# SWGui

Eine einfache GUI-Bibliothek auf Basis von QT

Reinhard Oldenburg

19.8.2011

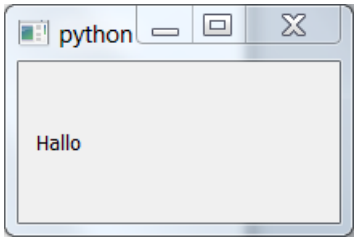
## Ein paar kleine Beispiele

SWGui ist eine Bibliothek, die das Erzeugen von Programmen mit graphischen Benutzeroberflächen möglichst einfach erlaubt. Sie setzt auf der C++-Bibliothek QT und deren Pythonanpassung PyQT auf, deren Funktionalität man vollständig nutzen kann, falls die Vereinfachungen in SWGUI zu wenig Spielraum lassen.

SWGui kann interaktiv von jedem Python-Interpreter benutzt werden. Beispiel:

```
C:\Oldenburg\Python\QT>python.exe
Python 2.7 (r27:82525, Jul  4 2010, 09:01:59) [MSC v.1500 32 bit
(Intel)] on win 32
Type "help", "copyright", "credits" or "license" for more
information.
>>> from SWGui import *
>>> SWshow()
>>> SWLabel("Hallo")
<SWGui.SWLabel object at 0x02547738>
>>>
```

Nach der ersten fett gedruckten Eingabe erscheint schon ein kleines Fenster und nach der zweiten Zeile steht darin „Hallo“:




Nach der folgenden Zeile ist ein Button hinzugekommen, der aber noch keine Funktion hat:

```
>>> b=SWButton("Ende")
```

Um auf das Klicken reagieren zu können, braucht man eine Funktion, die aufgerufen werden kann:

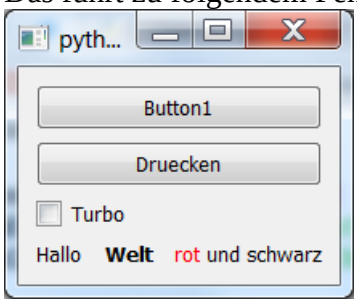
```
>>> def klickmich():
...     print "Button gedrueckt"
...     b.setText("doch noch kein Ende")
...
>>> b.setCommand(klickmich)
```

Das folgende komplette Programm (Demo1.py) zeigt ein paar weitere Möglichkeiten (je nach Art der Ausführung braucht man einen weiteren Befehl `SWstart()` am Ende – siehe Anhang):

```
from SWGui import *
SWshow()
def testEvent(*args): print ["Test-Event",args]
def buEvent(arg): print "Buttonklick!"
def cbComm(*args): print "Check-Box :" ,args
b1=SWButton(text="Button1",command=buEvent)
b2=SWButton(text="Druecken",command=testEvent)
```

```
cb=SWCheckBox("Turbo",command=cbComm)
SWHBox()
SWLabel("Hallo")
SWLabel("<b>Welt</b>")
SWLabel("<font color=#ff0000>rot</font> und schwarz")
```

Das führt zu folgendem Fenster.




Einige Kommentare dazu:

- Die einzelnen Elemente („Widgets“) werden ohne weiteres Zutun vertikal von oben nach unten angeordnet. Der Aufruf von SWHBox() stellt auf vertikale Anordnung um.
- Die aufzurufende Funktion kann nicht nur mit `setCommand` sondern auch direkt bei der Erzeugung eines Buttons angegeben werden.
- Texte können durch (eine Untermenge von) HTML formatiert werden.

Als erste (zumindest minimal) ernsthafte Anwendung ein kleiner Taschenrechner:

```
from SWGui import *
SWHBox()                        # Horizontal anordnen
a=SWTextField()                 # Ein Textfeld
SWLabel("+")
b=SWTextField()
ergebnisButton=SWButton("=")
c=SWTextField()
def rechne():
      A=float(a.getText())      # Hole Text aus Feld a und verwandle
      B=float(b.getText())
      C=A+B
      c.setText(str(C))         # Schreibt Ergebnis in Feld c

ergebnisButton .setCommand(rechne)
```

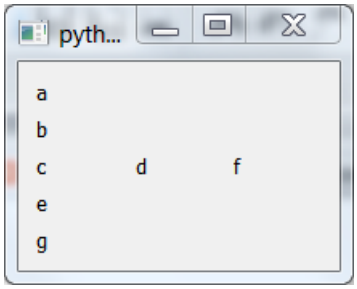
## Layout

Widgets, die mit Konstruktoren wie SWLabel, SWButton, SWTextFields u.s.w. erzeugt werden, landen der Reihe nach im SWGui-Fenster. Wenn einem das nicht gefällt, muss man das Konzept der Container verwenden. Das SWGui-Fenster ist selbst ein vertikal strukturierter Container, d.h. die einzelnen Widgets erscheinen untereinander.
Container werden erzeugt mit den beiden Konstruktoren SWHBox und SWVBox.

- Alle Widgets haben den Optionalen Parameter `container`. Damit kann der Container angegeben werden, in den das Widget gesetzt wird.
- Wird kein `container` angegeben, verwendet SWGui einen aktiven Container, der in der Variablen `TheCursor` gespeichert wird. Das ist noralerweise der letzte erzeugte Container, aber mit der Funktion `SWLastContainer()` kann man zu der davor aktuellen Fassung zurückgehen.

Das folgende langweilige Beispiel zeigt diese Möglichkeiten.

```
from SWGui import *
SWLabel("a")
SWLabel("b")
box=SWHBox()
SWLabel("c")
SWLabel("d")
SWLastContainer() # geht zum vertikalen Layout zurück
SWLabel("e")
SWLabel("f",container=box)
SWLabel("g")
```




## Event-Handler

Die meisten Elemente einer graphischen Benutzeroberfläche müssen auf bestimmte Aktionen des Benutzers reagieren können. In SWGui werden dazu bestimmte Funktionen aufgerufen. Welche Funktion beispielswiese beim Klick auf einen Button aufgerufen wird, kann man beim Erzeugen des Buttons bestimmen (optionales Argument `command=NameDerFunktion`), oder mit der Methode `setCommand`.

Häufig müssen Informationen übergeben werden. Bei einer Checkbox wird der aufgerufenen Funktion als erstes Argument True oder False übergeben, je nachdem, ob mit diesem Klick die Box gesetzt (True) wurde oder nicht.

Beachtung verdient der Fall, dass man in einem Programm mehrere Checkboxen hat.

```
cb1=SWCheckBox("Fussballfan?")
cb2=SWCheckBox("Handballfan?")
```

Um richtig reagieren zu können, muss man wissen, welche Checkbox geklickt wurde. Dazu hat man zwei Möglichkeiten.

1. Man kann verschiedene Event-Handler schreiben:
   ```
   def h1(zustand): print "Box1 ist ",zustand
   def h2(zustand): print "Box2 ist ",zustand
   cb1.setCommand(h1)
   cb2.setCommand(h2)
   ```
2. Man verwendet nur einen Handler, der dann abfragt, welche Box gemeint ist
   ```
   def h(zustand,box):
         if box==cb1: print "Box1 ist ",zustand
         if box==cb2: print "Box2 ist ",zustand
   cb1.setCommand(h)
   cb2.setCommand(h)
   ```

Argumente von Event-Handlern: Die Event-handler sind Funktionen, die beim entsprechenden Ereignis aufgerufen warden. Sie können unterschiedlich viele Argumente verarbeiten. Die meisten Event-Handler können ganz ohne Argumente

aufgerufen werden. Sie können aber auch mit Argumente bekommen, über die Information transportiert wird.

## Widgets

### Labels

Konstruktor

```
SWLabel(text="",container=None)
```

Methoden:

| | |
|---|---|
| `setText(str)` | setzt den Text |

### Buttons

Konstruktor

```
SWButton(text="",container=None, command=None)
```

Methoden:

| | |
|---|---|
| `setText(str)` | setzt den Text |
| `setCommand(cmd)` | setzt den Event-Händler. cmd muss eine Funktionsein, die entweder ein Argument übernimmt, oder eines (den Button, der gedrückt wurde) |

### Checkboxen

Konstruktor

```
SWCheckbox(text="",container=None, command=None)
```

Methoden:

| | |
|---|---|
| `isChecked()` | fragt ab |
| `setChecked(bool)` | setzt programmgesteuert |
| `setText(str)` | setzt den Text |
| `setCommand(cmd)` | setzt den Event-Händler |

### Slider (Schieberegler)

Konstruktor

```
SWSlider(orient, min, start,max,container=None, command=None)
```

| | |
|---|---|
| `orient` | “h” oder “H” für horizontal b.z.w. „v“ oder „V“ für vertikal |
| `min, max, start` | Wertebereich und Anfangsposition |

Der Eventhänder ist eine Funktion mit einem Parameter (neuer Wert) bzw. zwei Parametern (neuer Wert, Schieberegler) oder auch ganz ohne Parameter.

Methoden:

| | |
|---|---|
| `setValue(int)` | setzt Position |
| `getValue()` | fragt ab |
| `setCommand(cmd)` | setzt den Event-Händler |

### TextField

Konstruktor

```
SWTextField(container=None,text='', command=None)
```

Methoden:

| | |
|---|---|
| `setText(str)` | setzt den Text |
| `getText()` | liest den Inhalt |
| `getFloat()` | liest den Inhalt und gibt ihn als float-Zahl zurück. Wenn das nicht möglich ist, wird float(„nan“) ausgegeben (not a number) |
| `getInt()` | wie getFloat, liefert aber ganze Zahl |

| | |
|---|---|
| `setCommand(cmd)` | setzt den Event-Händler für die Enter-Taste. cmd nimmt entweder kein Argument oder ein Argument, nämlich das TextField selbst. |

## ComboBox

Konstruktor

```
SWComboBox(container=None,ItemList=[], command=None)
```

Methoden:

| | |
|---|---|
| `addItem(str)` | ergänzt ein Item |
| `removeItem(index)` | löscht Item mit übergebenem Index |
| `insertItem(ind,str)` | fügt ein Item an gegebener Stelle ein |
| `count()` | Zahl der Items |
| `currentText()` | derzeit ausgewählter Eintrag |
| `currentIndex()` | Index des ausgewählten Eintrags (beginnend mit 0) |
| `setCommand(cmd)` | setzt den Event-Händler für die Auswahl eines Eintrags. cmd muss drei Argumente nehmen können; Die Box selbst, den index und den Text des gewählten Items. |

## TextEdit

Mehrzeiliger Textbreich

Konstruktor

```
SWTextEdit(container=None,text='', command=None)
```

Methoden:

| | |
|---|---|
| `setText(str)` | setzt den Text |
| `getText()` | liest den Inhalt |
| `setCommand(cmd)` | setzt den Event-Händler für die Enter-Taste. cmd nimmt entweder kein Argument oder ein Argument, nämlich den TextEdit selbst |

## SWConsole

Python-Shell zum Einbetten ins eigene Programm. Das ist nützlich u.a. für die Fehlersuche.

Konstruktor

```
SWConsole(env={})
```

Das Argument env gibt die Umgebung an, in der gerechnet wird. Wenn Zugriff auf alle Variablen gewünscht ist, gibt man hier globals() an, also:

```
SWConsole(env=globals())
```

## SWHTMLDisplay

Darstellung von HTML-Dokumenten

Konstruktor

```
SWHTMLDisplay(command=None)
```

Methoden:

| | |
|---|---|
| `setText(str)` | setzt den Text. str ist ein HTML-Text als eine (ggf. riesige) Zeichenkette. Beispiel: H.setText('<html>….' |
| `setCommand(cmd)` | setzt den Event-Händler für den Klick auf einen Link. cmd muss zwei Argumente übernehmen: Das SWHTMLDisplay selbst und das Ziel des Links als Zeichenkette. |

## Tabelle

Eine SWTable ist ein Gitter von Zellen, in denen Zeichenketten dargestellt und eingegeben werden können.

Konstruktor

```
SWTable(Zeilenzahl,Spaltenzahl,container=None)
```

Methoden:

| | |
|---|---|
| `setClickedCommand(cmd)` | setzt den Event-Händler für Klick in Zelle |
| `setCchangedCommand(cmd)` | setzt den Event-Händler für einen neuen Eintrag |

Beide Eventhänder müssen zwei Argumente übernehmen, die Zeile und die Spalte:

```
def haendler(z,s): …
```

Methoden der Tabellen sind:

| | |
|---|---|
| `getItem(zeile,spalte)` | Holt den Text aus der Zelle |
| `setItem(zeile,spalte,text)` | Setzt den Text in der Zelle |
| `newRow(i,Text="")` | Fügt neue Zeile ein |
| `newCol(i,Text="")` | Fügt neue Spalte ein |
| `removeRow(i)` | Löscht Zeile |
| `removeColumn(i)` | Löscht Spalte |
| `setColHeaders(liste)` | legt die Spaltenbeschriftungen fest |
| `setRowHeaders(liste)` | legt die Zeilenbeschriftungen fest |
| `columnCount()` | Gibt die Zahl der Spalten |
| `rowCount()` | Gibt die Zahl der Zeilen |
| `showVerticalHeader(False)` | zeigt die Spaltenbeschriftungen nicht an |
| `showHorizontalHeader(False)` | zeigt die Zeilenbeschriftungen nicht an |

## Dialoge

Die Dialog-Funktionen öffnen kleine Fenster, in die eine Eingabe erfolgen kann. Es wird die vom Benutzer gemachte Eingabe zurückgegeben, oder **`None`**, falls der Benutzer Cancel gewählt hat.

| Dialog | Erklärung | Beispiel |
|---|---|---|
| SWWarning | Zeigt einen Text | `SWWarning("Problem! ")` |
| SWYesNoDialog | SWYesNoDialog(text) stellt eindeutige Frage. Rückgabe True oder False | `SWYesNoDialog("OK? ")` |
| SWFileDialog | Fragt nach einer Datei. Rückgabe ist der Pfad als Zeichenkette (oder None)<br>Optionale Argumente:<br>title: Überschrift<br>dir: Startverzeichnis | `SWFileDialog()`<br>`SWFileDialog("Datei? ")`<br>`SWFileDialog("Datei? ",`<br>`"c:\\Temp ")` |
| SWColorDialog | Fragt nach Farbe, liefert [R,G,B]-Liste | `SWColorDialog()` |
| SWColorDialogQ | Fragt nach Farbe, liefert QColor-Objekt | `SWColorDialogQ()` |
| SWIntDialog | Fragt nach einer ganzen Zahl<br>Optionaler parameter init | `SWIntDialog("Alter?")`<br>`SWIntDialog("Alter?",`<br>`init=18)` |
| SWFloatDialog | Fragt nach einer Kommazahl<br>Optionaler parameter init | `SWFloatDialog("Laenge?")` |
| SWTextDialog | Fragt nach einem Text | `SWTextDialog("Name?")` |

SWTextDialog ermöglicht mit zwei optionalen Parametern Tests und Umwandlungen einzubauen. Die folgende Abfrage akzeptiert nur Eingaben mit Mindestlänge 10 und gibt deren Länge zurück.

```
>>> def str10(s): return len(s)>=10
>>> print SWTextDialog("Geben Sie einen langen String ein",
                       oktest=str10,converter=len)
```

Wenn Dialoge noch stärker an an die eigenen Bedürfnisse angepast werden müssen, sollte man eigene Klassen definieren. Im Quelltext von SWGUI zeigt die Klasse SWTextDialog, wie man dazu prinzipiell vorgeht.

## Bitmap-Grafiken

SWImage ist ein Bereich im Fenster, der eine Pixelgrafik anzeigt. Es können einzelne Pixel gesetzt und gelesen werden.

- Koordinatensystem: Wie in der Computergrafik üblich sind die Koordinaten ganze Zahlen, die die Pixel-Position angebeben. x=0,y=0 ist die linke obere Ecke des Bildes, nach unten(!) wächst y, nach rechts wächst x.
- Überall wo Farben angegeben werden, kann dies in einer der folgenden Formen geschechen:
    - Ein englisches Farbwort: „white“, „black“, „red“, „green“, „blue“, “yellow”
    - Eine ganze Zahl aus dem Bereich 0..255: Graustufenhelligkeit (0=Schwarz)
    - Eine Liste mit drei Zahlen [R,G,B] Rot-, Grün und Blauanteil jeweils als Zahl von 0..255
    - Ein Farbobjekt der PyQT-Bibliotek

Konstruktor

```
SWImage(width=None,height=None,container=None, file=None,
dim=[200,200])
```

Das Bild kann enteder konstruiert warden durch Angabe einer Bilddatei oder durch Angabe der Größe (entweder mit wodth und height oder mit dims)

```
BM=SWImage(dim=[300,200])
BM2=SWImahge(file="bild1.jpg")
```

| **Methode** | **Erklärung** | **Beispiel** |
|---|---|---|
| `getWidth()`<br>`getHeight()` | Abmessung abfragen | |
| `setColor(col)` | setzt die Standardfarbe, die verwendet wird, wenn sonst nichts angegeben | `BM.setColor([255,100,100])` |
| `fill(), fill(color)` | Färbt die ganze Bitmap einfarbig | `BM.fill("white")` |
| `getPixel(x,y)` | Gibt eine [R,G,B]-Liste der Farbe an Position x,y | `BM.getPixel(100,50)` |
| `getPixelGray(x,y)` | Gibt den Helligkeitswert des Pixels an Position x,y | `BM.getPixelGray(100,50)` |
| `setPixel(x,y,vol)` | Setzt ein Pixel | `BM.setPixel(100,50, "blue")` |
| `drawLine(self,x0,y0, x1,y1,col)` | Zeichnet eine Strecke, farbe ist optional | `BM. drawLine(0,50,x100,50)` |
| `drawEllipse(x0,y0,xd,yd , col=None,filled=False)` | Zeichnet eine Ellipse | `BM.drawEllipse(50,h/2+70,20, 10,`<br>`  col="yellow",filled=True)` |
| `drawRect(x0,y0,xd,yd, col=None,filled=False)` | Zeichnet ein Rechteck | `BM.drawRect(50,70,20,10,`<br>`  col=[255,0,0],filled=True)` |
| `drawText(x,y,text, col=None)` | Zeichnet einen Text | `BM.drawText(50,70, "Hallo")` |
| | | |

| | | |
|---|---|---|
| `save(Dateiname)` | Speichert die Datei | `BM.save("bild.jpg")` |
| `update()` | Stellt sicher, dass deie angezeigte Grafik aktuell ist | `BM.update()` |
| `getTurtle()` | Holt eine Turtle | `T=BM.getTurtle()`<br>`T.forward(100)` |
| `setMousePressCommand(`<br>`  cmd)`<br>`setMouseReleaseComma(`<br>`  cmd)`<br>`setMouseDragCommand(`<br>`  cmd)`<br>`setMouseMoveCommand(`<br>`  cmd)` | Setzt die Ereignisbehandler | `def mp(bitmap,event):`<br>`  bitmap.setPixel(`<br>`  event.X,event.Y)`<br><br>`BM.setMousePressCommand(mp)` |

Kommentare zur Ereignisbehandlung

- Die Eventhandler bekommen zwei Argumente: Erstes Argument ist die bitmap selbst, das zweite ein event-Objekt von QT angereichert um die Eigenschaften X und Y, die Koorinaten des Klicks

Kommentare zur Turtlegrafik: Wenn T eine Turtle ist, gibt es folgende Befehle:

- `T.forward(100)` vorwärts
- `T.backwards(100)` rückwärts
- `T.left(45)` links um Winkel
- `T.right(45)` rechts um Winkel
- `T.setPosition(x,y)` gehe zur Position (ohne Zeichnen)
- `T.moveTo(x,y)` gehe zu
- `T.setDirection(alpha)` Drehe in Richtung alpha (gegen Rechtsachse)
- `T.penUp()` Stift hoch: Nicht zeichnen
- `T.penDown()` Stift runter
- `T.setPenColor(col)` Farbe setzen

Kommentare Zum Koordinatensystem: Zu fünf Befehlen zum Zeichnen gibt es Varianten, die nicht ein Pixelkoordinatensystem, sondern ein Weltkoordinatensystem zugrunde legen. Es sind: `setPixelW, setTextW, setCircleW, setEllipseW,setRectW`. Das Koordinatensystem ist standrdmäßig -10..10 in beide Richtungen. Diese Einstellungen können vor dem Zeichnen durch setzen des Attributs viewport geändert werden, zB mit: BM.viewport=[-100,100,-50,50]

## Objektbasierte Grafiken

Bei den Pixelbasierten Grafiken vom Typ SWImage bedeutet die Anweisung drawCircle nur, dass kreisförmig angeordnete Pixel gesetzt werden, es wird aber kein Kreis als Objekt erzeugt. Das bedeutet, dass man den Kreis anschließend nicht als Ganzes verschieben kann. Wenn er verschoben werden soll, muss man ihn an der alten Stelle die Pixel entfärben und an der neuen Stelle neu zeichnen. Das ist vor allem bei interaktiven Programmen sehr mühsam.
Objektbasierte Grafiken verhalten sich anders: Man fügt ihnen Objekte hinzu, die noch nachträglich verändert (verschoben, umgefärbt,…) werden können.

Konstruktor

```
SWGraphics(width=200,height=200, container=None)
```

Die folgenden Befehle erzeugen graphische Objekte und geben diese zurück.
Typische Verwendung ist also:

```
G=SWGraphics(400,400)
R1=G.addRectangle(200,130,40,50,color="blue")
R1.setCenter(100,150)
```

`addSWPixelImage(x,y,pixmap)` pixmap kann ein String sein, der Bilddatei angibt

```
addLine(x1,y1,x2,y2,color=None):
addRectangle(x,y,xd,yd,color=None,fillcolor=None)
addEllipse(x,y,xd,yd,color=mkQCol("black"),fillcolor=None):
addCircle(x,y,r,color=mkQCol("black"),fillcolor=None)
addText(x,y,text)
```

Für Line, Rechtecke, Kreise und Ellipsen gibt es auch die Varianten mit angehängtem W (`addRectangleW`, u.s.w) die genau wie bei den Pixelbildern sich auf ein mathematisches Koordinatensystem beziehen.

Methoden von SWGraphics:

| | |
|---|---|
| `removeItem(obj)` | Löscht ein Objekt |
| `getDim()` | Gibt die Größe in Pixeln zurück |
| `setViewport([xmin,xmax,ymin,ymax])` | gibt die Grenzen des mathematischen Koordinatensystems an |

Die wichtigsten graphischen Objekte (Bilder, Kreise, Ellipsen, Rechtecke, Linien) verfügen über folgende Methoden:

| | |
|---|---|
| `setAutoDrag(jaNein)` | erlaubt automatisches Ziehen |

```
setCenter(x,y)
scale(faktorX,factorY)
rotate(winkel)
setMousePressCommand(cmd)
setMouseReleaseCommand(cmd)
setMouseDragCommand(self,cmd)
setMouseMoveCommand(cmd)
```

Alle eventhander bekommen zwei Arguemte übergeben: Das Objekt selbst und ein Ereignisobjekt. Typsich ist also z.B.

```
def haendler(obj,event):
      print 'Hier ist es  passiert: ', [event.X, event.Y]
```

Das event-Objekt hat Felder `X,Y, XW,YW` für die Pixel- bzw Weltkoordinaten des Ereignisses. (Achtung alles sind Großbuchstaben!)

`setCollissionCommand(cmd)`
Dieser Befehl setzt den Befehl, der bei Kollision des Objektes aufgerufen wird. Immer wenn ein Objekt eine Position ändert, wird geprüft, ob es mit anderen kollidiert und falls dafür ein Befehl gesetzt wurde, wird er ausgeführt. cmd bekommt ein Argument übergeben, nämlich eine Liste der kollidierenden Objekte.

## Allgemeine Eigenschaften

| | |
|---|---|
| `SWsetWindowTitle(str)` | setzt den Titel des Fensters |
| `SWsetWindowSize(breite,hoehe)` | setzt die Größe des Fensters |

`SWshow()` zeigt das SWGui-Fenster an, SWshow(False) verbrigt es

`SWstart()` startet die Ereignisbehandlungsschleife, d.h. diese Prozedur sorgt dafür, dass das Programm nicht beendet wird und sie läuft solange, bis das SWGui Fenster geschlossen wird.

`setKeyCommand(key,cmd)` Setzt den Befehl, der beim Drücken der Taste key ausgelöst wird. cmd ist eine parameterlose Funktion. key ist eine Zeichenkette, entweder der Code der Taste, zB `"A", "a", "2"` oder eine Beschreibung einer Sondertaste: `"return","tab"`.
Die Cursortasten können aktuell nicht abgefragt werden.

`setAnyKeyCommand(cmd)` setzt einen Befehl, der bei einem beliebigen Tastendruck ausgelöst wird. cmd ist eine Funktion, die ein Argument übergeben bekommt, nämlich die Beschreibung der Taste als Zeichenkette(wie bei `setKeyCommand`). Gibt es für die Taste sogar einen mit `setKeyCommand` gesetzen Befehl, wird nur dieser ausgeführt.

# Anhang

**Installation**

Zuerst sind zu installieren:

- Python 2.6 oder 2.7
- PyQT (von riverbank)

Danach muss die Datei SWGui.py an einen Ort gespeichert werden, wo Python sie finden kann. Beispielsweise kann man sie nach c:\Python27\Lib\SitePackeges kopieren. Wenn sie nicht fest installiert werden soll, kann man alternativ dafür sorgen, dass sie immer im gleichen Verzeichnis liegt wie die Programme, die man damit schreibt.

**Hinweise zum Ausführen von Programmen**

Das Demoprogramm Demo1.py:

```
from SWGui import *
SWshow()
def testEvent(*args): print ["Test-Event",args]
def buEvent(arg): print "Buttonklick!"
def cbComm(*args): print "Check-Box :" ,args
b1=SWButton(text="Button1",command=buEvent)
b2=SWButton(text="Druecken",command=testEvent)
cb=SWCheckBox("Turbo",command=cbComm)
SWHBox()
SWLabel("Hallo")
SWLabel("<b>Welt</b>")
SWLabel("<font color=#ff0000>rot</font> und schwarz")
```

Dieses Programm kann auf folgende Arten ausgeführt werden:
Von der Kommandozeile des Betriebssystems:

```
C:\Oldenburg\Python\QT>python -i Demo1.py
```

Die Option –I sorgt dafür, dass der Python-Interpreter offen bleibt. Man kann dort interaktiv mit den Objekten interagieren, z.B. den Befehl eingeben:

```
b1.setText("Neue Beschriftung")
```

Ohne die Option –i würde das Programmfenster erstellt und sofort wieder geschlossen, weil Python vermeintlich mit allen Aktivitäten fertig ist.
Alternativ kann man als letzte Zeile in das Programm schreiben, die dann auch ohne die Option –i funktioniert:

```
SWstart()
```

Zur Ausführung in **Spyder** gibt es verschiedene Optionen (über Funktionstaste F6 einstellbar)

- „Execute in a new dedicated Python interpreter“
    - Programmzeile `SWstart()` einfügen
- IPython Konsole erzeugen (über Run/Open IPython) und Konfiguration “Execute in current Python or IPython interpreter“ wählen
    - Es ist dann möglich, interaktiv die GUI zu ändern
- Python Konsole erzeugen (über Run/Open Python) und Konfiguration “Execute in current Python or IPython interpreter“ wählen
    - Die Programmzeile `SWstart()` muss eingefügt werden. Es ist dann NICHT möglich, interaktiv die GUI zu ändern

Zur Ausführung in **IDLE** muss die Zeile `SWstart()` eingefügt werden