nicht alle!) Funktionen von tkinter haben wir nun kennengelernt und es ist Zeit das gelernte in die Tat umzusetzen! Im folgenden findet ihr ein paar Übungsaufgaben.

9.1 Gefälschter Picasso

- **Aufgabenstellung:**
  Erstelle eine tkinter-App die aus einem Canvas-Widget und zwei Buttons besteht. Einer der Buttons soll das Zeichnen von ca. 20 zufälligen, bunten Ellipsen im Canvas triggern. Verwende für diesen Button eine eigene Font! Die Font in meinem Beispiel heist: *UnPilgia*.
  Der zweite Button soll die App beenden.

- **Aufwand**:
  ca. 50 Zeilen Code bzw. 30-60min Arbeitszeit

- **Mögliche Lösung:**

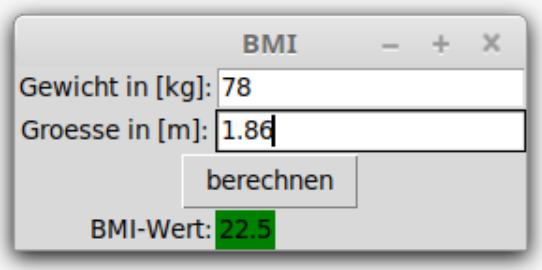


9.2 Body Mass Index

- **Aufgabenstellung:**
  Erstelle eine tkinter-App die es erlaubt den BMI zu berechnen. Die Formel für den BMI lautet: BMI=(Masse in kg)/(Groesse in m)²
  Es soll zwei Entry-Widgets zur eingabe von Zahlen geben. Drei Label-Widgets zur Beschriftung der Eingabefelder und des Ergebnis-Felds und das Ergebnis selbst soll auch in einem Label dargestellt werden. Weiters soll es einen Button für die Berechnung geben.
  Wenn das Ergebnis zwischen 19 und 24 ist, soll das Ergebniss-Label grün eingefärbt sein, und ansonsten orange.
  Arbeite mit dem Grid-Layout-Manager!

- **Aufwand**:
  ca. 50 Zeilen Code bzw. ca. 30-45 min. Arbeitszeit

- **Mögliche Lösung:**



9.3 Taschenrechner

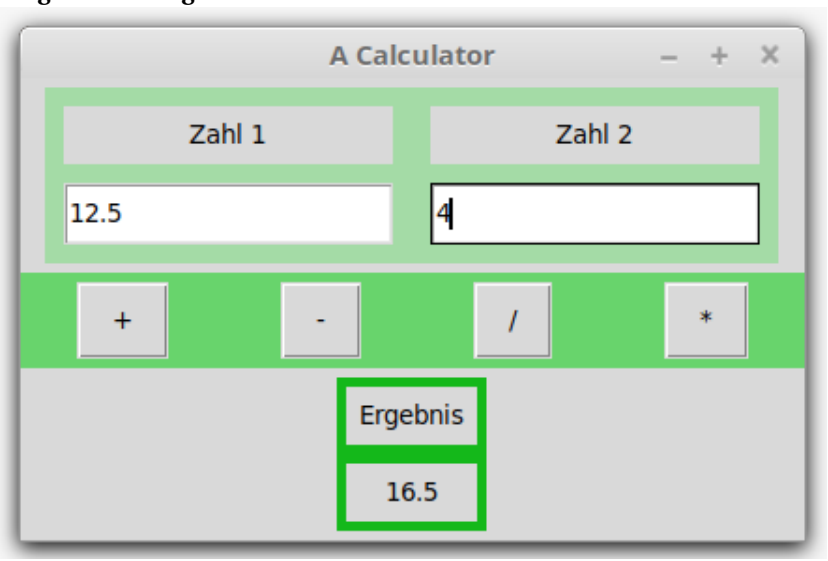
- **Aufgabenstellung:**
  Erstelle eine tkinter-App die einen kleinen Taschenrechner implementiert. Es soll zwei Entry-Widgets zur eingabe von Zahlen geben. Vier Label-Widgets zur Beschriftung der Eingabefelder und des Ergebnis-Felds und das Ergebnis selbst soll auch in einem Label dargestellt werden. Weiters soll es mindestens vier Buttons für die Rechenarten, +,-,+,/ geben. Verwende Frames um die einzelnen Elemente zu gruppieren und hierarchisch anzuordnen. **Arbeite mit dem Pack-Layout-Manager**!

  Kleiner Tipp: Das Packer-Beispiel Nr. 4 im Kapitel 6.1.2.4 auf Seite 6 ist für das Layout hilfreich!

  Noch ein Tipp: Lege auf alle vier Buttons das gleiche Event und hole dir mit `event.widget` jenes Widget (Button), welches das Event ausgelöst hat!

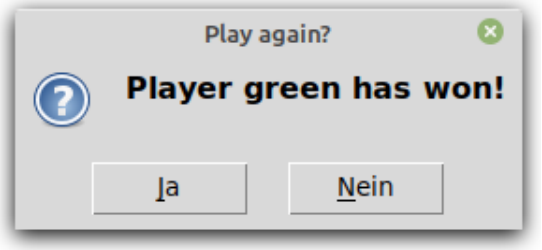
- **Aufwand**:
  ca. 90 Zeilen Code bzw. ca.60min Arbeitszeit

- **Mögliche Lösung:**



9.4 TicTacToe

- **Aufgabenstellung:**
  Erstelle eine tkinter-App die das Spiel TicTacToe implementiert. Bei Unentschieden oder einem Sieg eines Spieler soll mit Hilfe eines Pop-Up-Fenstern gefragt werden ob nochmals gespielt werden soll oder das Spiel beendet werden soll.
- **Aufwand**:
  ca. 130 Zeilen Code bzw. ca.90min Arbeitszeit
- **Mögliche Lösung:**



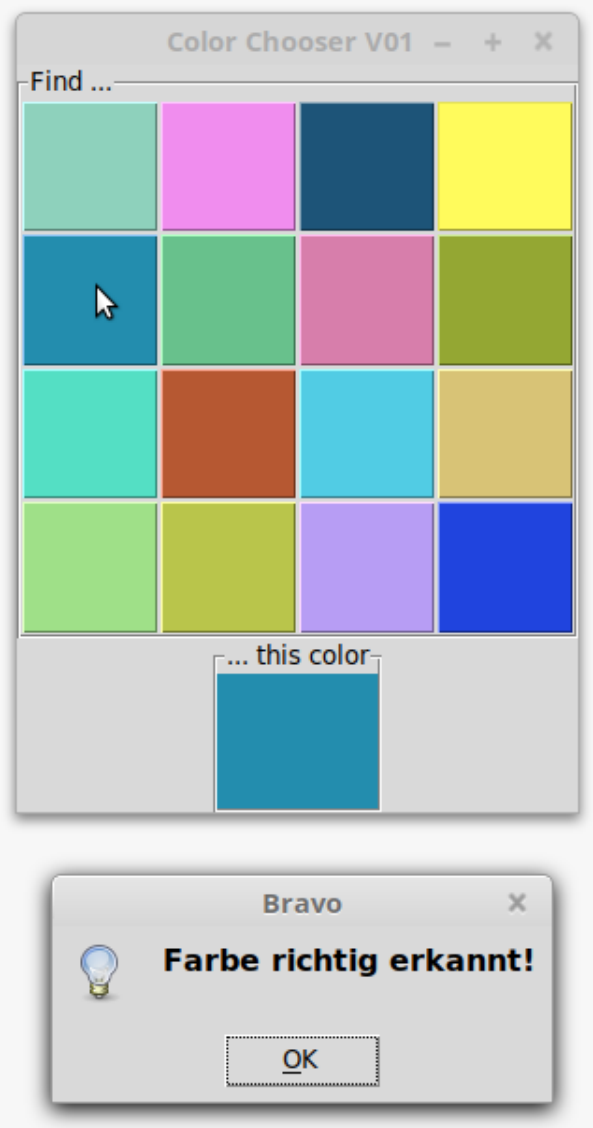
9.5 Color-Finder Verion 01

- **Aufgabenstellung:**
  Ein Spiel um Farben richtig zu erkennen. In Version 01 ist es sehr simple. 16 Buttons mit Zufallsfarben liegen in einem grid. Darunter ist ein Label mit der gleichen Farbe wie ein Button. Wenn man auf den richtigen Button drueckt, werden neue Farben verteilt und man bekommt eine pop-up Benachrichtigung.

  Kleiner Tipp: Das Grid-Beispiel Nr. 4 im Kapitel 6.2.1.3 auf Seite 9 ist für das Layout hilfreich!

- **Aufwand**:
  ca. 60 Zeilen Code bzw. ca.60min Arbeitszeit

- **Mögliche Lösung:**



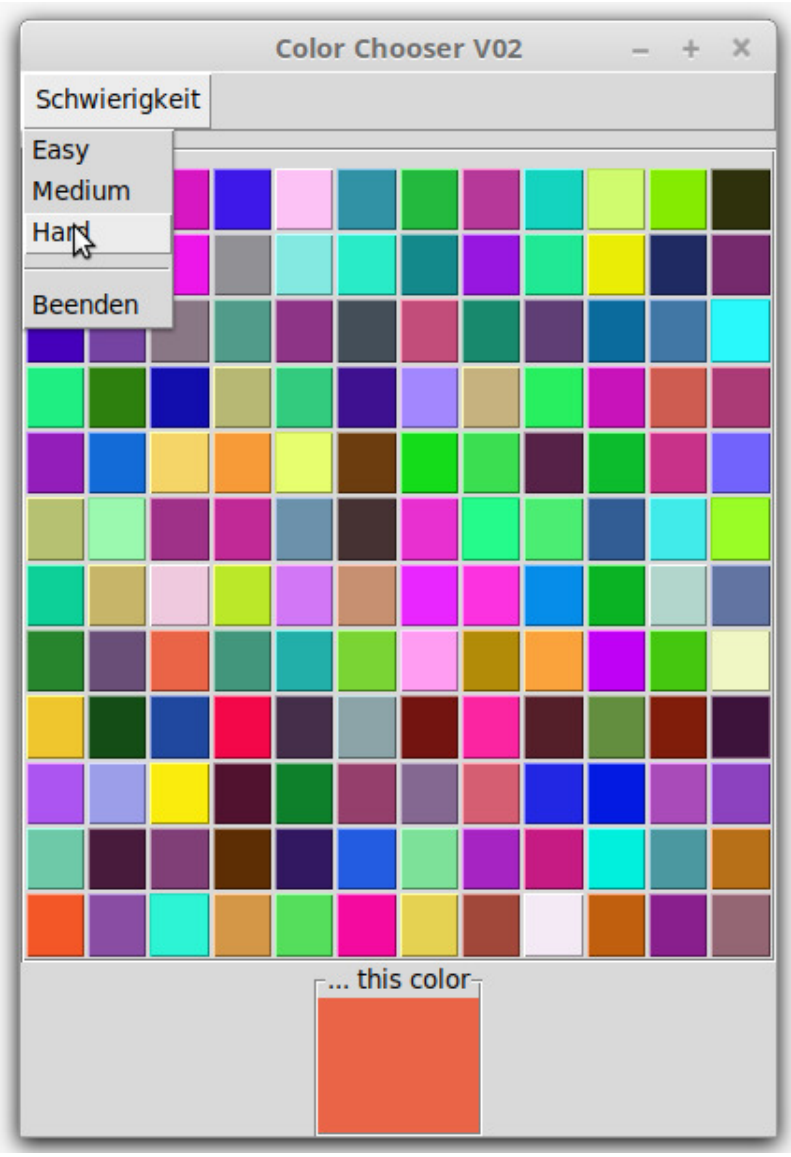
9.6 Color-Finder Verion 02

- **Aufgabenstellung:**
  Wie Cholor Chooser v01 nur erweitert um ein Menu um die Schwierigkeit zu ändern.
  Es soll drei Einträge geben: easy, medium und hard.
  easy entspricht einem 4x4 grid, medium einem 8x8 grid und hard einem 12x12 grid.

  Kleiner Tipp: Das Grid-Beispiel Nr. 4 im Kapitel 6.2.1.3 auf Seite 9 ist für das Layout hilfreich!
  Und das Beispiel im Kapitel 8.8 auf Seite 22 hilft beim Erstellen des Menus.

- **Aufwand**:
  ca. 90 Zeilen Code bzw. ca.30min Arbeitszeit zusätzlich zu Version 01

- **Mögliche Lösung:**



9.7 Color-Finder Verion 03

- **Aufgabenstellung:**
  Und nun zum krönenden Abschluss: Erweitere Color Chooser v02 noch um eine Anzeige wie viele Farben der Spieler schon gefunden hat und wie viele Clicks er dafür gebraucht hat. Bereche daraus die Genauigkeit in Prozent, mit der der Spieler die Farben findet. Also wenn er mit jedem Click gleich die richtige Farbe findet hätte er eine Genauigkeit von 100%.
  Weiters soll das Menu erweitert werden um Save-Games anzulegen und wieder zu lagen. Verwende dazu die File-Chooser-Dialoge im Kapitel 8.14 auf Seite 29.

- **Aufwand**:
  ca. 150 Zeilen Code bzw. ca.60min Arbeitszeit zusätzlich zu Version 02

- **Mögliche Lösung:**