

# Frog

## Grafik mit Fröschen

---



**Version 2.2.5**

12. September 2024

**Grafische Lernumgebung  
für Python 3.x  
basierend auf Tkinter**

Copyright 2008 – 2024

by Marco Haase

[www.pyxo.de](http://www.pyxo.de)

# Inhalt

|   |    |
|---|----|
| 1. Einführung.....                              | 3  |
| 1.1 Was ist frog?.....                          | 3  |
| 1.2 Systemvoraussetzungen und Lizenz.....       | 4  |
| 1.3 Über diese Dokumentation.....               | 4  |
| 1.4 Hallo Frog!.....                            | 5  |
| 2. Der Pool.....                                | 6  |
| 2.1 Einen Pool erzeugen.....                    | 6  |
| 2.2 Den Pool konfigurieren.....                 | 7  |
| 2.3 Eigenschaften des Pools ermitteln.....      | 8  |
| 2.4 Operationen im Pool.....                    | 9  |
| 2.5 Das Koordinatensystem.....                  | 9  |
| 2.6 Das Farbmodell.....                         | 9  |
| 2.7 Screenshot erstellen.....                   | 11 |
| 3. Die Frösche.....                             | 13 |
| 3.1 Der Frosch als Persönlichkeit.....          | 13 |
| 3.2 Frösche in Aktion.....                      | 15 |
| 3.3 Frösche lokalisieren und positionieren..... | 17 |
| 3.4 Weitere Eigenschaften.....                  | 18 |
| 3.5 Weitere Fähigkeiten.....                    | 20 |
| 3.6 Lesen und Schreiben.....                    | 22 |
| 3.7 Laden und Speichern.....                    | 25 |
| 4. Ereignisbehandlung.....                      | 26 |
| 4.1 Buttons im Pool.....                        | 26 |
| 4.2 Lauschen auf Ereignisse.....                | 27 |
| 4.2.1 Aufbau einer Event-Sequenz.....           | 28 |
| 4.2.2 Spezialfälle und Stolpersteine.....       | 30 |
| 4.2.3 Reagieren auf Ereignisse.....             | 32 |

# 1. Einführung

## 1.1 Was ist frog?

Frog ist ein Graphikmodul zur Einbindung in Python-Programme. Entwickelt wurde es für den Einsatz im Informatikunterricht und in Programmierkursen, in denen Jugendlichen das Programmieren auf der Basis von Python vermittelt werden soll.

Frog ermöglicht es, mit Hilfe weniger Befehle einen „Frosch“ (z.B. in Gestalt eines kleinen Dreiecks auf dem Bildschirm) dazu zu bringen, durch seine Bewegung auf dem Bildschirm Spuren zu hinterlassen und so graphische Darstellungen zu erzeugen. Die Verwendung von frog setzt praktisch keine Python-Kenntnisse voraus, so dass es (auch) möglich ist, auf der Grundlage des frog-Moduls die Programmiersprache Python zu erlernen.

Ich selbst setzte das Modul im Informatikunterricht der Jgst. 8/9 ein. Einige Jahre lang habe ich vor dem Einsatz des Froschs zunächst Python auf klassische Weise vermittelt und erst nach etwa halbjährigem Programmierkurs das frog Modul ergänzt. Bis dahin gibt es allerdings manche Durststrecke für die Schüler und so bin ich inzwischen dazu übergegangen, direkt mit dem frog-Modul einzusteigen. Nach und nach werden dann die Grundlagen von Python, alle wichtigen Datentypen und Kontrollstrukturen eingeführt.

Gibt es für diesen Einsatzzweck nicht schon das turtle-Modul? Ja, ABER: Als ich für meinen Unterricht ein entsprechendes Modul brauchte (2008), war Python 2.5 die aktuelle Version und das dort mitgelieferte turtle-Modul war mehr oder weniger unbrauchbar. Das hatte auch Gregor Lingl gemerkt und daraufhin sein xturtle-Modul entwickelt, mit dem sich nahezu alles realisieren ließ, was mir wichtig erschien. Seit Python 2.6 (und damit auch in Python 3.x) ist das alte turtle-Modul durch das bisherige xturtle von Gregor Lingl ersetzt worden, so dass die Standardbibliothek nun über ein brauchbares turtle-Modul verfügt.

Dass ich selbst nicht bei xturtle geblieben bin, liegt vor allem an der Art und Weise, wie die einzelnen Funktionen bei xturtle für den Anwender zugänglich sind – das Design der Schnittstelle. Zunächst habe ich mich damit beholfen, diese Schnittstelle nach meinen Bedürfnissen umzuarbeiten, wobei ich dann wiederum feststellen musste, dass das interne Design von xturtle aus meiner Sicht ziemlich verworren ist. Dies liegt u.a. darin begründet, dass großer Wert auf Abwärtskompatibilität zum alten turtle-Modul gelegt wurde. Mittlerweile ist xturtle auf mehr als 4000 Zeilen Quellcode angewachsen – das frog-Modul kommt mit weniger als der Hälfte aus und hat eine klare innere Struktur. Hinsichtlich des Funktionsumfangs sind xturtle und frog nicht identisch. Zwar dürften geschätzte 80% der Funktionalität identisch sein, aber jedes Modul hat auch seine besonderen Features, über die das jeweils andere Modul nicht verfügt. Für das frog-Modul gilt allerdings, dass ich keine Funktion des xturtle-Moduls sehe, von der ich meine, dass sie für den Einsatzzweck von frog erforderlich sei.

Dadurch, dass ich bei der Entwicklung keine Rücksicht auf Kompatibilität zu turtle genommen habe, hatte ich mehr Freiheit bei der Implementierung und Wahl der Bezeichner. Beispiel: turtle kennt die Befehle `forward()` und `backward()` zur Bewegung der turtle. Ob die turtle dabei zeichnet oder nicht, hängt davon ab, ob – irgendwann vorher – der Zeichenstift mit `pendown()` auf die Zeichenfläche gesetzt wurde oder mit `penup()` aufgenommen wurde. Insbesondere bei längeren Programmen ist es lästig, sich über den jeweiligen Zustand des Stifts zu vergewissern – auch, wenn es eine Methode gibt, um den Zustand des Stifts abzufragen. Im frog-Modul gibt es stattdessen die Befehle `move()` – Bewegung *mit* Zeichnen – und `jump()` – Bewegung *ohne* Zeichnen. Ist das übergebene Argument negativ, erfolgt die Bewegung rückwärts.

Ein weiterer Unterschied zu turtle besteht darin, dass sämtliche Funktionalität *nur* in Form von (Objekt-)Methoden implementiert ist. Während man in turtle sowohl direkt über Funktionen Zeichenoperationen vornehmen kann als auch über die Methoden einer Pen-Instanz, geht im frog-Modul ohne Objekte gar nichts. Dieser Ansatz ist gewollt, weil frog auch dazu gedacht ist, in das objektorientierte Programmieren einzuführen. Gerade bei dem von mir gewählten Modell – Frosch im Teich – ist es in hohem Maße einsichtig, dass einerseits zunächst ein konkreter Teich erzeugt werden muss und andererseits ein Frosch erst etwas tun kann, wenn er zuvor ins Leben gerufen wurde – und zwar nicht im luftleeren Raum, sondern in einem zuvor erzeugten Teich.

## 1.2 Systemvoraussetzungen und Lizenz

Was braucht man nun, um frog einsetzen zu können? – Eine lauffähige Python-Installation ab Python 3.0 mit Tkinter. Nutzt man die Entwicklungsumgebung [Thonny](#), dann ist Python mit Tkinter automatisch mit dabei.

Und natürlich das frog-Modul, das man in der jeweils aktuellen Fassung von [www.pyxo.de](http://www.pyxo.de) herunterladen kann. Frog läuft problemlos unter Linux, Windows und Mac OS. Das Modul steht unter der GPL3-Lizenz (<http://www.gnu.org/licenses/lgpl.html>) und kann entsprechend den Lizenzbestimmungen in eigenen Programmen eingesetzt werden.

## 1.3 Über diese Dokumentation

Diese Dokumentation ist nicht als einführender Kurs für die o.g. Zielgruppe des Moduls (also Jugendliche) gedacht. Sie richtet sich vielmehr an diejenigen, die das frog-Modul in ihrer Lerngruppe oder zum privaten Vergnügen etc. einsetzen wollen und bereits über Programmierkenntnisse, möglichst auch Python (Grund-)Kenntnisse verfügen.

Der Schwerpunkt dieser Dokumentation liegt darum auch nicht auf einem didaktisch durchdachten Aufbau zum Erlernen der objektorientierten Programmierung in Python mit frog, sondern liefert eine **vollständige Darstellung aller Möglichkeiten des frog-Moduls**. In der unterrichtlichen Praxis wird man kaum auf alle Möglichkeiten des frog-Moduls zurückgreifen (ich zumindest tue das nicht), aber es ist sicher von Nutzen, wenn man sich zunächst einen Überblick verschafft über das, was mit frog möglich ist.

## 1.4 Hallo Frog!

Am Anfang (fast) jeder Einführung in eine Programmiersprache steht ein „Hallo Welt!“-Programm. Nun ist `frog` zwar keine Programmiersprache, aber eine Erweiterung von Python, mit der zu programmieren man ja auch erst einmal lernen muss. Obwohl das `frog`-Modul Frösche kennt, die Schreiben können, ist das Schreiben nicht das, was diese Frösche hauptsächlich tun. In erster Linie bewegen sich die Frösche nämlich durch einen Teich – schwimmend oder hüpfend, manchmal auch tauchend – und hinterlassen dabei Spuren, die sich im günstigsten Fall zu einer ansehnlichen grafischen Darstellung formen. Darum als Einstieg also ein Programm, das einen Frosch zeigt, der ein Quadrat zeichnet:

```
1  from frog import Pool, Frog
2
3  teich = Pool()
4  kermit = Frog(teich)
5  for n in range(4):
6      kermit.move(50)
7      kermit.turn(90)
8  teich.ready()
```

*Das ist kermit*



In Zeile 1 werden zunächst die beiden Klassen **Pool** und **Frog** importiert. Dies ist alles, was man aus dem `frog`-Modul benötigt, weil sämtliche Funktionalität in diesen beiden Klassen gekapselt ist. Bei allen folgenden Beispielen werde ich die explizite `import`-Anweisung weglassen und den entsprechenden Import voraussetzen.

In Zeile 3 wird ein `Pool`-Objekt erzeugt. Ohne weitere Angaben – so wie hier – wird dadurch ein neues Fenster mit weißem Hintergrund geöffnet, das ansonsten nichts weiter enthält. Dies ist der „Teich“, in dem sich der Frosch *kermit*, der in Zeile 4 erzeugt wird, Zeit seines Lebens aufhält. Die „Geburt“ von *kermit* zeigt sich dadurch, dass sich in der Mitte des Teichs nun ein kleines Dreieck befindet - das ist *kermit*.

In den Zeilen 5-7 zeichnet *kermit* ein Quadrat. Er bewegt sich viermal um 50 px vorwärts und dreht sich anschließend um 90° gegen den Uhrzeigersinn.

Zeile 8 zeigt an, dass der Teich nun (vorläufig) fertiggestellt ist und weitere Veränderungen an der Darstellung nur als Folge von Anwenderaktionen („Events“) möglich sind - der Teich ist nun bereit („ready“), auf Anwenderaktionen zu reagieren. Technisch gesprochen wird hier die eventloop aufgesetzt, das entspricht der `mainloop()`-Methode eines Tkinter Toplevel-Widgets.

Fehlt der abschließende Aufruf von `ready()`, dann schließt sich der Teich sofort wieder, sobald der letzte Befehl abgearbeitet wurde – zumindest dann, wenn der Teich als Standalone-Anwendung läuft und nicht aus der IDLE gestartet wurde. Kurz gesagt: Ein abschließendes `ready()` für den Teich ist meistens zwingend erforderlich, in allen anderen Fällen schadet es nicht – bei einem eingebetteten Pool wird dieser Aufruf schlicht ignoriert, die `mainloop()`-Methode der umgebenden GUI ist hier zuständig.

## 2. Der Pool

### 2.1 Einen Pool erzeugen

Ohne Pool geht nichts. Jeder Frosch braucht (s)einen Pool. Die einfachste Möglichkeit zur Erzeugung einer Pool-Instanz Namens *teich* ist ein schlichtes

```
teich = Pool()
```

Geöffnet wird ein leeres Grafikfenster mit voreingestellten Standardmaßen und weißem Hintergrund, das als Standalone-Anwendung läuft. Optional kann man die Abmessungen des Pools auch selbst festlegen, eine selbst gewählte Hintergrundfarbe setzen, den Standardtitel „The Pool“ in der Titelleiste durch einen eigenen Titel ersetzen oder auch anstelle eines einfarbigen Hintergrundes eine Hintergrundgrafik festlegen. Dazu ein paar Beispiele:

```
teich = Pool(width=8000,height=6000,bgcolor="red",title="Der Teich")
```

Geöffnet wird ein Grafikfenster, das in diesem Fall vermutlich den ganzen Bildschirm ausfüllt. Überschreiten *width* und/oder *height* die Bildschirmauflösung, dann werden stattdessen die für den konkreten Bildschirm maximalen Werte gesetzt. Das funktioniert unter Linux einwandfrei („echte Maximierung“), auf Windowssystemen annähernd korrekt. Wem das nicht genügt, der kann als Anwender das Fenster dann immer noch „echt maximieren“. Sind die Abmessungen des Pools kleiner als der Bildschirm, wird das Grafikfenster per Voreinstellung links oben platziert. Wem das nicht gefällt, der kann über den Schlüsselwortparameter **pos** den Pool bei seiner Erzeugung auch an anderer Stelle positionieren. Mögliche Werte für **pos** sind "left", "right", "top", "topleft", "topright", "bottom", "bottomleft", "bottomright" und "center". Bei Verwendung einer unbekannten Positionsangabe wird der Pool links oben platziert.

Zu den unterschiedlichen Möglichkeiten zur Angabe von Farben findet man Ausführliches in Abschnitt 2.6 („Das Farbmodell“).

```
teich = Pool(bgimage="forest.gif",title="Im Wald")
```

Geöffnet wird ein Grafikfenster, das als Hintergrund das Bild aus der Datei *forest.gif* enthält. Fehlen – so wie hier – Angaben zur Breite und Höhe des Pools, dann werden diese entsprechend der Abmessungen der Hintergrundgrafik gewählt. Wird Breite und Höhe festgelegt *und* eine Hintergrundgrafik gewählt, dann erhält der Pool die gewünschten Abmessungen und die Hintergrundgrafik wird zentriert gesetzt. Hat sie kleinere Abmessungen als der Pool, dann bleibt außen herum ein Rand in der eingestellten Hintergrundfarbe; ist sie größer, dann wird der überschüssige Rand abgeschnitten.

Als Dateiformat für Hintergrundgrafiken stehen – bedingt durch die Einschränkungen von Tkinter – das gif-Format und (ab Tk 8.6, d.h. ab Python 3.2) auch png zur Verfügung; jpeg-Dateien können nicht verwendet werden. Existiert die angegebene Datei nicht, dann wird die Zuweisung ignoriert, der Pool bekommt keine Hintergrundgrafik.

Für eine fortgeschrittene Programmierung gibt es außerdem noch die Möglichkeit, den Pool in eine umgebende Tkinter-GUI einzubinden. Dazu muss dann das entsprechende übergeordnete Element der GUI als *root* angegeben werden. Im Fall eines Toplevel- oder Frame-Widgets *frame* also so:

```
teich = Pool(root=frame)
```

Da die Klasse Pool von der Tkinter-Klasse Canvas abgeleitet ist und keine Methoden überschrieben wurden, können sämtliche Canvas-Methoden auch auf Pool-Objekte angewendet werden. Für den Einstieg in die Programmierung mit frog dürfte das aber nur von geringem Interesse sein. Auf diese Möglichkeit wird im weiteren Verlauf auch nicht weiter eingegangen.

## 2.2 Den Pool konfigurieren

Die Abmessungen des Pools sind nach der Konstruktion aus dem Programm heraus unveränderlich – jedenfalls, wenn es ein Standalone-Pool ist. Es ist lediglich möglich, den gesamten Pool unsichtbar bzw. dann wieder sichtbar zu machen, in dem man das Datenattribut **visible** auf False bzw. True setzt. Dabei bleiben alle Zeichnungen im Pool erhalten – der Pool wird also nicht geschlossen! Interessant dürfte diese Möglichkeit hauptsächlich dann sein, wenn man mit mehreren Pools, etwa einem Haupt- und einem Nebenfenster, operiert.

Weiterhin hat der Anwender die Möglichkeit, das den Pool umgebende Fenster nach Belieben in der Größe zu verändern und den Pool dadurch zu vergrößern oder zu verkleinern – es sei denn, man dies durch die Verwendung der Option **resizable=False** bei der Erzeugung des Pools ausdrücklich unterbunden. Auch diese Festlegung kann nur einmalig bei der Instanzbildung des Pools getroffen werden. Unveränderlich ist ebenfalls die Entscheidung, ob der Pool standalone läuft oder in eine umgebende Tkinter-GUI eingebettet ist.

Die anderen Eigenschaften, die optional bei der Konstruktion eines Pools festgelegt werden können, sind jederzeit änder- oder ergänzbar. Zusätzlich gibt es noch die Möglichkeit, die Cursorform im Pool zu ändern. Dazu ein Beispiel:

```
1  teich = Pool()
2  teich.bgcolor = "orange"
3  teich.title = "Orangenteich"
4  teich.cursor = "cross"
5  teich.visible = False # teich unsichtbar - nicht sinnvoll!
6  teich.ready()
```

Unmittelbar nach Zuweisung einer Cursorform wird diese wirksam und gilt für den gesamten Pool. Im Beispiel nimmt der Cursor die Form eines Kreuzes an, die Standardform (Pfeil) erhält man durch Zuweisung einer leeren Zeichenkette. Weitere Cursorformen sind „pencil“, „target“, „hand1“ und „watch“. Für eine vollständige Liste aller 78 Formen verweise ich auf die Übersicht bei Shipman (<http://infohost.nmt.edu/tcc/help/pubs/tkinter/cursors.html>).

## 2.3 Eigenschaften des Pools ermitteln

Zwar können nicht alle Eigenschaften nach Erzeugung eines Pools *geändert* werden, aber alle diese – und wenige weitere – Eigenschaften können *abgefragt* werden. Dazu ein Beispiel:

```
1  teich = Pool(bgcolor="snow")
2  pool_id = teich.id           # "Pool-0001" (laufende Nummer)
3  sichtbar = teich.visible
4  hintergrundfarbe = teich.bgcolor
5  hintergrundbild = teich.bgimage # in diesem Fall None
6  breite = teich.width
7  hoehe = teich.height
8  titel = teich.title
9  cursor = teich.cursor
10 aufloesung = teich.resolution # Bildschirmauflösung in dpi
11 froschliste = teich.frogs     # Liste aller Frog-Objekte
12 aktiv = teich.action          # True, wenn mind. 1 Frosch aktiv
13 teich.ready()
```

Die Zugriffe auf die (Daten-)Attribute, die die Eigenschaften des Pools enthalten, sind dabei nur scheinbar direkte Zugriffe. Alle Zugriffe auf diese Attribute werden über get-/set-Methoden im Hintergrund verwaltet, so dass z.B. die Werte aus den Attributen `width` und `height` zwar ausgelesen, aber nicht geändert werden können. Der Versuch, diesen Attributen neue Werte zuzuweisen, führt zu einer Fehlermeldung. Dies gilt auch für die Bildschirmauflösung, die Liste der Frösche im Teich und den Aktivitätszustand, der darüber Auskunft gibt, ob mindestens einer der Frösche auf dem Bildschirm gerade in Bewegung ist. Dies zu wissen, kann bei der Ereignisbehandlung hilfreich sein – mehr dazu in Kapitel 4.

Wer die Zugriffe auf die Eigenschaften des Pools lieber über Methoden vornehmen möchte, der kann auch die entsprechenden getter/setter verwenden. Sie tragen jeweils die gleichen Namen wie die Datenattribute mit – je nach Funktionalität – vorangestelltem `get` bzw. `set`. Das könnte dann so aussehen:

```
1  teich = Pool(width=400,height=300)
2  breite = teich.getwidth()
3  hoehe = teich.getheight()
4  teich.settitle("Mein Teich")
5  teich.setbgcolor("brown")
6  teich.ready()
```

Ich persönlich bevorzuge die „direkten“ Zugriffe auf die Datenattribute. Es macht deutlicher, dass es hier nur um die Änderung einer *Eigenschaft* geht und keine eigentliche Tätigkeit verrichtet wird – das wäre dann die Aufgabe von Methoden. Außerdem entspricht es eher der Philosophie von Python, während die getter-/setter-Variante mehr an Java erinnert.



## 2.4 Operationen im Pool

Da alle Aktionen, die sich innerhalb des Pools abspielen, von Fröschen vorgenommen werden, verfügt der Pool selbst nur über fünf Methoden, von denen zwei wiederum nur wirksam sind, wenn ein Pool standalone läuft: Die im Einführungsbeispiel schon behandelte Methode **ready()**, mit der die eventloop aufgesetzt wird, und die Methode **close()**, durch die ein Pool aus dem Programm heraus geschlossen werden kann. Wichtig: Sind mehrere Pools gleichzeitig geöffnet, dann bewirkt das Schließen des zuerst geöffneten Pools auch das Schließen aller anderen Pools. Ansonsten wird immer nur die Pool-Instanz geschlossen, die **close()** aufgerufen hat.

Die Methoden **snapshot()** und **snapinfo()** dienen zur Erstellung eines Screenshots des Pools bzw. eines Pool-Ausschnitts und werden in Kapitel 2.7 behandelt, die Methode **listen()** dient der Ereignisbehandlung, die in Kapitel 4 gesondert behandelt wird.

## 2.5 Das Koordinatensystem

Die Orientierung, Positionierung und Lokalisierung innerhalb eines Pools geschieht über Koordinaten in einem kartesischen Koordinatensystem in der Einheit Pixel. Der Nullpunkt dieses Koordinatensystems befindet sich stets genau in der Mitte eines Pools. In einem Pool mit den Abmessungen 400×300 Pixel liegt also z.B. der Punkt P(200,150) rechts oben in der Ecke und der Punkt Q(-200,-150) links unten in der Ecke.

## 2.6 Das Farbmodell

Das frog-Modul ist bei der Wahl der Farben sehr flexibel und ermöglicht insbesondere auf einfache Weise das rechnerische Verändern oder zufällige Wählen eines Farbtons aus dem gesamten RGB-Farbraum. Zur Überprüfung und Konvertierung der verschiedenen zulässigen Farbangaben enthält frog eine Hilfsklasse **Color**, die dazu einige statische Methoden („Klassenmethoden“) bereitstellt. Der explizite Einsatz dieser Methoden wird in der Regel nicht erforderlich (aber möglich) sein – einige Methoden der Klassen Frog und Pool greifen jedoch darauf zurück. Zu beachten ist in jedem Fall, dass sich keine Instanzen von Color erzeugen lassen – Farben sind also keine Color-Objekte, sondern je nach Art der Farbangabe ein String oder ein Zahlentripel. Beispiel:

```
1  from random import random, choice
2
3  pool = Pool()
4  tomato = "tomato"
5  lightsteelblue = "#b0c4de"
6  indianred = 205,92,92
7  peachpuff = 1.0,0.8549,0.7255
8  randomcolor = random(),random(),random()
9  farben = [tomato,lightsteelblue,indianred,peachpuff,randomcolor]
10 pool.bgcolor = choice(farben)  # wählt eine Farbe zufällig aus
11 pool.ready()
```

Die Zeilen 4 bis 7 zeigen der Reihe nach die vier unterschiedlichen Möglichkeiten, einen Farbwert festzulegen. Die ersten beiden entsprechen den Varianten, die auch in HTML/CSS verwendet werden. Die erste Variante ist die Angabe eines *vordefinierten Farbnamens* aus der Menge der – auf dem jeweiligen Betriebssystem – verfügbaren Farbnamen. Zumindest unter Linux scheint es so zu sein, als seien fast alle 140 CSS-Farbnamen bekannt („aqua“, „crimson“ und „silver“ funktionieren auf meinem System aber z.B. nicht). Ist man sich unsicher und will nicht durch die Angabe eines nicht existierenden Farbnamens einen Programmabbruch riskieren, dann kann man mit der Methode `isvalid()` der Klasse `Color` zuvor die Gültigkeit des Farbnamens prüfen:

```
gueltig = Color.isvalid("middlegreenblue") # liefert hier False
```

Die zweite Möglichkeit zur Angabe eines Farbwerts ist ein sechsstelliger *hexadezimaler RGB-Code* mit vorangestellter Raute, und zwar als Zeichenkette. Will man eine der anderen drei Farbdarstellungen in diese Darstellung umwandeln, so kann man dazu die Methode `toRGBstr()` der Klasse `Color` verwenden, also z.B.

```
farbcode = Color.toRGBstr("lightsteelblue") # liefert "#b0c4de"
```

Bei den beiden übrigen Varianten zur Festlegung eines Farbwerts wird jeweils ein *Tupel aus drei numerischen Werten* angegeben, und zwar entweder dezimale Integerwerte aus dem Bereich von 0 bis 255 oder Fließkommawerte aus dem Bereich von 0.0 bis 1.0, wobei jeweils der entsprechende Rot-, Grün- und Blauanteil angegeben wird. Diese numerischen Werte erlauben es, mit relativ einfachen Mitteln bestimmte Teile des Farbraums (nahezu) stufenlos zu durchlaufen, eine schon vorhandene Farbe heller oder dunkler zu machen oder zufällige Farbwerte zu generieren.

Die Methoden der Klasse `Color` ermöglichen es, jede dieser vier Darstellungsweisen von Farben in jede andere umzuwandeln – mit Ausnahme der Umwandlung in einen Farbnamen. Dazu verwendet man eine der Methoden `toRGB()`, `toRGBstr()` oder `toRGBrel()`, denen als Argument eine der vier möglichen Darstellungsarten übergeben wird. Noch ein Beispiel:

```
farbstr = Color.toRGBstr(0.3,0.6,0.75) # liefert "#4d99bf"
farbrel = Color.toRGBrel("#22af93")    # liefert
(0.1333,0.6863,0.5765)
```

Zum Schluss sei noch erwähnt, dass es auch möglich ist, *Transparenz* als Farbe festzulegen, indem man eine leere Zeichenkette als Farbwert zuweist. Das ist zwar für den Hintergrund des Pools weniger interessant, ist aber z.B. die Voreinstellung für die Füllung des Standardfroschs. Die Umwandlung von Transparenz mit Hilfe der Methoden der Klasse `Color` in eine andere Farbdarstellung ist jedoch nicht möglich, weil es keine RGB-Darstellung von Transparenz gibt.

## 2.7 Screenshot erstellen

Wird das frog-Modul auf einem Windows- oder Linuxsystem eingesetzt, dann lässt sich direkt aus einem frog-Programm heraus der gesamte Pool oder ein frei wählbarer Ausschnitt des Pools mittels der Methode **snapshot()** in einem der Grafikformate gif, png oder jpg abspeichern. Da das GUI-Toolkit Tkinter, auf dem frog basiert, selbst keine Möglichkeit mitbringt, einen Screenshot zu erstellen, muss dazu auf externe Programme zurückgegriffen werden. Dazu stehen sowohl unter Windows als auch unter Linux zwei Möglichkeiten zur Wahl.

**Voraussetzungen unter Windows:** Benötigt wird das kostenlose Programm **IrfanView** ([www.irfanview.de](http://www.irfanview.de)) oder die **Python Image Library** ([www.pythonware.com/products/pil](http://www.pythonware.com/products/pil)). Sofern ein Pool als eigenständiges Programm läuft, erzielt man mit beiden Tools gleich gute Resultate. Bei einem in eine Tkinter-GUI eingebetteten Pool wird bei Verwendung der PIL nicht in allen Fällen der richtige Bildausschnitt gespeichert. Die Verwendung der PIL hat jedoch gegenüber IrfanView den Vorteil, dass man sich nicht darum kümmern muss, wo sich das Programm befindet. Da IrfanView ohne Installation einsatzfähig ist („Stickware“), kann es sich hingegen an einer beliebigen Stelle des Dateisystems ohne Eintrag in der Registry befinden. Die PIL wird mit einem einfachen Installer geliefert und ist auch für Laien problemlos installierbar. Sind sowohl IrfanView als auch PIL installiert, dann verwendet **snapshot()** automatisch IrfanView (sofern es auffindbar ist).

**Voraussetzungen unter Linux:** Benötigt wird die Grafik-Toolsammlung **ImageMagick** (<http://www.imagemagick.org>), die auf vielen Systemen bereits vorinstalliert ist oder über den Paketmanager des Systems nachinstalliert werden kann. Ist ImageMagick nicht installiert, wird das bei den meisten Linux-Distributionen zum graphischen Grundsystem gehörige Tool **xwd** in Kombination mit der Konvertersammlung **netpbm** eingesetzt, wobei dann aber nur der gesamte Pool gespeichert werden kann und das gif-Format nicht zur Verfügung steht.

Da die unter Linux zum Einsatz kommenden Programme das Pool-Fenster über dessen Titel identifizieren, sollte man darauf achten, bei Verwendung mehrerer Pools über das **title**-Attribut unterschiedliche Fenstertitel zu vergeben.

In Einzelfällen kann es passieren, dass die Verwendung des gif-Formats in Kombination mit der Festlegung eines Pool-Ausschnitts nicht zum gewünschten Ergebnis führt. In diesen Fällen hilft das Speichern im png-Format und die manuelle Umwandlung ins gif-Format.

Um sich vor einem Screenshot einen Überblick über die konkreten Möglichkeiten bei der Durchführung zu verschaffen, kann man die Methode **snapinfo()** verwenden, der als optionaler Parameter der Pfad zum gewünschten Graphikprogramm übergeben werden kann. Dies sollte – wenn überhaupt – nur im Falle von „Windows mit IrfanView“ erforderlich sein. In allen anderen Fällen werden die Programme automatisch im System lokalisiert. Ein Aufruf von **snapinfo()** liefert ein Dictionary zurück, das u.a. folgende Informationen enthält:

- ["tool"]: Name des verwendeten Tools als String oder None, falls keins verfügbar ist.
- ["area"]: False, wenn nur der gesamte Pool aufgenommen werden kann, sonst True.
- ["format"]: Eine Liste der möglichen Grafikformate, maximal ["png", "gif", "jpg"]

Will man einen Screenshot des gesamten Pools anfertigen, genügt ein Aufruf der Methode **snapshot()** – ohne Parameter. Dies bewirkt, dass im Moment des Aufrufs ein Screenshot des gesamten Pools (ohne Fensterdekoration!) angefertigt wird, der im gleichen Verzeichnis wie das aufrufende Programm unter dem Namen `pool-YYYYMMDD-HHMMSS.png` abgespeichert wird (dabei werden die Großbuchstaben ersetzt durch das aktuelle Datum und die aktuelle Zeit). Zurückgeliefert wird jeweils der tatsächlich verwendete Name der angefertigten Grafikdatei bzw. eine leere Zeichenkette – je nachdem, ob der Screenshot erfolgreich angefertigt und in einer Datei gespeichert werden konnte oder eben nicht. Es folgen zwei Beispiele zur Erläuterung weiterer Möglichkeiten:

```
1 pool = Pool()
2 irfanpath = "C:\\Programme\\IrfanView"
3 snapdict = pool.snapinfo(irfanpath)
4 success = False
5 if "gif" in snapdict["format"]:
6     filename = pool.snapshot("pool.gif",path=irfanpath)
7 else:
8     filename = pool.snapshot("pool.jpg",path=irfanpath)
9 if filename:
10    print("Screenshot gespeichert als Datei %s" %filename)
11 pool.ready()
```

Das gewünschte Grafikformat wird über die Endung des Dateinamens festgelegt. Fehlt eine Endung oder handelt es sich um ein nicht unterstütztes Grafikformat, wird automatisch das png-Format verwendet, das im übrigen auch die erste Wahl für einen Screenshot ist. Gute Gründe für das jpg-Format gibt es eigentlich nicht, weil es verlustbehaftet ist (obwohl die Qualität auf 95% gesetzt wird) und in der Regel zu größeren Dateien führt.

```
1 sheet = Pool()
2 snapdict = sheet.snapinfo()
3 if snapdict["area"]:
4     snapshot("poolpart.png",area=(-100,0,200,50))
5 else:
6     snapshot("fullpool")
```

Erlaubt das zum Einsatz kommende Tool das Speichern eines Pool-Ausschnitts, dann kann dieser Ausschnitt als 4-Tupel angegeben werden. Die ersten beiden Werte sind die x- und y-Koordinate der linken unteren Ecke, die letzten beiden Werte die Koordinaten der rechten oberen Ecke des gewünschten Poolausschnitts – jeweils bezogen auf das Koordinatensystem des Pools. Auch bei Größenänderung des Pools durch den Anwender wird immer der gleiche Ausschnitt beibehalten (sofern der Pool nicht kleiner ist als der Ausschnitt).

Im obigen Beispiel wird für den Fall, dass eine Ausschnittspeicherung nicht möglich ist, das png-Format automatisch gewählt, da die Datei keine entsprechende Endung aufweist.

## 3. Die Frösche

Die Klasse `Frog` enthält den Bauplan für die Frösche, die sich im Teich aufhalten sollen. Da jeder Frosch einen Lebensraum braucht, muss bei der Instanzbildung ein Teich für den Frosch angegeben werden – vor der Erzeugung eines Frosches liegt also immer die Erzeugung eines Teichs.

Soll der Frosch unmittelbar nach der Erzeugung zunächst unsichtbar bleiben, dann kann das mit dem Schlüsselwortparameter `visible` festgelegt werden – standardmäßig ist ein Frosch nach der Erzeugung sichtbar und befindet sich in der Mitte des Pools. Ein Beispiel:

```
1  teich = Pool()
2  arno = Frog(teich)
3  bruno = Frog(teich, visible=False)
4  teich.ready()
```

### 3.1 Der Frosch als Persönlichkeit

Jeder Frosch führt ein Eigenleben im Pool. Sein Handeln ist unabhängig von dem aller anderen Frösche im Teich. Er ist selbst für seine Aktionen verantwortlich – er bewegt sich in seiner ihm eigenen Geschwindigkeit, hinterlässt Spuren, entfernt sie bei Bedarf wieder und beendet seine Existenz im Pool, wenn er nicht mehr leben will.

Die Persönlichkeit eines Frosches im Teich ist äußerlich sichtbar dadurch gekennzeichnet, dass u.a. die Gestalt und die Farbe eines Froschs frei wählbar sind. Ein Beispiel:

```
1  teich = Pool()
2  arno = Frog(teich)
3  arno.color = "saddlebrown"
4  arno.bodycolor = "salmon"
5  arno.shape = "frog"
6  teich.ready()
```



Für die Datenattribute des Froschs gilt wie schon für den Pool selbst: Es handelt sich dabei ausschließlich um „managed attributes“, bei denen die Zugriffe nur scheinbar direkt erfolgen. In Wahrheit stecken entsprechende getter/setter dahinter, die auch in allen Fällen als Alternative zu einem direkten Attributzugriff verwendet werden können. In den wenigen Fällen, wo ein Attribut nur lesbar ist, existiert dann nur der entsprechende getter.

Die Möglichkeiten der Farbwahl wurden bereits im Abschnitt 2.6 behandelt. Während sich `bodycolor` auf das Innere des Froschs bezieht, wird durch `color` nicht nur die Umrandung des Froschs festgelegt, sondern zugleich auch die aktuelle Zeichenfarbe, in der der Frosch seine Spuren hinterlässt, wenn er durch den Teich schwimmt. Diese Eigenschaften bleiben so lange erhalten, bis sie durch neue Werte überschrieben werden. Der Standardfrosch hat eine schwarze Umrandung und einen durchsichtigen Körper.

Mittels **shape** wird die Gestalt des Frosches festgelegt. Sie kann aus einer Liste vordefinierter Froschformen ausgewählt werden, die die Formen „arrow“ (= dreieckige Standardform), „cross“ (= Kreuz), „turtle“ (= Schildkröte) und „frog“ (= Frosch) umfasst. Eine Liste aller möglichen Froschformen lässt sich über das Attribut **shapes** (Plural!) abrufen (nur lesbar).

Die Liste der möglichen Froschformen kann durch eigene Definitionen beliebig erweitert werden, und zwar sowohl durch Froschformen, die durch ein Polygon gebildet werden (so wie die vordefinierten Froschformen), als auch durch Froschformen, die als Grafik aus einer Datei (*gif*, ab Python 3.2 auch *png*) gelesen werden. Das Beispiel zeigt beide Möglichkeiten:

```
1  see = Pool()
2  arno = Frog(see)
3  bruno = Frog(see)
4  arno.shape = "ente"      # Fehlermeldung! Es gibt keine Form "ente"
5  arno.shape = "ente", "ente.gif"  # Entengestalt wird definiert
6  arno.color = "purple"
7  bruno.shape = "square", (0,0),(10,0),(10,10),(0,10)
8  bruno.color = "seagreen"
9  carlo = Frog(see)
10 carlo.shape = "ente"    # FUNKTIONIERT! Es gibt nun eine Form "ente"
11 see.ready()
```

Der Versuch, arno in Zeile 4 eine Form Namens „ente“ zuzuweisen, führt zu einem Laufzeitfehler, weil es (noch) keine Form dieses Namens gibt. Ein vorheriger Aufruf von

```
if "ente" in arno.shapes: arno.shapes = "ente"
```

hätte das verhindern können. In Zeile 5 wird eine neue Froschgestalt Namens „ente“ definiert. Dazu wird zunächst der gewünschte Name angegeben und danach – durch Komma getrennt – die Definition der Gestalt, in diesem Fall die Grafikdatei ente.gif. Falls keine korrekte Datei dieses Namens existiert, behält der Frosch seine bisherige Form. War die Definition erfolgreich, dann steht diese Froschgestalt ab sofort auch allen anderen Fröschen zur Verfügung – unabhängig davon, ob sie zum Zeitpunkt der neuen Formendefinition schon existierten oder nicht. So kann also in Zeile 10 dem neu geborenen Frosch *carlo* ebenfalls die Entenform zugewiesen werden.

Die Zuweisung der Farbe in Zeile 6 wirkt sich zunächst nicht sichtbar aus. Da *arno* als Form eine Bitmapgrafik besitzt, bleiben **color** und **bodycolor** ohne Wirkung. Sie können natürlich trotzdem gesetzt werden und werden dann sofort wirksam, sobald *arno* eine Polygongestalt erhält. Dennoch ist die Zuweisung in Zeile 6 sinnvoll, denn **color** legt ja auch die Zeichenfarbe für die nachfolgenden Operationen fest.

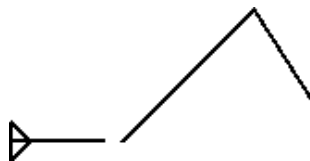
In Zeile 7 wird dem Frosch *bruno* die Form „square“ zugewiesen. Da diese bisher nicht existiert, wird sie gleichzeitig mit der Zuweisung definiert durch Angabe der Eckpunkte eines Polygons. Sind Anfangs- und Endpunkt nicht identisch, werden sie verbunden. Die Festlegung der Farbe in Zeile 8 wirkt sich sofort sichtbar auf seine Umrandungsfarbe aus.

Intern wird die Liste der möglichen Formen in einem dictionary gehalten, das als Klassenattribut von Frog definiert ist. Hier fungieren die Namen der Formen als Schlüssel und die Definitionen (Dateinamen oder Tupel von Punkten) als zugeordnete Werte. Ein direkter Zugriff auf dieses dictionary ist nicht ohne weiteres möglich (natürlich geht auch das, aber für die vorge-sehene Zielgruppe wird es eher nicht möglich sein), so dass Änderungen daran nur über die Definition einer Froschgestalt – wie oben beschrieben – möglich sind. Um eine vorhandene Form durch eine andere zu ersetzen, genügt es, sie durch eine neue Definition zu überschreiben. Die mitgelieferten Standardformen sind jedoch nicht (auf diese Weise) überschreibbar.

## 3.2 Frösche in Aktion

Bereits im einführenden Beispiel „Hallo Frog!“ wurde ein Frosch in Aktion gezeigt. Insgesamt stehen mehr als 20 Methoden zur Verfügung, die einem Frosch Beine machen. Unmittelbar nach der Erzeugung eines Froschs befindet sich dieser – in der Anfangsgestalt eines schwarzen Dreiecks mit durchsichtigem Inneren – an Position (0,0) des Pools (also genau in der Mitte) und ist nach Osten hin ausgerichtet (0° im mathematisch üblichen Sinne). Beispiel:

```
1 brunnen = Pool()
2 arno = Frog(brunnen)
3 arno.move(50)
4 arno.jump(10)
5 arno.turn(45)
6 arno.move(100)
7 arno.turn(77.81)
8 arno.move(-63.2)
9 arno.home()
10 brunnen.ready()
```



Unter Verwendung der drei grundlegenden Bewegungsfunktionen – **move()**, **jump()** und **turn()** – lässt sich ein Frosch im Pool bewegen. Mittels **turn()** wird die Ausrichtung geändert, und zwar in Gradmaß gegen den Uhrzeigersinn (mathematisch positive Drehrichtung); bei negativen Werten entsprechend *im* Uhrzeigersinn. Sofern der Frosch eine Polygongestalt hat, dreht sich diese sichtbar mit; bei einer Bitmapgestalt ist dies aus technischen Gründen nicht möglich, weil Tkinter die dafür benötigten Voraussetzungen nicht mitbringt.

Die Methoden **move()** und **jump()** unterscheiden sich dadurch, dass der Frosch bei einer Bewegung mittels **move()** eine sichtbare Spur hinterlässt; bei **jump()** ist das nicht der Fall. Beiden Methoden gemeinsam ist, dass sie *relativ* bezogen auf die gerade aktuelle Position und Ausrichtung des Frosches wirken und den Frosch um die als Argument übergebene Anzahl von Pixeln im Pool vorwärts (bei negativen Werten: rückwärts) bewegen. Die Entfernung kann als Integer- oder Fließkommazahl angegeben werden. Dies gilt auch für die Angabe des Winkels als Argument für **turn()**, der sich relativ auf die aktuelle Ausrichtung des Froschs bezieht.

Das abschließende **home()** bewirkt, dass der Frosch – ohne zu zeichnen – wieder seine Ausgangsposition einnimmt: In der Mitte des Pools mit 0° Ausrichtung.

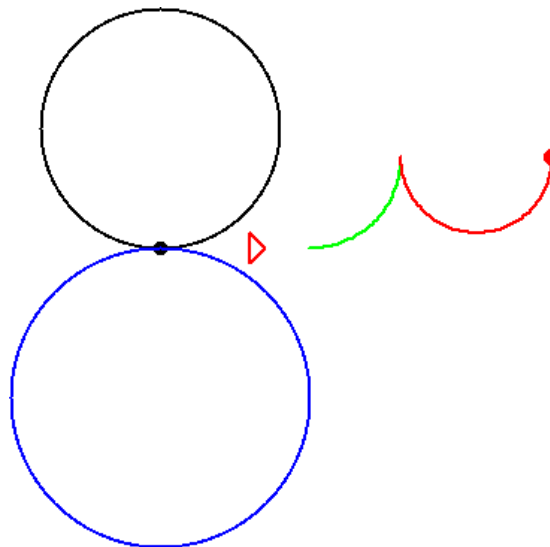
Im Grunde sind diese Funktionen ausreichend, um die Erzeugung (fast) beliebiger grafischer Figuren zu ermöglichen. Ganz bewusst wurde z.B. keine Methoden implementiert, um ein Rechteck oder ein regelmäßiges Vieleck zu zeichnen. Dadurch bietet es sich an, entsprechende Funktionen selbst zu definieren oder – je nach Konzept – eine Klasse Superfrog zu entwerfen, deren Frösche über entsprechende Methoden verfügen.

Weil sich das ein oder andere aber – insbesondere für die Zielgruppe – nicht so einfach realisieren lässt, habe ich noch einige weitere Zeichenmethoden ergänzt. Dazu gehört insbesondere die Methode **circle()** zum Zeichnen von Kreis(bögen). Ein Beispiel:

```

1  becken = Pool()
2  ben = Frog(becken)
3  ben.dot()
4  ben.circle(80)
5  ben.color = "blue"
6  ben.circle(-100)
7  ben.jump(100)
8  ben.color = "green"
9  ben.circle(60,90)
10 ben.color = "red"
11 ben.circle(-50,-180)
12 ben.dot(10)
13 ben.home()
14 ben.jump(60)
15 becken.ready()

```



Die Methode **circle()** in ihrer Grundform – mit dem Radius als Argument – zeichnet einen Kreis mit dem gegebenen Radius und den aktuellen Zeicheneigenschaften des Frosches, und zwar ausgehend von der aktuellen Orientierung des Froschs links herum, so dass der Mittelpunkt in einer Entfernung von Radius px links vom Frosch liegt, bezogen auf dessen Orientierung. Um zu erreichen, dass der Mittelpunkt rechts vom Frosch liegt, wird der gewünschte Radius als negativer Wert angegeben.

Außerdem lassen sich mit **circle()** Kreisbögen zeichnen. In diesem Fall muss als zweiter Parameter der Mittelpunktswinkel des gewünschten Kreisbogens angegeben werden. Ist dieser Wert positiv, bewegt sich der Frosch vorwärts, ist er negativ, rückwärts. Möchte man nur erreichen, dass der Frosch sich auf einer kreis(bogen)förmigen Bahn bewegt, ohne dabei eine Spur zu hinterlassen, dann verwendet man den Schlüsselwortparameter **draw=False**.

Mit der Methode **dot()** hinterlässt ein Frosch an seiner aktuellen Position einen Punkt in Zeichenfarbe. Der Durchmesser kann optional angegeben werden; fehlt eine Angabe, beträgt er 8 px. Optional kann über den Schlüsselwortparameter **fill** eine Farbe angegeben werden, mit der der Kreis gefüllt werden soll, oder **fill=False**, falls der Kreis nicht gefüllt werden soll. Gegenüber **circle()** erreicht man mit **dot()** eine deutlich höhere Zeichengeschwindigkeit.



### 3.3 Frösche lokalisieren und positionieren

Die bisher behandelten Methoden zur Bewegung eines Froschs im Pool sind alle *relativer* Natur – die Koordinaten des Froschs werden dafür nicht benötigt. In manchen Fällen ist es aber durchaus von Vorteil, darauf zurück greifen zu können, und so lassen sich die entsprechenden Daten eines Frosches auch jederzeit abrufen. Beispiel:

```
1 pool = Pool()
2 hans = Frog(pool)
3 hans.turn(30)
4 hans.jump(100)
5 hans.turn(40)
6 x, y = hans.pos          # x=86.6, y=50
7 x, y = hans.x, hans.y    # ebd.
8 angle = hans.angle       # angle=70
9 pool.ready()
```

Die Attribute **pos** und **angle** enthalten die Informationen zur aktuellen Position und Ausrichtung von *hans*; die Positionsangaben kann man auch einzeln über die Attribute **x** und **y** abfragen. Über entsprechende Zuweisungen an diese Attribute kann ein Frosch auch absolut im Pool positioniert und ausgerichtet werden. Alternativ kann man dafür die Methoden **jump-to()** und **turnto()** verwenden, wobei **jump-to()** entweder ein Koordinatenpaar (Tupel) oder zwei einzelne Koordinaten erwartet, **turnto()** ein Winkelmaß, auf das der Frosch dann ausgerichtet wird. Die Wirkung von **turnto()** ist die gleiche wie eine Zuweisung an **angle**. Zu beachten ist hier, dass – anders als bei einer relativen Drehung mittels **turn()** – bei absoluter Zuweisung eines Winkels über **angle** oder **turnto()** der Frosch jeweils den kleineren der beiden möglichen Drehwinkel wählt, um die gewünschte Ausrichtung zu erreichen.

Die Wirkung von **jump-to()** ist im *Ergebnis* identisch mit einer Zuweisung an **pos** (bzw. an **x** und **y**), beide Varianten unterscheiden sich jedoch in der Ausführung. Während bei Verwendung von **jump-to()** ein Frosch zunächst in die Zielrichtung ausgerichtet wird, sich dann zum Zielpunkt bewegt und anschließend seine ursprüngliche Ausrichtung wieder einnimmt, erfolgt bei direkter Zuweisung neuer Koordinaten keine Ausrichtungsänderung. Dadurch werden auch Bewegungen quer zur aktuellen Ausrichtung eines Froschs ermöglicht.

Schließlich gibt es noch die Methode **moveto()**, um einen Frosch mit Spur an eine bestimmte Stelle im Pool zu bewegen – Parameter wie bei **jump-to()**. Beispiel:

```
1 pool = Pool()
2 rudi = Frog(pool)
3 rudi.pos = 100,50        # oder: rudi.jump-to(100,50)
4 rudi.angle = 47.5        # oder: rudi.turnto(47.5)
5 ziel = -80,30
6 rudi.moveto(ziel)        # zeichnet von (100,50) bis (-80,30)
7 pool.ready()
```

Insbesondere bei der Programmierung von Spielen mit mehreren Fröschen ist es hilfreich, wenn man Entfernung und Winkel zu einem anderen Frosch oder einer bestimmten Stelle im Pool kennt. Wer das nicht selbst ausrechnen möchte, der kann sich dafür der Methoden **angleto()** und **distanceto()** bedienen – das folgende Beispiel zeigt wie.

```
1  from random import randrange
2
3  teich = Pool()
4  paul = Frog(teich)
5  paul.color = "firebrick"
6  rudi = Frog(teich)
7  rudi.color = "chocolate"
8  paul.pos = randrange(200),randrange(100)
9  rudi.jumpto(randrange(-200,0),randrange(-100,0))
10 paulsweg, rudisweg = paul.distanceto(0,0), rudi.distanceto((0,0))
11 if paulsweg>rudisweg:
12     print "Paul ist weiter gesprungen!"
13 else:
14     print "Rudi ist weiter gesprungen!"
15 # Jetzt schwimmt Rudi bis kurz vor Paul
16 entfernung, winkel = rudi.distanceto(paul), rudi.angleto(paul)
17 rudi.turnto(winkel)
18 rudi.move(0.9*entfernung)
19 teich.ready()
```

Wie man am Beispiel erkennen kann, bieten beide Methode verschiedene Möglichkeiten zur Angabe des Ortes, zu dem die Entfernung und das Winkelmaß ermittelt werden sollen: Entweder die beiden Koordinaten eines Punktes *oder* ein Wertepaar *oder* eine Frosch-Instanz. Der von **angleto()** gelieferte Winkel ist derjenige Winkel, der als absoluter Winkel eingestellt werden muss, um in Richtung des anvisierten Punktes zu gelangen.

### 3.4 Weitere Eigenschaften

Über die bislang genannten Eigenschaften hinaus verfügt jeder Frosch noch über weitere Attribute, die abgefragt und in den meisten Fällen auch geändert werden können. Dazu gehört die Spurbreite der Zeichnungen im Attribut **width** sowie die Umrandungsbreite (bei polygonaler Gestalt) im Attribut **borderwidth**, beides in der Einheit px mit Defaultwert 2. Möglich sind Zahlenwerte größer oder gleich 1, bei **borderwidth** auch 0 (= keine Umrandung).

Gezeichnete Linien haben standardmäßig abgerundete Ecken und das liefert in den meisten Fällen auch das gewünschte Ergebnis – etwa, wenn man einen Streckenzug zeichnet oder eine geschlossene Figur. Wenn man jedoch sehr dicke Linien nutzen möchte, um etwa eine rechteckige Fläche einzufärben, ist es kontraproduktiv. Hilfe bietet hier das Attribut **lineend**, das die Werte "round" (Enden abgerundet) und "flat" (Enden abgeschnitten) kennt.

Die Größe eines Froschs – genauer gesagt, die Maße seiner „bounding-box“, d.h. des kleinsten den Frosch umgebenden Rechtecks – lässt sich aus dem nicht veränderbaren Frosch-Attribut **size** abrufen. Zurückgeliefert wird ein Tupel mit Breite und Höhe in px. Bei unsichtbaren Fröschen sind Breite und Höhe 0. Ebenfalls unveränderbar sind die Datenattribute **pool**, eine Referenz auf den Pool, in dem dieser Frosch zu Hause ist, und **id**, ein eindeutiger Identifikationsstring der Gestalt „Frog-0001“, wobei der erste erzeugte Frosch die Nummer „0001“ erhält, der zweite die Nummer „0002“ usw.

Die Geschwindigkeit, mit der ein Frosch sich im Pool bewegt, kann im Attribut **speed** festgelegt werden (Defaultwert ist 2), und zwar „stufenlos“ durch einen Fließkommawert aus dem Intervall [1;10], wobei die Geschwindigkeit mit der Größe des Wertes zunimmt. Setzt man **speed** = „max“, dann werden alle Aktionen ganz ohne künstliche Verzögerung durchgeführt – schneller geht's dann nicht mehr, jedenfalls nicht, wenn man dem Frosch beim Zeichnen zusehen möchte.

Wer darauf keinen Wert legt, der kann ihn auch tauchen lassen – dazu setzt man das Attribut **visible** auf False. Hat das speed-Attribut einen numerischen Wert, dann erreicht man dadurch nur eine geringe Geschwindigkeitssteigerung, da im wesentlichen nur die Zeit eingespart wird, die für die sichtbare Darstellung von Ausrichtungsänderungen benötigt wird. Ein unsichtbarer Frosch, bei dem speed auf „max“ gesetzt ist, geht aber ab wie eine Rakete.

Das nur lesbare Attribut **lastitem** enthält die Kennung des zuletzt gezeichneten, noch vorhandenen Objekts bzw. den Wert 0, falls kein gezeichnetes Objekt des Froschs vorhanden ist. Einsetzen kann man diese Kennung zum späteren gezielten Löschen von Objekten.

Weiterhin kann man einen Frosch anweisen, beim Zeichnen von geschlossenen Figuren diese zu füllen. Dazu muss einerseits eine Füllfarbe über **fillcolor** gesetzt sein, andererseits der Füllmodus im Attribut **fill** auf True gesetzt werden. Letzteres bewirkt, dass beginnend mit der darauf folgenden Zeichenoperation nach jeder weiteren Zeichenoperation geprüft wird ob dadurch eine geschlossene Figur entstanden ist, d.h. man wieder an den Punkt gelangt ist, von dem die erste Zeichenoperation nach **fill=True** ausgegangen ist. Tritt dieser Fall ein, dann wird die so entstandene Figur mit der gewählten Füllfarbe ausgefüllt. Das funktioniert, wenn man einen geschlossenen Streckenzug inkl. Kreisbögen an einem Stück zeichnet. In einigen Fällen wird eine geschlossene Figur auch dann als solche erkannt, wenn der Streckenzug unterbrochen und die schließende Verbindung andersherum hergestellt wird.

Sobald eine geschlossene Figur entstanden ist und ausgefüllt wurde, wird der Zeiger, der sich den Startpunkt gemerkt hat, zurückgesetzt – neuer Startpunkt für eine weitere geschlossene Figur ist wiederum der Anfangspunkt der auf den Füllvorgang folgenden Zeichenoperation. Bei Bedarf kann man diesen Zeiger durch einen erneuten Aufruf von **fill=True** auch manuell zurücksetzen – die darauf folgende Zeichenoperation markiert nun einen neuen Startpunkt für ein potentiell neues, zu füllendes Polygon. Durch ein **fill=False** lässt sich der Füllmechanismus wieder abschalten.

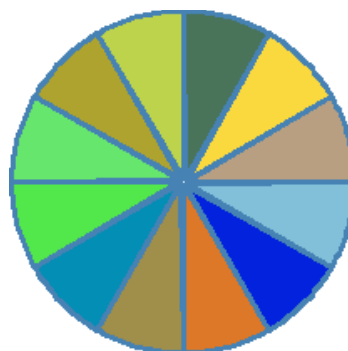
Schließlich sei an dieser Stelle noch erwähnt, dass jeder Frosch über einen eingebauten Kilometerzähler verfügt, der bei der „Geburt“ bei 0.0 beginnt und die Längen aller zurückgelegten Wege aufzeichnet – egal, ob schwimmend (move), hüpfend (jump) oder tauchend (unsichtbar). Den aktuellen „Kilometerstand“ kann man über das Attribut **way** jederzeit abfragen. Ein schreibender Zugriff auf **way** ist nur möglich, um den Zähler auf Null zurückzusetzen; der Versuch, einen anderen Wert zuzuweisen, führt zu einer Fehlermeldung.

Bemerkung am Rande: Unter Verwendung des Pool-Attributs **resolution** kann man tatsächlich einen echten Kilometerzähler daraus machen:

```
km = frog.way/pool.resolution*25.4/1000000
```

Abschließend noch ein Beispiel, das die genannten Eigenschaften in der Praxis zeigt:

```
1  from random import random
2
3  pool = Pool()
4  ben = Frog(pool)
5  ben.speed = 5
6  ben.width = 3
7  ben.color = "steelblue"
8  ben.fill = True
9  ben.visible = False
10 for k in range(12):
11     ben.fillcolor = random(),random(),random()
12     ben.move(100)
13     ben.turn(90)
14     ben.circle(100,30)
15     ben.turn(90)
16     ben.move(100)
17     ben.turn(180)
18 wegstrecke = ben.way    # hier: 3043.34 px
19 teich.ready()
```



## 3.5 Weitere Fähigkeiten

Die Methode **wait()** erwartet als Argument eine Zeitdauer in Millisekunden und hält den gesamten Programmablauf (nicht nur die Aktionen des betreffenden Froschs!) für diese Zeit an. Zum zeitgesteuerten Aufrufen bestimmter Aktionen eignet sich **wait()** in der Regel nicht. Im Rahmen von Kapitel 4 („Ereignisbehandlung“) wird eine geeignete Möglichkeit gezeigt.

Neu in Version 2.2 sind die Methoden **timer\_start()** und **timer\_stop()** zur Zeitmessung. Durch den Aufruf von **timer\_start()** wird die aktuelle Zeit genommen. Bei jedem Aufruf von **timer\_stop()** wird die seit dem letzten Aufruf von **timer\_start()** vergangene Zeit in Millisekunden als ganzzahliger Wert zurück geliefert, ohne dabei die Startzeit zurück zu setzen.

Ein Aufruf von **stamp()** bewirkt, dass ein Frosch an dieser Stelle einen Abdruck in Gestalt seiner derzeitigen Form in der aktuellen Ausrichtung und Farbe hinterlässt.

Mittels **beep()** kann ein Frosch einen kurzen Piepton von sich geben – jedenfalls theoretisch den auf vielen Computern scheint das nicht (mehr) zu funktionieren. Wer dennoch Töne hören möchte oder wem ein Piepen nicht genügt, der kann ihn mittels **sing()** eine Audiodatei (wav-Dateien; unter Linux ggf. auch mp3-Dateien) abspielen lassen, deren Dateiname als Parameter übergeben wird. Möchte man die Audiodatei im Hintergrund abspielen, ohne den Programmablauf bis zum Ende der Abspielzeit anzuhalten – das ist die Voreinstellung –, dann kann man dies durch den optionalen Schlüsselwortparameter „background=True“ beim Aufruf von **sing()** erreichen.

Da bisher noch kein Modul zur Standardbibliothek von Python gehört, mit dem sich Betriebssystem übergreifend Audiodateien abspielen lassen, ist die konkrete Implementation der **sing()**-Methode abhängig von verwendeten Betriebssystem: Unter Windows wird das Python-Modul winsound verwendet, unter Linux wird auf das Konsolen-Tool play aus dem Paket sox zurückgegriffen und unter Mac OS bleiben die Frösche gänzlich stumm. Letzteres gilt auch dann, wenn auf einem Linux-System das Paket sox nicht installiert ist. Unter Windows sollte das Abspielen einer Audiodatei immer funktionieren.

Sind die Voraussetzungen für das Abspielen eines Soundfiles nicht gegeben oder eine abzuspielende Audiodatei nicht vorhanden, dann bleibt Frosch einfach stumm.

Will man einen Frosch erzeugen, der eine identische Kopie eines schon vorhandenen Froschs ist, dann ist es mit einer einfachen Zuweisung nicht getan – das erzeugt lediglich eine Referenz und man hat dann einfach zwei verschiedene Namen für denselben Frosch. Stattdessen kann man die Methode **clone()** verwenden und erhält so einen neuen Frosch mit eigener Identität, der in allen Merkmalen identisch ist mit seinem Klonvorbild. Der geklonte Frosch befindet sich nach seiner „Geburt“ jedoch an Position (0,0) in 0°-Ausrichtung und sein „Kilometerzähler“ beginnt mit 0.0

Weiterhin verfügt jeder Frosch über zwei Methoden, um die von ihm hinterlassenen Spuren wieder zu entfernen. Ein Aufruf von **clear()** entfernt auf einen Schlag sämtliche Spuren des Froschs, ohne jedoch seine aktuelle Position zu verändern. Man kann jedoch gezielt auch nur einzelne Zeichenobjekte eines Froschs entfernen, indem man als Parameter eine einzelne Objektkennung oder auch eine Liste von Objektkennungen übergibt. An eine solche Objektkennung gelangt man, indem man unmittelbar nach der Objekterzeugung diese Kennung aus dem Attribut **lastitem** abfragt und sich für das spätere Löschen merkt. Der Versuch, ein nicht (mehr) existierendes Objekt zu löschen, führt im Übrigen nicht zu einer Fehlermeldung.

Da **lastitem** jeweils die Kennung des zuletzt gezeichneten und noch nicht gelöschten Objekts enthält, kann man durch fortgesetztes Aufrufen von **lastitem** alle gezeichneten Objekte eines Froschs Stück für Stück von hinten nach vorne löschen, z.B. so:

```
while frosch.lastitem:
    frosch.clear(frosch.lastitem)
```

Ist ein Frosch schließlich irgendwann seines Lebens im Pool überdrüssig, dann kann er ihn auch endgültig verlassen. Natürlich könnte er einfach abtauchen und sich unsichtbar machen, nachdem er alle seine Spuren beseitigt hat. Will man den Frosch aber wirklich „rückstands-frei“ entfernen, dann hilft ein `exit()` und der Frosch ist weg – ebenso wie alles, was er zuvor im Pool an Spuren hinterlassen hat. Erst durch ein `exit()` wird ein Frosch auch aus der Frosch-liste entfernt, die im Pool-Attribut `frogs` gehalten wird.

Und nun noch ein Beispiel, das die Einsatzmöglichkeiten zeigt:

```
1  from random import random
2
3  teich = Pool()
4  knut = Frog(teich)
5  knut.beep()                                # knut ist da!
6  knut.shape = "cross"
7  knut.color, knut.bodycolor = "burlywood", "chartreuse"
8  knut.move(120)
9  lars = knut.clone()
10 lars.stamp()
11 lars.sing("birthday.wav",background=True)    # lars ist auch da!
12 lars.move(-120)
13 for k in range(100,9,-10):
14     knut.color = random(), random(), random()
15     lars.color = random(), random(), random()
16     knut.dot(k)
17     lars.dot(k)
18 for k in range(10):
19     lars.wait(500)
20     lars.clear(lars.lastitem) # rückwärts schrittweise entfernen
21     lars.beep()
22 knut.wait(2000)
23 knut.clear()                               # knuts Zeichnung komplett entfernen
24 lars.exit()                                # lars verzieht sich aus dem teich
25 teich.ready()
```

## 3.6 Lesen und Schreiben

Jeder Frosch im Pool kann lesen und schreiben. Dazu bedient er sich der Methoden `read()`, `readnum()`, `readlist()`, `write()` und `writeln()`. Für die Ausgabe im Pool kann aus einer größeren Anzahl von Schriftarten ausgewählt werden, die wiederum skaliert und bei Bedarf fett und/oder kursiv gesetzt werden können. Die verfügbaren Schriftarten sind abhängig vom Betriebssystem und den auf einem konkreten System installierten Schriften. Eine sortierte Liste aller im frog-Modul verfügbaren Schriften enthält das Attribut `fonts` eines jeden Froschs.

```
1  teich = Pool()
2  Emil = Frog(teich,visible=False)
3  name = Emil.read("Wie heißt du?")
4  Emil.font = "Arial",24,"bold"
5  Emil.color = "tomato"
6  Emil.write(name)
7  Emil.color = "darkslateblue"
8  breit,hoch = Emil.textsize(name)
9  for k in range(2):
10     Emil.move(breit); Emil.turn(90)
11     Emil.move(hoch); Emil.turn(90)
12  teich.ready()
```



**Bernhard**

Die `read()`-Methode der Frösche entspricht funktional dem bekannten `input()` (Python 3). Es kann optional als Parameter ein Text mitgegeben werden, der oberhalb der Eingabezeile angezeigt wird. Zurückgeliefert wird die Eingabe des Anwenders als Zeichenkette – oder `None`, falls der Anwender statt einer Eingabe auf den [Cancel]-Knopf drückt oder das Eingabefenster „von außen“ über dessen Titelleiste schließt.

Die in Version 2.0 neu hinzugekommenen Methode `readnum()` erlaubt das „sichere“ Einlesen von Zahlenwerten, ohne dass zur Vermeidung eines Laufzeitfehlers eine Typprüfung oder Ausnahmebehandlung erforderlich wäre. Hat der Anwender einen gültigen Zahlenwert (nur float oder int) eingegeben, dann wird dieser als Zahlenwert zurückliefert. Enthält die Eingabe einen Dezimalpunkt, dann liefert `readnum()` einen Fließkommawert, ansonsten einen Integerwert. Wurde kein gültiger Zahlenwert erkannt, wird `False` zurückgeliefert.

Die Methode `readlist()` erlaubt die komfortable Eingabe beliebiger, durch Komma getrennter Werte in der Eingabezeile. Der Rückgabewert ist eine Liste dieser getrennten Werte, wobei Zahlenwerte in der gleichen Weise wie bei `readnum()` direkt umgewandelt werden.

Das Attribut `font` eines jeden Froschs kapselt die aktuell gesetzten Schriftparameter. Wurde im Programm noch keine Schrift gesetzt, dann wird ein Defaultfont verwendet, der abhängig vom konkreten System und seiner Schrifteninstallation ist. Eine Abfrage des `font`-Attributs ist also auch ohne vorherige Zuweisung möglich und führt nicht zu einem `NameError`.

Zur Festlegung einer Schrift wird dem `font`-Attribut ein Tripel aus Schriftname (z.B. „Times“, Schriftgröße (in pt) und Schriftstil („normal“, „bold“, „italic“ oder „bolditalic“) zugewiesen – die umgebenden Klammern sind dabei optional, ein ungültiger Schriftstil führt zu einer Fehlermeldung.

Alternativ kann man auch nur die Schriftart oder nur Schriftart und Größe ändern, und die übrigen Eigenschaften des aktuell gesetzten Fonts beibehalten. Möglich ist also zum Beispiel

```
schrift = "times"                # Größe und Stil bleiben erhalten
schrift = "times", 18            # Stil bleibt erhalten
schrift = "times", 14, "normal"  # alle Parameter werden gesetzt
schrift = schrift[0], schrift[1]+2 # Schrift um 2pt größer
```

Wird eine *Schriftart* oder *Schriftgröße* gewählt, die auf dem System nicht verfügbar ist, führt dies nicht zu einer Fehlermeldung, sondern es wird eine alternative Schriftart bzw. die nächstliegende passende Schriftgröße gesetzt. Welche Schriftart und -größe tatsächlich verwendet werden, kann man ermitteln, indem man sich nach der Zuweisung ansieht, welche Werte das `font`-Attribut zurückliefert. Wie alle (öffentlichen) Attribute eines Froschs ist auch das `font`-Attribut ein „managed attribute“, so dass bei der Zuweisung eine entsprechende Prüfung und ggf. Änderung der Wunschdaten erfolgt.

Der als Zahl angegebene Wert für die *Schriftgröße* wird als Maßzahl in der Einheit pt interpretiert, bei Verwendung negativer Maßzahlen als px. Fließkommawerte sind zwar möglich, werden intern aber in die nächstgelegene Ganzzahl umgewandelt.

Die *Schriftfarbe* entspricht der aktuellen Zeichenfarbe des Froschs im `color`-Attribut.

Die Methode `write()` erwartet als Parameter den Inhalt (beliebigen Typs), der einzilig auf dem Bildschirm ausgegeben wird, wobei die Position des Froschs der linken unteren Ecke der Ausgabe entspricht. Mit Hilfe der Methode `textsize()` kann ermittelt werden, welche Breite und Höhe in px der als Parameter übergebene Text hat – beides wird gemeinsam als Tupel zurückgeliefert. So kann die Ausgabe durch entsprechende Berechnung und Positionierung des Froschs z.B. auch zentriert oder rechtsbündig ausgerichtet, unterstrichen oder umrandet werden. Beide Methoden verfügen zu dem über den Schlüsselwortparameter *width*, über den eine Maximalbreite in px für die Ausgabe festgelegt werden kann. Der Text wird dann so umbrochen, dass er diese Breite nicht überschreitet.

Die Methode `writeln()` (für „write line“) ist als „Komfortfunktion“ für Anfänger gedacht und insbesondere dann sinnvoll einsetzbar, wenn der Einstieg in Python direkt über das `frog`-Modul erfolgt und keine Ein- und Ausgabe über die Konsole erfolgt. `writeln()` fügt zusätzlich einen Zeilenumbruch ein, setzt den Frosch also nach der Ausgabe linksbündig eine Zeile darunter. Die Ausgabe beginnt vertikal zentriert direkt rechts des nach Osten ausgerichteten Froschs. Nützlich ist in diesem Zusammenhang auch die (Hilfs-)Methode `topleft()`, die den Frosch (eine Zeile unterhalb der Oberkante des Pools) in die linke obere Ecke bewegt, so dass die Ausgabe mit `writeln()` ähnlich wie auf der Konsole links oben beginnen kann.

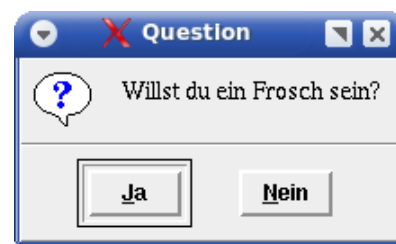


Die Methoden `textsize()`, `write()` und `writeln()` akzeptieren auch **Mehrfachausgaben**. Alle Argumente, durch Komma getrennt, werden in einer Zeile nebeneinander durch je ein Leerzeichen getrennt ausgegeben. Mit Hilfe der **Sternchenausgabe** lassen sich so auch Tupel, Listen und selbst Iteratoren wie `range()` in dieser Form ausgeben:

```
zahlen = [3,5,8,13]
frog.write(*zahlen)          # Ausgabe: 3 5 8 13
frog.writeln(*range(2,10,3)) # Ausgabe: 2 5 8
```

Ergänzend zur direkten Ausgabe von Text im Pool gibt es noch die Möglichkeit, mit Hilfe der Methode `message()` eine Meldung in einem modalen Toplevel-Widget auszugeben. Dazu übergibt man als ersten Parameter die Art der Botschaft („Info“, „Error“, „Warning“, „Question“, „Confirm“) und als zweiten Parameter den gewünschten Ausgabertext. Bei den drei ersten handelt es sich um pure Mitteilungen mit Überschrift und Symbol, deren Kenntnissnahme man durch [Ok] bestätigt. Bei „Question“ muss man sich zwischen [Ja] und [Nein] entscheiden, bei „Confirm“ zwischen [Ok] und [Abbruch]. Jeder Aufruf von `message()` liefert als Rückgabewert **True** (bei [Ok] bzw. [Ja]) oder **False** (bei [Nein] bzw. [Abbruch]). Ein kurzes Beispiel:

```
1 wiese = Pool(title="Die Wiese")
2 hase = Frog(wiese)
3 ja = hase.message("Question",
4     "Willst du ein Frosch sein?")
5 if ja:
6     hase.shape = "frog"
7 wiese.ready()
```



## 3.7 Laden und Speichern

Im Hinblick auf die Zielgruppe wurden dem Frosch auch Möglichkeiten mitgegeben, Daten einfach, d.h. ohne explizite Dateibehandlung und ohne die Notwendigkeit von Ausnahmebehandlungen auf einem Datenträger mit der Methode `save()` zu speichern und von dort mittels der Methode `load()` wieder zu laden. Dazu bedient sich der Frosch (intern) der Python eigenen pickle-Technik, die im pickle Modul implementiert ist. Alle einfachen Datentypen sowie Listen, Tupel und Dictionaries können auf diese Weise (in einem speziellen Binärformat) gespeichert und wieder geladen werden, ohne dass man sich Gedanken über die Struktur der Datei machen müsste. So lassen sich z.B. Spielstände einfach speichern und wieder einlesen.

Die Methode `save(daten, dateiname)` erwartet genau zwei Parameter: Zunächst einen Bezeichner auf ein Objekt eines der o.g. Datentypen (oder ein entsprechender Literal) und anschließend der Name der Datei, in der die Daten abgelegt werden sollen. Als Standardendung dieser Dateien ist „.frog“ vorgesehen. Hat der übergebene Dateiname *nicht* diese Endung, dann wird sie automatisch angehängt. Die Methode `save()` liefert entweder **True** (= Daten wurden erfolgreich gespeichert) oder **False** (= Daten konnten nicht gespeichert werden). Mögliche Laufzeitfehler z.B. durch fehlende Schreibrechte werden abgefangen und führen ebenfalls zur Rückgabe von **False**.

Die Methode **load(dateiname)** erwartet genau einen Parameter, und zwar den Namen der Datei, aus der Daten geladen werden sollen. Wie bei **save()** gilt: Die Endung „.frog“ wird automatisch angehängt, wenn sie nicht schon vorhanden ist. Der Rückgabewert der Methode ist das aus der Datei geladene Objekt. Der Objekttyp entspricht dem des zuvor mit **save()** gespeicherten Objekts. War der Prozess des Ladens nicht erfolgreich, etwa weil die Datei nicht existiert, dann wird **None** zurückgeliefert. Der Wert **None** lässt sich also nicht sinnvoll mit **save()** speichern (obwohl pickle dies kann), weil der Rückgabewert **None** fälschlich als missglücktes Laden interpretiert würde – eine echte Einschränkung dürfte das nicht sein.

Abschließend ein Beispiel:

```
1  spielbrett = Pool(title="Frosch ärgere dich nicht")
2  spiel = Frog(spielbrett)
3  figuren = {1:[0,3,12,17], 2:[0,0,5,11]} # Position Spielfiguren
4  gesichert = spiel.save(figuren,"Spielstand")
5  spiel.topleft()
6  if gesichert:
7      spiel.writeln("Spielstand wurde gesichert!")
8  figuren = spiel.load("Spielstand")
9  if figuren is not None:
10     spiel.writeln("Spielstand wurde geladen!")
11     for spieler in figuren:
12         spiel.writeln("Spieler",spieler,":",*figuren[spieler])
13 else:
14     spiel.writeln("Spielstand konnte nicht geladen werden!")
15 spielbrett.ready()
```

## 4. Ereignisbehandlung

Das frog-Modul verfügt über zwei grundsätzliche Möglichkeiten zur Ereignisbehandlung, d.h. zur Reaktion auf Anwenderaktionen in Gestalt von Mausklicks oder Tastatureingaben.

### 4.1 Buttons im Pool

Zum einen verfügt jeder Frosch über die Methode **button()**, deren Aufruf an der aktuellen Froschposition eine beliebig beschriftbare Schaltfläche im Pool erstellt, bei deren Betätigung eine ebenfalls frei definierbare Funktion oder Methode ausgeführt wird. Ein Beispiel:

```
1 pool = Pool()
2 tim = Frog(pool)
3 tim.pos = -100,100
4 tim.color = "royalblue"
5 tim.bodycolor = "whitesmoke"
6 tim.fillcolor = "snow"
7 tim.button("Geh nach Hause!",tim.home)
8 pool.ready()
```

Durch den Frosch *tim* wird an der Position (-100,100) ein Button mit der Aufschrift „Geh nach Hause“ generiert – wie bei Textausgaben mittels **write()** bezieht sich die Position des Froschs auf die linke untere Ecke des Buttons. Die Größe des Buttons wird automatisch an die Beschriftung angepasst, je nach verwendetem Font und seiner Größe. Diese Größe lässt sich – auch bereits vor der Erzeugung! – mit Hilfe der Methode **buttonsize()** ermitteln, die genauso funktioniert wie **textsize()** bei Textausgaben. Somit es ist es auch bei Buttons möglich, durch entsprechende Berechnung diese pixelgenau zu platzieren. Anders als alle anderen Objekte, die ein Frosch im Pool hinterlassen kann, befinden sich Schaltflächen IMMER im Vordergrund – sogar die Frösche selbst schwimmen unter ihnen her. Das mag man bedauern oder nicht, das zugrunde liegende Tkinter Canvas-Widget zwingt dieses Verhalten auf.

Frei gestalten kann man neben der verwendeten Schriftart, die im **font**-Attribut des Froschs abgelegt ist, auch die Farbe der Beschriftung (**color**-Attribut), die Farbe des Hintergrundes (**bodycolor**-Attribut) sowie die Hintergrundfarbe beim Überfahren (nur Linux) und Anklicken des Buttons mit der Maus (**fillcolor**-Attribut). Wurden diese nicht explizit gesetzt, werden Defaultwerte verwendet.

Beim Aufruf der **button()**-Methode erwartet diese genau zwei Parameter. Der erste Parameter ist eine Zeichenkette für die Beschriftung des Buttons, der zweite der Event-Handler, also eine Funktion oder Methode, die ausgeführt wird, wenn die Schaltfläche gedrückt wird. Wichtig: Die Funktionsklammern müssen weggelassen werden – es soll ja die Funktion selbst übergeben werden, nicht ihr Rückgabewert! Ohne weitere Tricks verwendbar als Event-Handler sind alle Funktionen oder Methoden, die *genau einen Parameter* erwarten – an diesen wird das **Event-Dictionary** beim Klick auf den Button übergeben.

Das Event-Dictionary erhält beim Eintreten eines Ereignisses – auch bei den im nächsten Abschnitt behandelten Maus- und Tastatur-Events – alle relevanten Informationen dieses Ereignisses. Bei einem Buttonklick beschränken sich diese Informationen auf den Event-Typ (dieser lautet „ButtonWidget“) und die Beschriftung der angeklickten Schaltfläche, die man unter dem Schlüssel „name“ im Event-Dictionary findet. Dazu ein kleines Beispiel:

```
1 def zeige_aufschrift(event):
2     print(event["name"])
3
4 teich = Pool()
5 knopf = Frog(teich, visible=False)
6 knopf.button("Start", zeige_aufschrift)
7 knopf.pos = 3, -36
8 knopf.button("Stop", zeige_aufschrift)
9 teich.ready()
```



Durch Verwendung von Schlüsselwortparametern ist es auch möglich, Funktionen und Methoden als Event-Handler einzusetzen, die mehr oder weniger als *einen* Parameter haben. Der Einsatz von lambda-Konstruktionen erlaubt es außerdem, einem Event-Handler beim Aufruf beliebige Parameter mitzugeben, die im Handler ausgewertet werden können.

Da beide Varianten nicht nur für Buttons, sondern für alle Event-Typen gelten, findet man ein entsprechendes Beispiel in Abschnitt 4.3 („Reagieren auf Ereignisse“).

## 4.2 Lauschen auf Ereignisse

Wesentlich mächtiger, aber auch komplexer als die Handhabung von Schaltflächen ist die Möglichkeit, auf Maus-, Tastatur- und zeitgesteuerte Ereignisse zu reagieren. Dazu dient die Methode **listen()**, die sowohl als Pool-Methode als auch als Frog-Methode verfügbar ist und mit Hilfe derer man festlegt, dass als Reaktion auf ein bestimmtes, festzulegendes Ereignis, eine ebenfalls festzulegende Funktion oder Methode aufgerufen werden soll.

Im Unterschied etwa zur Python-Funktion `input()` oder der Frog-Methode `read()` wird der Programmablauf durch **listen()** nicht angehalten – das Lauschen erfolgt unmerklich im Hintergrund. Je nachdem, ob man die Pool-Methode **listen()** oder die gleichnamige Frog-Methode verwendet, lauscht der gesamte Pool oder nur ein einzelner Frosch auf ein bestimmtes Ereignis, wobei Frösche sinnvollerweise nur auf Mausereignisse reagieren können.

Achtung: Auf Windows-Systemen (evtl. auch unter MacOS) können Frösche nicht (mehr) auf das Mausexplorer lauschen! Das ist der fehlerhaften Implementation der tkinter `tag_bind()`-Methode für `MouseWheel`-Events geschuldet.

Beim Aufruf von **listen()** müssen (mindestens) zwei Parameter übergeben werden: eine Event-Sequenz und ein Event-Handler. Die **Event-Sequenz** ist eine Zeichenkette, die angibt, auf welches Ereignis bzw. welche Ereignisse reagiert werden soll. Der **Event-Handler** ist wie bei der **button()**-Methode eine Methode oder Funktion mit *genau einem Parameter*, über den das Event-Dictionary mit allen relevanten Daten an den Event-Handler übergeben wird.

## 4.2.1 Aufbau einer Event-Sequenz

Eine **Event-Sequenz** sieht allgemein so aus: "<modifier-type-detail>".

Dabei sind „modifier“ und „detail“ optionale Angaben, die das Ereignis genauer spezifizieren. Eine Event-Sequenz besteht also mindestens aus der Angabe des *Event-Typs*, eingefasst in spitze Klammern. Es gibt zwar für einige Event-Typen auch Abkürzungen gemäß dem Tkinter-Standard, auf diese werde ich aber hier nicht eingehen.

Genau zu beachten ist die Schreibweise innerhalb der Event-Sequenz. Event-Sequenzen, die andere als die in der Tabelle aufgeführten Typen spezifizieren, führen zu einer Fehlermeldung.

Gleiches gilt für den Versuch, einen Frosch auf ein anderes als ein mausbezogenes Event lauschen zu lassen. Weitergehende Informationen zu gültigen Event-Sequenzen in Tkinter findet man bei Shipman (<http://infohost.nmt.edu/tcc/help/pubs/tkinter/event-types.html>).

| Event-Typ     | Event-Auslöser            |
|---------------|---------------------------|
| Key           | Drücken einer Taste       |
| KeyRelease    | Loslassen einer Taste     |
| Button        | Klick auf eine Maustaste  |
| ButtonRelease | Loslassen einer Maustaste |
| Enter         | Betreten des Mausursors   |
| Leave         | Verlassen des Mausursors  |
| MouseWheel    | Drehen am Mauseurad       |
| Motion        | Bewegung der Maus         |
| Timer         | Zeitgesteuerte Aktion     |
| Close         | Pool über [X] schließen   |

Der optionale **Modifier** gibt z.B. an, ob zusätzlich zu einer bestimmten (Maus-)Taste noch die <Strg>-Taste gedrückt sein muss oder ob nur auf einen Doppel- oder Dreifachklick mit der Maus reagiert werden soll. Es werden u.a. folgende Modifier unterstützt, die natürlich inhaltlich zum Event-Typ passen müssen: „Double“ und „Triple“ sind z.B. nicht kombinierbar mit Key-Events. Bei Timer-Events besteht der Modifier aus einem selbst gewählten Namen, der nach den Regeln für Python-Bezeichner gebildet werden kann. Dies ist dann erforderlich, wenn ein bestimmter Timer später gelöscht werden soll. Ist das nicht beabsichtigt, ist kein Modifier erforderlich. Das Close-Event erlaubt keinen Modifier.

| Modifier | Event-Auslöser     |
|----------|--------------------|
| Any      | Beliebige Taste    |
| Alt      | [Alt]-Taste        |
| Control  | [Strg]-Taste       |
| Shift    | [Umschalt]-Taste   |
| Double   | Maus-Doppelklick   |
| Triple   | Maus-Dreifachklick |
| B1 .. B3 | Button bei Motion  |
| <Name>   | Timer              |

Schließlich kann ein Event durch eine optionale **Detail**-Angabe noch präzisiert werden. Bei einem *Mausklick* kann dies die Angabe der Maustaste sein (1 = links, 2 = Mitte, 3 = rechts). Bei einem *Timer-Event* gibt man als Detail die Zeit in Millisekunden an, die bis zum Aufruf des Handlers gewartet werden soll. Die Event-Typen *MouseWheel*, *Motion*, *Enter*, *Leave* und *Close* kennen keine Detail-Angabe.

Bei einem *Tastatur-Event* ist zu unterscheiden zwischen Tasten, die ein sichtbares Zeichen auf dem Bildschirm erzeugen, und solchen Tasten, die dies nicht tun. Während Erstere einfach durch die Angabe des entsprechenden Zeichens spezifiziert werden, gibt es für letztere jeweils eine spezielle Tastenbezeichnung. Die folgende Tabelle enthält nur eine Zusammenstellung einiger wichtiger Tasten. Die Tastenbezeichnungen für die übrigen Tasten kann man mit Hilfe des nachfolgenden kleinen Programms selbst ermitteln:

```

1  # Ermittlung der Tastenbezeichnungen
2  p = Pool()
3  p.listen("<Key>", lambda e:
4      \Frog(p).message("Info", e["name"]))
5  p.ready()

```

Um die Bindung zwischen einem Event und dem mittels **listen()** zugeordneten Event-Handler zu lösen, ruft man die Methode **listen()** erneut auf, und zwar mit der gewünschten Event-Sequenz und **None** als Handler, z.B.

```

frog.listen("<Button-1>", None)
frog.listen("<Anfang-Timer>", None)

```

Ab sofort erfolgt keine Reaktion mehr auf einen Klick mit der linken Maustaste und der Timer namens „Anfang“ wird abgeschaltet. Fehlt beim Timer-Event der Modifier, dann werden alle gesetzten Timer abgeschaltet. Hier einige Beispiele für zulässige Event-Sequenzen:

| Event-Sequenz      | Reaktion auf   |
|--------------------|--|
| <Key>              | Beliebigen Tastendruck                                   |
| <Key-A>            | Taste [A], d.h. [Umschalt]-[A] (Großbuchstabe!)          |
| <Alt-KeyRelease-5> | Loslassen der Tastenkombination [Alt]-[5]                |
| <Button>           | Beliebigen Mausklick                                     |
| <Enter>            | „Eintreten“ der Maus in den Bereich des Pools oder Frogs |
| <Control-Button-3> | Rechten Mausklick bei gedrückter [Strg]-Taste            |
| <B2-Motion>        | Mausbewegung mit gedrückter linker Maustaste             |
| <Shift-MouseWheel> | Drehung am Mausrad bei gedrückter [Umschalt]-Taste       |
| <Timer-2000>       | Timer: Ausführung nach 2 Sekunden                        |
| <Start-Timer-500>  | Timer „Start“: Ausführung nach ½ Sekunde                 |

| Detail    | Taste         |
|-----------|---------------|
| Control_L | [Strg] links  |
| Control_R | [Strg] rechts |
| BackSpace | Löschen [↵]   |
| Delete    | [Entf]        |
| Down      | [↓]           |
| Left      | [←]           |
| Right     | [→]           |
| Up        | [↑]           |
| Escape    | [Esc]         |
| space     | Leertaste     |
| F1 .. F12 | [F1] .. [F12] |
| Next      | [Bild ↓]      |
| Prior     | [Bild ↑]      |
| Return    | [↵]           |
| Tab       | [↹]           |

## 4.2.2 Spezialfälle und Stolpersteine

Während es ohne weiteres möglich ist, ein und denselben Event-Handler von mehreren Events aus anzusteuern, ist dies per Voreinstellung nicht so, wenn man einem bestimmten Event (Event-Sequenz) verschiedene Handler zuordnen will. Dann gilt nämlich: Die zuletzt vorgenommene Zuordnung ist die einzig gültige – eine eventuell zuvor vorgenommene Zuordnung zu einem anderen Handler wird überschrieben. Unter Verwendung des optionalen Schlüsselwortparameters `add` der Methode `listen()` kann dieses Verhalten geändert werden: Durch Angabe von `add=True` wird ein neu zugeordneter Handler *ergänzt*, die bisherige Zuordnungen bleiben erhalten. Dazu ein kurzes Beispiel:

```
1  def drehdich(event):
2      if ben.visible: ben.turn(720)
3
4  def versteckdich(event):
5      ben.visible = not ben.visible
6
7  teich = Pool()
8  ben = Frog(teich)
9  teich.listen("<Button-2>",drehdich)
10 teich.listen("<Button-2>",versteckdich,add=True)
11 teich.ready()
```

Beim Klick mit der mittleren Maustaste dreht sich *ben* zweimal um sich selbst, bevor er verschwindet. Ein weiterer Klick macht ihn wieder sichtbar – aber er dreht sich nicht. Erst beim nächsten Klick dreht er sich erneut und verschwindet dann wieder usw. Hätte man auf die Angabe `add=True` verzichtet, dann hätte *ben* bei jedem Klick mit der mittleren Maustaste nur seinen Sichtbarkeitsstatus geändert, gedreht hätte er sich gar nicht, weil die Bindung an die Funktion `drehdich()` durch die nachfolgende Zuordnung zu `versteckdich()` überschrieben worden wäre.

Um zu erreichen, dass *ben* das gleiche Verhalten nur dann zeigt, wenn man mit der Maus auf seine Gestalt im Pool klickt (und nicht, wenn man irgendwo anders im Pool klickt), liegt es nahe, nicht den *teich*, sondern *ben* lauschen zu lassen, also an Stelle der Pool-Methode die gleichnamige Frog-Methode zu verwenden. Das funktioniert jedoch in obigem Beispiel nicht ohne weiteres, weil *ben* nach dem ersten Rechtsklick unsichtbar wird – ein erneutes Klicken auf seine Gestalt ist also gar nicht möglich, er taucht nicht wieder auf. Unerheblich ist es hingegen, ob ein Frosch beim Aufruf der `listen()`-Methode gerade sichtbar oder unsichtbar ist – die vorgenommene Bindung ist dennoch gültig und tritt sofort in Kraft, wenn der Frosch sich im Pool zeigt.

Das folgende Beispiel zeigt einen rechteckigen Frosch, der beim Überfahren mit der Maus seine Farbe ändert und einen dicken Rand bekommt, und nach jedem Klick mit der linken Maustaste sich jeweils 45° weiter um die eigene Achse dreht. Außerdem führt das Klicken bzw. Loslassen der Maustaste zum Wechsel der Hintergrundfarbe.

```

1  # Button-Simulation mit Hintergrundfarbwechsel
2
3  def handler(event):
4      rufer = event["object"] # Referenz d. aufrufenden Frog-Instanz
5      umgebung = rufer.pool   # Referenz auf den Pool des Froschs
6      if event["type"] == "Enter":
7          rufer.bodycolor = "red"
8          rufer.borderwidth = 5
9      if event["type"] == "Leave":
10         rufer.bodycolor = "green"
11         rufer.borderwidth = 2
12     if event["type"] == "Button":
13         umgebung.bgcolor = "yellow"
14     if event["type"] == "ButtonRelease":
15         umgebung.bgcolor = "white"
16         rufer.turn(45)
17
18     feld = Pool()
19     knopf = Frog(feld)
20     knopf.shape = "Knopf", (-40,-20),(40,-20),(40,20),(-40,20)
21     knopf.bodycolor = "green"
22     knopf.listen("<Button-1>",handler)
23     knopf.listen("<ButtonRelease-1>",handler)
24     knopf.listen("<Enter>",handler)
25     knopf.listen("<Leave>",handler)
26     feld.ready()

```

Das Beispiel zeigt auch, wie man über den Schlüssel „object“ des Event-Dictionaries denjenigen Frosch ansprechen kann, der für den Aufruf des Event-Handlers verantwortlich ist. Durch die im Event-Dictionary mitgelieferte Referenz auf die aufrufende Frog-Instanz benötigt man keine Kenntnis eines Namens dieses Froschs (hier also „knopf“), um auf diesen Frosch zugreifen zu können. Über das Datenattribut pool des Frosches kann man sogar auf den den Frosch umgebenden Pool zugreifen, ohne dessen Namen zu kennen.



### 4.2.3 Reagieren auf Ereignisse

Bisher wurde vor allem über die Event-Sequenzen und deren vielfältige Möglichkeiten gesprochen. Abschließend geht es nun darum, wie man einen Event-Handler zweckmäßig programmieren kann, welche Daten des Events an den Handler übermittelt werden und wie man sie auswerten kann.

Alle Event-Handler benötigen genau *einen* Parameter. Über diesen wird beim Aufruf als Reaktion auf ein eingetretenes Ereignis ein **Event-Dictionary** übergeben. Alle Schlüssel dieses Dictionaries sind Zeichenketten, die Werte Zeichenketten bzw. Zahlentupel. Ist ein bestimmter Schlüssel aufgrund des Event-Typs nicht mit einem Wert belegt, dann ist der Wert eine leere Zeichenkette. Folgende Einträge enthält ein Event-Dictionary:

| Schlüssel | Wert  |
|-----------|---|
| "object"  | Referenz auf die aufrufende Pool- oder Frog-Instanz   |
| "type"    | Event-Type gemäß obiger Tabelle als String  |
| "name"    | Bei Schaltfläche: Beschriftungstext der Schaltfläche<br>Bei Tastendruck: Name der Taste gemäß obiger Tabelle<br>Bei Mausklick: Name der Maustaste („Button-1“, „Button-2“, „Button-3“)<br>Bei Mausrad: „Up“ oder „Down“, je nach Drehrichtung<br>Bei Timer-Event: Name des Timers, falls festgelegt |
| "char"    | Bei darstellbaren Zeichen eines Key-Events dieses Zeichen   |
| "pos"     | Aktuelle Mausposition bei Key- und Maus-Events als Tupel (x,y)  |
| "time"    | Bei Key- und Maus-Events ein Integerwert, der der Zeit des Aufrufs in ms entspricht. Als relativer Wert brauchbar, um Event-Abstände zu ermitteln.  |

Zu den **Timer-Events** ist noch anzumerken, dass man sich auf die Genauigkeit der festgelegten Zeit nicht verlassen kann. Zwar wird der Event-Handler nicht *vor* Ablauf der angegebenen Zeit aufgerufen, aber je nach Systemauslastung kann es auch später werden.

Grundsätzlich eignen sich Timer-Events auch dazu, Frösche zu autarken Wesen zu machen, die ohne weitere Anweisung eine bestimmte Bewegung oder Tätigkeit fortlaufend verrichten. Da ein Timer-Event aber nur ein einziges Mal – nach der festgelegten Zeit – den vorgesehenen Handler aufruft, bedient man sich eines Tricks: Der Handler enthält wiederum einen Timer gesteuerten Aufruf seiner selbst.

Bei solchen Event-Handleern, die sich selbst über ein Timer-Event wieder aufrufen, kann es indes passieren, dass man mit dem Löschen nicht mehr hinterher kommt, falls die Zeitabstände zu kurz gewählt wurden.

Eine gewollte Eigenart von Events ist die, dass unmittelbar nach dem Eintreten des Events eine Reaktion erwartet wird. Dies kann allerdings *dann* zu ungewollten Ergebnissen führen, wenn ein Frosch gerade in Aktion ist, etwa um eine geometrische Figur zu zeichnen. Wird er mitten in diesem Prozess durch ein eingetretenes Ereignis vom Handler angewiesen, etwas anderes zu tun, bricht er seine aktuelle Aktion ab und tut dies – die Figur bleibt unfertig.

Um dies zu vermeiden, kann man über das Frosch-Attribut **active** abfragen, ob ein Frosch gerade in Aktion ist bzw. über das Pool-Attribut **action**, ob irgendein im Pool lebender Frosch gerade in Aktion ist. Diese Attribute liefern **True** im Falle einer Aktion, ansonsten **False**. Durch eine entsprechende Abfrage zu Beginn eines Event-Handlers kann man auf diese Weise erreichen, dass der Handler nur dann abgearbeitet wird, wenn ein bestimmter Frosch oder alle Frösche gerade Pause haben.

Möchte man eine Funktion oder Methode als Event-Handler verwenden, die mehr oder weniger als einen Parameter haben soll, dann gibt es auch dafür Lösungen.

**Fall 1:** Eine Funktion soll nicht nur als Event-Handler eingesetzt werden, sondern auch aus anderem Zusammenhang heraus ohne Parameter aufgerufen werden.

Lösung: Man verwendet für den obligatorischen Parameter des Handlers einen Schlüsselwortparameter mit Defaultwert (z.B. `event=None`).

**Fall 2:** Eine Funktion soll nicht nur als Event-Handler eingesetzt werden, sondern auch aus anderem Zusammenhang heraus mit mehreren Parametern aufgerufen werden.

Lösung: Man verwendet für diese Argumente Schlüsselwortparameter mit Vorgabewerten.

**Fall 3:** Beim Aufruf des Event-Handlers soll nicht nur das automatisch mitgelieferte Event-Dictionary übergeben werden, sondern zusätzlich weitere Parameter.

Lösung: In solchen Fällen muss man etwas tiefer in die Python-Trickkiste greifen und sich eines **lambda**-Ausdrucks bedienen. Dieses Konstrukt der Programmiersprache Python wird in der Regel nicht Bestandteil eines Einführungskurses in der Mittelstufe sein, so dass zu überlegen ist, ob man Schülern diese Möglichkeit überhaupt zeigt. Zur Not wird es eben wie eine Black-box eingesetzt. So ähnlich werde ich es hier auch halten – wer mehr über **lambda**-Ausdrücke wissen möchte, den verweise ich auf die einschlägige Dokumentation. Das folgende Beispiel zeigt, wie man sie zur Ereignisbehandlung einsetzen kann:

```
1  def swim(event,d):
2      max.move(d)
3
4  teich = Pool()
5  max = Frog(teich)
6  dist = max.read("Wie viel Pixel weit soll ich schwimmen?")
7  max.jumpto(-60,50)
8  max.button("Schwimm "+dist+" px",lambda e:swim(e,float(dist)))
9  max.jumpto(-200,0)
10 max.color = "peru"
11 teich.ready()
```

Zum Schluss noch ein etwas längeres Beispiel zur Ereignisbehandlung:

```

1  from frog import *
2  from random import randrange
3
4  def fall(event):
5      if not teich.action and teich.closeit:
6          teich.listen("<Fall-Timer>",None)
7          teich.close()
8          return
9      if event["name"] == "Button-3":
10         teich.closeit = True # siehe Kommentar unten
11         lars.message("Info","Gleich ist Schluss!")
12         return
13         lars.visible = False
14         lars.speed = "max"
15         lars.pos = randrange(-150,150),200
16         lars.angle = -90
17         lars.bodycolor = "linen"
18         lars.visible = True
19         lars.speed = 1
20         lars.jump(400)
21         teich.listen("<Fall-Timer-2000>",fall)
22
23  def fang(event): # Treffer
24      lars.beep()
25      lars.bodycolor = "red"
26
27  teich = Pool()
28  teich.cursor = "cross"
29  teich.closeit = False # dynamisch ergänztes Attribut statt global!
30  lars = Frog(teich)
31  lars.shape = "turtle"
32  lars.color = "darkseagreen"
33  lars.speed = 1
34  lars.turn(450)
35  lars.listen("<Button-1>",fang)
36  teich.listen("<Fall-Timer-1000>",fall)
37  teich.listen("<Button-3>",fall)
38  teich.ready()

```

Das Programm beginnt damit, dass ein Frosch in Schildkrötengestalt in der Mitte des Pools eine Drehung um sich selbst macht und dann zunächst verschwindet. Danach erscheint alle 2 Sekunden eine Schildkröte am oberen Rand des Pools und schwimmt zum unteren Rand. Schafft man es, die Schildkröte in dieser Zeit mit einem (linken) Mausklick zu treffen, färbt sie sich rot und gibt einen Piepton von sich. Klickt man mit der rechten Maustaste in den Pool, wird das Programm nach einer kurzen Information beendet.

In Zeile 21 haben wir den Fall eines sich selbst aufrufenden Handlers. Diese Zeile bewirkt, dass alle 2 Sekunden die Schildkröte erneut von oben nach unten durch den Pool schwimmt.