Einführung in die Programmierung

`mit Python und gtk`

Victor Hahn

Version 0.8.1 – letztes Update: 2. Oktober 2013

Kontakt: info@victor-hahn.de

Inhaltsverzeichnis

1 Einleitung

Herzlich willkommen in der Welt der Informatik! Dieses Script soll dir die Möglichkeit geben, (fast) alles, was wir im Unterricht besprochen haben, noch einmal nachzulesen. Wahrscheinlich findest du hier auch ein paar Informationen mehr, als wir im Unterricht gemacht haben – die Darstellung ist etwas „theoretischer“, um fachlich korrekt zu sein. Lass dich davon bitte nicht verwirren.

Informatik ist „die Lehre von der automatischen Verarbeitung von Information“. Das macht man natürlich mit Computern – und auf diesen Computern laufen Programme. Deswegen beschäftigen wir uns in diesem Kurs mit dem Programmieren, also dem Schreiben von Programmen.

Dieses Script ist noch nicht fertig! Es wird im Laufe der Zeit wachsen und neue Kapitel werden hinzukommen. Das bedeutet auch, dass es wahrscheinlich nicht frei ist von Fehlern und anderen hässlichen Dingen. Wenn du einen Fehler findest oder einfach etwas, was man besser hätte erklären können, sag mir bitte Bescheid. Danke!

An einigen Stellen findest du in diesem Script Fußnoten[1]. Manchmal erkläre ich Dinge etwas „zu einfach“, damit wir uns nicht in Details verfangen. Immer, wenn ich das tue und dabei das Gefühl habe, etwas eigentlich Wesentliches weggelassen zu haben, weise ich darauf in einer Fußnote hin. Du kannst diese Fußnoten lesen, wenn dich die Details interessieren – oder sie einfach überspringen. Für diesen Kurs sind sie nicht wichtig.

1.1 Was ist eigentlich ein Programm?

Sicher kennst du viele verschiedene Programme und hast ein Verständnis davon, was das eigentlich ist. Wir benutzen zum Beispiel einen Browser wie Firefox, Internet Explorer oder Chrome, um im Internet zu surfen – oder ein Textverarbeitungsprogramm wie Microsoft Word oder OpenOffice Writer. Was haben alle diese Programme gemeinsam?

Ein Programm ist eine Folge von Befehlen an den Computer – es sagt ihm, was er zu tun hat. Wenn du zum Beispiel in deinen Browser eine Internetadresse eingibst, steuert er deinen PC mit Befehlen so, dass er die gewählte Website für dich aus dem Internet zieht.

Die meisten Programme machen aber mehr, als deinem Computer nur einen Befehl nach dem anderen zu geben und irgendwann damit fertig zu sein. Sie „reden“ mit dem Benutzer – fragen dich, was du tun möchtest und zeigen dir dann (hoffentlich) das gewünschte Ergebnis an. Im Browser gibst du zum Beispiel erst oben in der Adressleiste eine Adresse ein und weiter unten bekommst du kurz darauf die Website zu sehen. *Fenster* sind also das Mittel, womit heute die meisten Programme am Rechner mit uns kommunizieren.

Es gibt aber auch noch andere Möglichkeiten. Früher waren die meisten Programm *textbasiert* – sie haben Text ausgegeben, der Benutzer musste (manchmal) Text eingeben.

Und wie muss so ein Programm jetzt aussehen, damit der Computer es versteht?

Computer sind eigentlich ziemlich dumme Maschinen. Sie kennen nur zwei verschiedene Zustände: Strom an oder Strom aus, 1 oder 0. Wissenschaftlicher ausgedrückt sagt man, Computer arbeiten mit Binärcode. Ein Programm muss für den Computer also in dieser binären

[1] Ich bin eine Fußnote! ^^

Form vorliegen – also als Folge von Einsen und Nullen. Für Menschen ist diese Form sehr umständlich und schwer verständlich. Deshalb gibt es Programmiersprachen: Sie erlauben uns, Programme in einer Form einzugeben, in der wir sie verstehen – als Text. Das ist dann immer noch eine Folge von Befehlen, aber aus Befehlen, die so logisch sind, dass wir damit umgehen können. Ein anderes Programm auf dem Computer, der *Interpreter*[2], übersetzt dann das von uns geschriebene Programm in die Einsen und Nullen, die der Computer versteht.

1.2 Was wir vorhaben

Wir werden in diesem Kurs vor allem grafische Programme schreiben – also Fenster und andere grafische Elemente programmieren. Dafür verwenden wir die Programmiersprache *Python* sowie ein System namens *gtk*. gtk ermöglicht es uns, grafisch zu programmieren, ohne jedes Mal das Rad neu zu erfinden. Wir müssen nicht erst programmieren, was überhaupt ein Fenster oder ein Button ist – gtk stellt uns diese Elemente als fertige Bausteine zur Verfügung.

gtk steht übrigens für „GIMP Toolkit“: Das Bildbearbeitungsprogramm GIMP, die Desktopumgebung GNOME und viele andere Programme wurden mit gtk gemacht. Wir lernen also mit einem System, das auch wirklich verwendet wird.

1.3 Unser erstes Python-Programm

Zum Aufwärmen schreiben wir zunächst ein einziges textbasiertes Programm – also eines der „alten“ Sorte. Dieses Programm macht nichts Anderes, als „hallo, Welt“ auf den Bildschirm zu schreiben. Es ist eine Art Tradition, dieses kleine Programm als erstes zu schreiben, wenn man eine Programmiersprache lernt.

Um ein Programm eingeben zu können, brauchen wir einen Text-Editor – also ein Programm, mit dem wir einfachen Text eingeben und speichern können. Praktisch alle Betriebssysteme liefern so ein einfaches Programm mit. Unter Windows findest du es unter Start → Programme → Zubehör → Editor.

Wenn dieses Script im Folgenden auf die Bedienung des Rechners eingeht, werden wir uns auf die Rechner in unserem Informatikraum beschränken – also Windows mit der von mir eingerichteten Python-Umgebung. Natürlich kann man auch unter fast allen andere Systemen, wie zum Beispiel Linux oder Mac OS X, mit Python programmieren. Vielleicht fällt euch beim Lesen dieses Scripts auf, dass ich die Screenshots unserer Programme auf meinem Linux-Rechner und nicht mit Windows erstellt habe. Wenn du Unterstützung dabei brauchst, Python auf deinem eigenen Rechner einzurichten, sprich mich einfach an.

Im Editor geben wir jetzt folgendes ein:

```
print "hallo, Welt"
```

Der Befehl print (englisch „drucken“) sorgt dafür, das alles, was danach in Anführungszeichen geschrieben wird, auf dem Bildschirm geschrieben wird (also sozusagen gedruckt wird, nur eben auf den Bildschirm und nicht auf Papier).

[2]Python verwendet einen Interpreter, wärend viele andere Programmiersprachen diese Aufgabe durch einen *Compiler* erledigen lassen. Der Unterschied ist, dass der Interpreter das Programm Stückchen für Stückchen übersetzt, während es schon läuft – der Compiler dagegen einmal das komplette Programm übersetzen muss, bevor man es starten kann.

Das war's schon! Dieses Programm können wir jetzt mit Klick auf *Datei* → *Speichern unter* abspeichern. Um dem Rechner mitzuteilen, dass es sich um ein Python-Programm und nicht etwa um einfachen Text handelt, muss der Name der Datei mit .py aufhören. Wir könnten das Programm also zum Beispiel als HalloWelt.py speichern.

Mit einem Doppelklick auf diese Datei, die wir eben erzeugt haben, startet unser Programm. Es zeigt „hallo, Welt" an – und ist dann auch schon wieder zu Ende. Auf den Rechnern in der Schule habe ich euch eingerichtet, dass zusätzlich ein kurzer Text angezeigt wird, der euch sagt, dass das Programm zu Ende ist und ihr das Fenster jetzt schließen könnt.

2 Objekte und Fenster

Nach der Aufwärmübung im letzten Kapitel gehen wir über zu modernen, grafischen Programmen. Als ersten Schritt dazu werden wir ein Programm schreiben, das ein Fenster anzeigt – zunächst nur ein leeres Fenster. Dazu öffnen wir wieder den Editor. Zunächst einmal müssen wir Python mitteilen, dass wir mit dem gtk-System programmieren möchten. Dazu beginnen wir unser Programm mit der Zeile:

```
import gtk
```

Das sorgt dafür, dass Python das gtk-System lädt und sozusagen in unser Programm mit rein packt. Mit diesem Befehl werden ab jetzt alle unsere Python-Programme anfangen.

Wie schon erwähnt, müssen wir nicht das Rad neu erfinden: Dinge wie Fenster bekommen wir als fertige Bausteine von gtk. Diese Bausteine nennen wir Objekte.

Für Informatiker besteht die Welt aus Objekten. Objekte sind für Informatiker nicht viel anders als das, was wir auch vom Gefühl her als Objekte bezeichnen würden. Jeder Tisch in unserem Computerraum ist zum Beispiel so ein Objekt; genauso wie jeder der Computer, die auf den Tischen stehen, ein einzelnes Objekt ist. Jede einzelne Lampe an der Decke, jeder Heizkörper an der Wand ist ein Objekt und schließlich ist auch der Computerraum als ganzes – du errätst es – ein Objekt.

2.1 Konstruieren von Objekten

Ein Fenster ist auch ein Objekt! Und so erzeugen wir uns Objekte in Python:

1. Zuerst braucht jedes Objekt einen eindeutigen Namen. Den kannst du dir frei aussuchen. Einzige Einschränkung: Leerzeichen, Umlaute und viele Sonderzeichen dürfen im Namen nicht drin sein. Er muss mit einem Buchstaben anfangen. Ansonsten ist es aber völlig egal, ob wir unser Objekt mein_Fenster, komisches_eckiges_Teil oder Paula nennen.
2. Hinter den Namen des neuen Objekt kommt ein Gleichheitszeichen.

Programm 1: Das wohl einfachste Python-Programm

```
1 print "hello, world"
```

3. Und zuletzt kommt der wichtigste Teil: Wir müssen angeben, was für eine Art von Objekt wir erstellen möchten. Wir wollen ein Fenster – auf Englisch „Window“. Ein solches Window-Objekt kann das gtk-System für uns erstellen. Also „reden“ wir mit diesem System, indem wir erst „gtk“ aufschreiben, direkt dahinter einen Punkt und dann die Art des gewünschten Objekts: „Window“. Dieser Punkt hinter gtk bedeutet, dass wir mit dem gtk-System kommunizieren – also soviel wie, „gtk, ich Rede mit dir!“. Wir wollen, dass gtk etwas für uns tut – nämlich ein neues Window-Objekt erstellen.

   Zum Schluss brauchen wir noch eine öffnende und eine schließende runde Klammer am Ende unseres Befehl. Immer, wenn wir möchten, dass irgendein System oder Ding etwas für uns tut, brauchen wir diese Klammern am Ende – dazu in Kürze mehr.

Zusammenfassend brauchen wir also einen Befehl wie den folgenden, um uns ein neues Fenster-Objekt zu erstellen:

```
mein_Fenster = gtk.Window()
```

Einen solchen Befehl, der ein neues Objekt erstellt, nennen wir *Konstruktion*.

Was fehlt noch zum Fenster?

Würden wir das Programm jetzt speichern und ausführen, wären wir enttäuscht. Von einem Fenster ist so noch nichts zu sehen. Zwar haben wir uns ein *Window*-Objekt mit dem Namen *mein_Fenster* erstellt – dieses ist aber noch nicht sichtbar. Nur, weil ein Objekt existiert, wird es uns noch nicht automatisch am Bildschirm angezeigt. Unser neues Objekt mein_Fenster kann sich aber durchaus anzeigen – wir sagen, es hat die *Funktion* dazu.

Genau wie mit dem gtk-System können wir auch mit unserem neu erzeugten Objekt „reden“. Hierzu schreiben wir hinter den Namen unseres Objektes einen Punkt und dann die gewünschte Funktion. Rufen wir nun als die Funktion *show()* (englisch „anzeigen”) unserer Objekts mein_Fenster auf:

```
mein_Fenster.show()
```

Unser Programm besteht jetzt also aus drei Befehlen und sieht wie folgt aus:

```
import gtk
mein_Fenster = gtk.Window()
mein_Fenster.show()
```

2.2 Hilfe, mein Programm stirbt!

Im Prinzip sind wir damit fertig: Wir haben mit dem ersten Befehl das gtk-System aktiviert, mit dem zweiten ein Fenster-Objekt erzeugt und mit dem dritten dieses neu erstellte Fenster sichtbar gemacht. Wenn wir dieses Programm nun speichern und ausführen, stellen wir jedoch fest, dass das Fenster nur kurz aufblitzt und sofort wieder verschwindet. Warum ist das so?

Erinnern wir uns an unsere Definition eines Programms: Ein Programm ist eine Folge von Befehlen. Unser Programm hat im Moment drei Befehle. Hat es diese abgearbeitet, ist es beendet – und ein beendetes Programm schließt sich.

Wir müssen also verhindern, dass unser Programm sich beendet. Im gtk-System ist hierzu eine besondere Lösung vorgesehen: Ganz am Ende übergeben wir die Kontrolle unseres Programms an gtk. Hierzu stellt uns das gtk-System die Funktion *main* zur Verfügung. Wir setzen also ans Ende dieses Programms – und aller unserer folgenden Programme – den Befehl *gtk.main()*

Einen Schönheitsfehler wollen wir noch beheben, bevor wir unser noch leeres Fenster mit Inhalt füllen: Dem Fenster fehlt noch ein Titel. Unter Windows würde beispielsweise in der Titelleiste unseres Fensters „python.exe" angezeigt (da es sich um ein Python-Programm handelt) oder unter Linux der Name unserer Programm-Datei. Geben wir unserem Fenster also noch einen eigenen Titel.

Wie wir bereits wissen, ist ein Fenster ein Objekt, und ein Objekt kann Funktionen haben. Von unserem Objekt *mein_Fenster* haben wir bereits die Funktion *show* benutzt, die das Fenster sichtbar macht. Eine weitere Funktion unseres Fensters ist *set_title* (englisch „Titel einstellen"). An dieser Funktion erkennen wir nun auch, warum wir Funktionen immer mit runden Klammern hinter dem Funktionsnamen aufrufen müssen: Ein Befehl wie „mein_Fenster.set_title()" würde so keinen Sinn machen – was für einen Titel soll das Fenster denn nun bekommen? Die Funktionsklammern erlauben uns, einer Funktion beim Aufruf weitere Informationen mitzugeben. Welche Informationen das genau sein können, bestimmt die jeweilige Funktion.

Abbildung 1: Unser erstes Fenster

Die Funktion *set_title* erwartet in den Klammern einen Text, der als Titel verwendet werden soll. Wenn wir unserem Fenster also beispielsweise den Titel „Ich bin ein Fenster!" geben möchten, nutzen wir den folgenden Befehl:
`mein_Fenster.set_title("Ich bin ein Fenster!")`

Beachte die Anführungszeichen um den Titel: Hiermit machen wir klar, dass es sich um einen einfachen Text handelt – in diesem Fall etwas, das auf dem Bildschirm ausgegeben werden soll. Die Anführungszeichen zeigen, dass dieser Text nicht etwa der Name eines Objektes oder einer Funktion oder Ähnliches sein soll. Schon in unserem „hallo, Welt"-Programm auf Seite 4 haben wir Anführungszeichen so verwendet.

Unser fertiges Fenster-Programm sieht also so aus:

Programm 2: Leeres Fenster mit Titel

```
1 import gtk
2
3 mein_Fenster = gtk.Window()
4 mein_Fenster.show()
5 mein_Fenster.set_title("Ich bin ein Fenster!")
6
7 gtk.main()
```

2.3 Objekte und Funktionen

Wir haben mit unserem Beispielprogramm 2 in Abschnitt 2 Objekte und Funktionen kennengelernt.

Ein **Objekt**, wie beispielsweise ein Fenster, ist ein konkretes Ding, das (meistens) etwas bestimmtes *hat* und etwas bestimmtes *kann*. Das Objekt mein_Fenster in unserem Beispiel *hat* den Titel „Ich bin ein Fenster!“. Es *kann* sich zum Beispiel anzeigen (mit der Funktion *show*) oder seinen Titel ändern (mit der Funktion *set_title*).

Eine **Funktion** ist also zum Beispiel etwas, das ein Objekt *kann* – oder andersherum: Alles, was ein Objekt kann, ist jeweils eine Funktion. Erinnern wir uns jedoch an unsere Definition eines Programms, können wir Funktionen genauer definieren. Ein Programm ist eine Folge von Befehlen. Eine Funktion ist ebenfalls eine Folge von Befehlen – jedoch werden diese Befehle nicht sofort beim Programmstart ausgeführt, sondern erst, wenn die entsprechende Funktion *aufgerufen* wird.

In unserem Beispiel haben wir aus unserem Programm heraus zum Beispiel die Funktion *show* unseres Objektes mein_Fenster aufgerufen. Daraufhin hat unser Programm eben diese Funktion *show* ausgeführt – und erst mit unserem nächsten Befehl weitergemacht, als die Funktion *show* fertig abgelaufen ist. Eine Funktion ist also eine Art Programm im Programm: ein *Unterprogramm*.

2.4 Reihenfolge von Befehlen

Die Reihenfolge von Befehlen in einem Programm ist oft von Bedeutung. In welcher Reihenfolge wir ein Objekt konstruieren und benutzen, ist logischerweise wichtig: Ein Objekt muss erst konstruiert werden, bevor es benutzt werden kann – denn ein Objekt, das es noch gar nicht gibt, kann auch nichts tun.

Würden wir zum Beispiel in unserem Programm 2 (Seite 7) die Zeilen 3 und 4 vertauschen – also erst die Funktion *mein_Fenster.show()* benutzen wollen, bevor wir das Objekt *mein_Fenster* überhaupt konstruieren – würde unser Programm mit einer Fehlermeldung abstürzen.

Genauso können wir die Zeilen 1 und 3 nicht vertauschen: Erst muss das gtk-System mit *import gtk* geladen werden, erst dann können wir ein Fenster mit Hilfe von gtk erstellen.

Die Zeilen 4 und 5 – also die Aufrufe von *show* und *set_title* – können wir dagegen vertauschen, ohne dass sich etwas merklich ändert.

Es kommt also auf den Inhalt unserer Befehle an – darauf, was wir damit tun. Obwohl die Reihenfolge bei einigen Befehlen egal ist, ist sie prinzipiell wichtig.

2.5 Inhalt für das Fenster

Unser Fenster soll nicht ewig leer bleiben. Deshalb lernen wir jetzt eine neue Klasse (d.h. einen neuen Typ) von Objekten kennen, die in Fenstern angezeigt werden können: Den Button (drückbarer Knopf). Dabei erweitern wir unser Programm 2 (Seite 7) – in das Fenster kommt eben ein Button rein.

Zunächst müssen wir ein neues Button-Objekt konstruieren (erstellen). Wie wir das tun, wissen wir eigentlich schon – schließlich haben wir schon ein Fenster-Objekt konstruiert. Wir erinnern uns: Das Objekt braucht einen eigenen Namen, den wir uns ausdenken können. Auch beim Button könnten wir wie beim Fenster darüber nachdenken, ihn „komisches_eckiges_Ding“ oder „Paula“ zu nennen. Wählen wir stattdessen trotzdem den Namen „mein_Button“.

Um ein Objekt zu konstruieren, schreiben wir erst seinen Namen, dann ein Gleichheitszeichen, und dann, was für ein Objekt es werden soll. In unserem Fall:

```
mein_Button = gtk.Button()
```

Genau wie unserem Fenster müssen wir auch unserem Button erst sagen, dass er sichtbar sein soll. Auch der Button hat eine Funktion *show*, die wir hierfür benutzen:

```
mein_Button.show()
```

Schließlich können wir, so ähnlich wie unserem Fenster, auch unserem Button noch eine Beschriftung geben. Bei Buttons heißt die nicht „title“ (englisch „Titel“) sondern „label“ (englisch „Aufkleber“, „Aufschrift“). Um etwas auf unseren Button draufzuschreiben, bietet er uns deshalb die Funktion *set_label* an. Die benutzen wir genauso, wie wir die Funktion *set_title* bei unserem Fenster benutzt haben.

```
mein_Button.set_label("Ich bin ein Button!")
```

Jetzt haben wir zwar ein Fenster-Objekt und ein Button-Objekt – trotzdem ist unser Fenster immer noch leer. Die beiden Objekte existieren zwar, kennen sich aber noch nicht. Wir müssen sie noch miteinander bekannt machen: Unser Button muss zu unserem Fenster hinzugefügt werden. Hierfür hat unser Fenster die Funktion *add* (englisch „hinzufügen“). In den Funktionsklammern geben wir der Funktion *add* die Information mit, welches Objekt zum Fenster hinzugefügt werden soll – nämlich *mein_Button*:

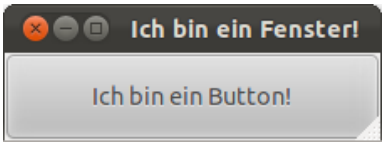

Abbildung 2: Ein Fenster mit Button

```
mein_Fenster.add(mein_Button)
```

Und wo in unserem Programm fügen wir diese drei Befehle jetzt ein?

Erinnern wir uns daran, was wir über die Reihenfolge von Befehlen (Seite 8) wissen:

1. Buttons erstellt das gtk-System für uns. Dafür muss gtk aber erst einmal geladen sein. Wir können mein_Button also erst konstruieren, wenn wir *import gtk* ausgeführt haben.

2. Unser Button soll unserem Fenster hinzugefügt werden – und dazu benutzen wir die Funktion *add* unseres Fensters. Das können wir erst, nachdem wir mein_Fenster konstruiert haben.

3. Mit dem Befehl *gtk.main()* (also dem Aufruf der Funktion *main* des gtk-Systems) geben wir die Kontrolle über unser Programm ab. Zu allen Befehlen, die danach kommen, kommt der Rechner nicht mehr. Unseren Button müssen wir also vor *gtk.main()* erstellen.

Eine sinnvolle Möglichkeit wäre also, die vier neuen Befehle direkt vor *gtk.main()* einzufügen. Unser Programm sieht dann so aus:

Programm 3: Ein Fenster mit Button

```
1 import gtk
2
3 mein_Fenster = gtk.Window()
4 mein_Fenster.show()
5 mein_Fenster.set_title("Ich bin ein Fenster!")
6
7 mein_Button = gtk.Button()
8 mein_Button.show()
9 mein_Button.set_label("Ich bin ein Button!")
10
11 mein_Fenster.add(mein_Button)
12
13 gtk.main()
```

Sicher fällt dir auf, dass (noch) überhaupt nichts passiert, wenn du das Programm startest und den Button anklickst. Was auch – wir haben keine Reaktion auf diesen Klick programmiert. Für den Moment beschäftigen wir uns noch etwas weiter damit, nur die Oberflächen von Programmen, also Fenster und ihren Inhalt, zu programmieren. In Kapitel 3 schreiben wir dann unser erstes Programm, bei dem ein Klick wirklich etwas auslöst.

2.6 Kommentare

Ein Tipp am Rande: Je länger und komplizierter ein Programm wird, desto schwieriger ist es, es später noch zu verstehen. Was man an dem einen Tag noch locker runterprogrammiert hat, kann eine Woche später schon zum großen Rätsel geworden sein. Deshalb gibt es in allen Programmiersprachen die Möglichkeit, Text einzufügen, der eigentlich gar nicht zum Programm gehört und vom Computer ignoriert wird – Text, der einfach nur dazu da ist, zu erklären, was das Programm an dieser Stelle tut.

Solche Texte nennt man Kommentare. In Python ist alles, was mit einer Raute (#) anfängt, ein Kommentar. Mit ein paar erklärenden Sätzen geschmückt, sieht unser Programm 3, das Fenster mit Button, jetzt so aus:

Programm 4: Ein Fenster mit Button, kommentiert

```
1 # Wir wollen das gtk-System benutzen:
2 import gtk
3
4 # Als erstes erstellen wir uns ein Fenster:
5 mein_Fenster = gtk.Window()
6 # Das Fenster soll sichtbar sein:
7 mein_Fenster.show()
8 # Und bekommt einen Titel:
9 mein_Fenster.set_title("Ich bin ein Fenster!")
10
11 # Jetzt erstellen wir uns einen Button:
```

```
12 mein_Button = gtk.Button()
13 # Auch der Button soll sichtbar sein:
14 mein_Button.show()
15 # Und bekommt eine Beschriftung:
16 mein_Button.set_label("Ich bin ein Button!")
17
18 # Schliesslich muss der Button noch in das Fenster rein:
19 mein_Fenster.add(mein_Button)
20 # Jetzt kennt das Fenster den Button
21 # und wir koennen ihn sehen!
22
23 # Am Ende geben wir die Kontrolle ueber unser Programm
24 # an das gtk-System ab, damit sich unser Programm
25 # nicht gleich wieder beendet.
26 gtk.main()
```

Wie viele solcher Kommentare du in dein Programm einfügt, ist dir überlassen – am Besten genau so viele, dass das Programm noch gut zu verstehen ist. Für unser Fenster mit Button würde man normalerweise sicher viel weniger Kommentare benutzen.

2.7 Objektdiagramme

In unserem Programm 4, dem Fenster mit Button, haben wir uns zwei Objekte konstruiert (mein_Fenster und mein_Button) und sie miteinander verbunden. Welche Objekte es in einem Programm gibt und wie deren Verbindungen untereinander sind, kann man in einem *Objektdiagramm* darstellen.

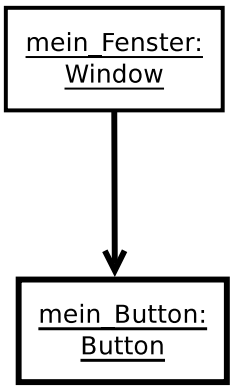

Abbildung 3: Ein Fenster mit Button, Objektdiagramm

In einem solchen Objektdiagramm werden die einzelnen Objekte, die in einem Programm vorkommen, als Kästen dargestellt. In jedem Kasten steht der Name des Objekts und dahinter, nach einem Doppelpunkt, von welcher Klasse das Objekt ist. Ein Pfeil zeigt eine Verbindung zwischen Objekten.

In unserem Beispiel kennt das Objekt mein_Fenster das Objekt mein_Button. Man kann auch sagen, das Objekt mein_Fenster *hat* das Objekt mein_Button. Wir erinnern uns an unsere Definition von *Objekt*: Ein Objekt ist ein konkretes Ding, das (meistens) etwas *kann* und etwas *hat*. Dieses Fenster hat einen Button. Abb. 3 zeigt das Objektdiagramm zum Programm „Fenster mit Button“.

Ist das Objektdiagramm eindeutig?

Zu einem gegebenen Programm, zum Beispiel unserem Programm 4, gibt es immer nur genau ein richtiges Objektdiagramm – nämlich das in Abb. 3. Schließlich stellt das Objektdiagramm

genau dar, welche Objekte es in diesem Programm gibt und wie sie miteinander verbunden sind.[3]

Andersherum gibt es diese Eindeutigkeit jedoch nicht: Zu einem Objektdiagramm können viele verschiedene Programme passen. Ein Objektdiagramm beschreibt nur die in einem Programm verwendeten Objekte – und nicht, was das Programm tut. Zum Beispiel könnten wir in Programm 4 den Titel des Fensters noch einmal ändern oder ein *print "hello, world"* einfügen. Damit haben wir das Programm verändert, aber das Objektdiagram aus Abb. 3 passt immer noch.

2.8 Gleich zwei Fenster auf einmal o.Ô

Natürlich können Programme auch mehr als nur ein Fenster enthalten. Zur Übung schreiben wir jetzt ein Programm, das zwei Fenster anzeigt – beide Fenster sollen jeweils einen Button enthalten. Wie machen wir das?

Im Prinzip müssen wir nur den Teil unserer Programms, der ein Fenster und einen Button erzeugt, kopieren. Dabei müssen wir nur beachten, dass jedes Objekt einen eigenen Namen braucht. Für unser Beispiel wählen wir einfach *zweites_Fenster* und *zweiter_Button* als Namen für die neuen Objekte. Das Ergebnis findest du unter Programm 5.

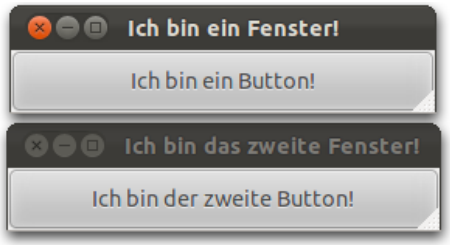

Abbildung 4: Zwei Fenster mit je einem Button

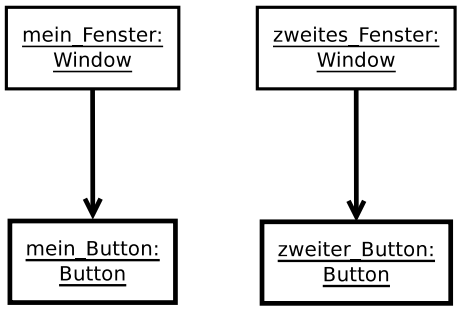

Abbildung 5: Zwei Fenster mit je einem Button, Objektdiagramm

Auch zu diesem Programm erstellen wir wieder ein Objektdiagramm (Abb. 5). Wie wir darin sehen, ist je ein Fenster-Objekt zu seinem Button-Objekt verbunden. Die beiden Fenster oder die beiden Buttons untereinander kennen sich jedoch nicht.

3 Definition eigener Funktionen

In diesem Kapitel werden wir endlich ein Programm schreiben, das wirklich etwas tut! Wir werden eigene Funktionen schreiben und diese mit unseren Buttons verbinden – so dass auch wirklich etwas passiert, wenn wir einen Button drücken.

[3]Natürlich kann man ein Objektdiagramm verschieden zeichnen – zum Beispiel könnte man in Abb. 3 auch den Button oben und das Fenster unten zeichnen. Trotzdem bezeichnet man das dann immer noch als *das gleiche* Objektdiagramm.

Programm 5: Zwei Fenster mit je einem Button

```
1  import gtk # diese Zeile brauchen wir nur einmal!
2
3  # Zuerst erstellen wir uns wie gehabt je ein Fenster
4  # und einen Button und verbinden sie miteinander:
5
6  mein_Fenster = gtk.Window()
7  mein_Fenster.show()
8  mein_Fenster.set_title("Ich bin ein Fenster!")
9
10 mein_Button = gtk.Button()
11 mein_Button.show()
12 mein_Button.set_label("Ich bin ein Button!")
13
14 mein_Fenster.add(mein_Button)
15
16 # Danach erstellen wir ein zweites Fenster und
17 # einen zweiten Button und verbinden diese beiden:
18
19 zweites_Fenster = gtk.Window()
20 zweites_Fenster.show()
21 zweites_Fenster.set_title("Ich bin das zweite Fenster!")
22
23 zweiter_Button = gtk.Button()
24 zweiter_Button.show()
25 zweiter_Button.set_label("Ich bin der zweite Button!")
26
27 zweites_Fenster.add(zweiter_Button)
28
29 gtk.main() # diese Zeile brauchen wir nur einmal!
```

Bisher haben wir immer nur vorhandene Funktionen verwendet – wie zum Beispiel die Funktion *show* von Button-Objekten oder die Funktion *set_title* von Window-Objekten. Jetzt gehen wir auf die andere Seite und programmieren selbst Funktionen. Dazu erinnern wir uns zunächst an unsere grundsätzliche Definition einer Funktion:

Eine Funktion ist, ähnlich wie ein Programm, eine Folge von Befehlen. Diese Befehle werden aber nicht automatisch ausgeführt, sondern erst, wenn wir diese Funktion auch aufrufen. Eine Funktion ist also ein *Unterprogramm*.

Was kann eine Funktion also tun, was kann sie für Befehle enthalten? Prinzipiell alles, was auch unser (Haupt-)Programm auch kann. Unser erstes grafisches Programm in Abschnitt 2 hat einfach nur ein leeres Fenster angezeigt. Also schreiben wir jetzt eine Funktion, die uns immer, wenn wir sie aufrufen, ein neues, leeres Fenster anzeigt. Diese Funktion wollen wir *zeig_mir_ein_Fenster* nennen.

Eine Funktion definieren wir, indem wir das Schlüsselwort **def** (für definieren) aufschreiben, dahinter den Namen der neuen Funktion mit den bekannten Funktionklammern, und zum Schluss einen Doppelpunkt:

```
def zeig_mir_ein_Fenster():
```

Alles, was wir jetzt *eingerückt* (also z.B. mit zwei Leerzeichen am Anfang) in die folgenden Zeilen schreiben, gehört zur Funktion – diese Befehle sind dann Teil der Funktion und nicht des Hauptprogramms. Unsere Funktion *zeig_mir_ein_Fenster* könnte also so aussehen:

```
def zeig_mir_ein_Fenster():
  mein_Fenster = gtk.Window()
  mein_Fenster.show()
  mein_Fenster.set_title("Ich bin ein Fenster!")
```

Programm 6 enthält diese Funktionsdefinition und ruft die Funktion *zeig_mir_ein_Fenster* danach dreimal auf. Beachte, dass die Befehle, die die Funktion aufrufen, nicht mehr eingerückt sind – sie gehören also nicht mehr zur Funktion sondern wieder zum Hauptprogramm. Im Ergebnis zeigt dieses Programm also sofort nach dem Start drei leere Fenster an.

3.1 Neue Fenster auf Knopfdruck

Die Funktion *zeig_mir_ein_Fenster*, die wir im letzten Abschnitt geschrieben haben, wollen wir jetzt auch benutzen! Dafür schreiben wir ein Hauptprogramm, das ein Fenster mit Button anzeigt (das haben wir ja schon einmal gemacht – siehe Programm 3). Über das Hauptprogramm setzen wir die Definition der Funktion *zeig_mir_ein_Fenster*. Fehlt nur noch die Verbindung zwischen dieser Funktion und unserem Button!

Hierfür stellt uns jedes *Button*-Objekt die Funktion *connect* (englisch „verbinden“) zur Verfügung[4]. Die Funktion *connect* erwartet zwei Parameter: Zunächst in Anführungszeichen den Namen eines Ereignisses, mit dem eine Funktion verbunden werden soll. Wir möchten, dass

[4]Genauer: *connect* ist eine Funktion der Klasse *GObject*, von der die Klasse *Button* indirekt, über vier Zwischenklassen, abgeleitet ist. Für uns reicht es aber zu sagen, *connect* ist eine Funktion der Klasse *Button*. Mehr über Klassen erfährst du im nächsten Kapitel.

Programm 6: Benutzt unsere erste eigene Funktion

```
1 import gtk
2
3 def zeig_mir_ein_Fenster():
4   mein_Fenster = gtk.Window()
5   mein_Fenster.show()
6   mein_Fenster.set_title("Ich bin ein Fenster!")
7 # Die drei Befehle oben gehoeren zur Funktion:
8 # sie sind eingerueckt.
9
10 # Die folgenden Befehle gehoeren nicht mehr zur Funktion,
11 # denn sie sind nicht eingerueckt!
12 zeig_mir_ein_Fenster()
13 zeig_mir_ein_Fenster()
14 zeig_mir_ein_Fenster()
15
16 gtk.main()
```

etwas passiert, wenn der Button geklickt wird – also nehmen wir das Ereignis *"clicked"*. Dieses Ereignis wird uns von gtk zur Verfügung gestellt – gtk erkennt immer, wenn auf unserem Button ein Klick stattfindet und prüft, ob etwas mit diesem Ereignis verbunden wurde.[5]

Als zweiten Parameter erwartet die Funktion *connect* den Namen einer Funktion, die aufgerufen werden soll, wenn das Ereignis eintritt – also *zeig_mir_ein_Fenster*. Beachte, dass wir hinter *zeig_mir_ein_Fenster* hier keine Funktionsklammern brauchen: Diese brauchen wir nur, wenn wir die Funktion definieren oder wenn wir sie aufrufen. Wir rufen die Funktion *berechnen* an dieser Stelle nicht auf – das tut später das gtk-System für uns, wenn der Button geklickt wurde.

Wenn du deinen Button wie in Programm 3 *mein_Button* genannt hast, lautet der Befehl zum Verbinden des Buttons mit unserer Funktion berechnen also so:
`mein_Button.connect("clicked", zeig_mir_ein_Fenster)`

Diesen Befehl fügen wir irgendwo hinter der Definition des Buttons ein. Das wars fast! Nur noch eine Kleinigkeit müssen wir anpassen, damit wir auf Knopfdruck neue Fenster kriegen...

3.2 Funktionsparameter

Mehrfach haben wir uns bisher mit Funktionsklammern beschäftigt und damit, dass man manchen Funktionen zusätzliche Informationen in diesen Klammern mitgeben kann. Unser erstes Beispiel war das Setzen des Titels eines Fensters:
mein_Fenster.set_title(**"Ich bin ein Fenster!"**)

Diese Informationen, die man Funktionen in Funktionsklammern mitgibt, nennt man *Parameter*. Innerhalb der Funktion – also aus Sicht desjenigen, der die Funktion programmiert –

[5] Genau dieses Prüfen auf eintretende Ereignisse ist die Hauptaufgabe der Funktion *gtk.main*, die man auch als *Event Loop* (englisch: „Ereignisschleife“) bezeichnet.

stehen diese Parameter dann zur Verfügung und die Funktion kann sie benutzen. Im Beispiel der Funktion *set_title* kann sie also den gewünschten Text als Parameter lesen und dem Fenster als Titel geben. Andere Funktionen könnten zum Beispiel Zahlen als Parameter erwarten und dann mit ihnen rechnen.[6]

Wenn wir eine Funktion programmieren und diese soll einen Parameter akzeptieren, müssen wir einfach in den Funktionsklammern einen Namen für diesen Parameter angeben – genau so, wie wir auch Objekten in unserem Programm immer einen Namen geben. Nehmen wir also an, unsere Funktion *zeig_mir_ein_Fenster* würde noch einen Parameter akzeptieren. Dann sähe ihre erste Zeile so aus:

```
def zeig_mir_ein_Fenster(mein_Parameter):
```

Unter dem Namen *mein_Parameter* könnten wir dann innerhalb der Funktion auf die mitgelieferten Daten zugreifen.

Unsere Funktion *zeig_mir_ein_Fenster* braucht tatsächlich einen Parameter! Und zwar sobald wir möchten, dass unsere Funktion automatisch bei Klick auf einen bestimmten Button ausgeführt wird. Wir selbst werden den Parameter überhaupt nicht verwenden – aus unserer Sicht brauchen wir ihn also nicht. Wenn aber die Funktion aber bei Klick auf einen Button laufen soll, dann ist das gtk-System dafür zuständig, sie aufzurufen (wir bringen ihm gleich bei, das zu tun). gtk gibt so aufgerufen Funktionen immer die Information mit, wer ihren Aufruf ausgelöst hat – also in diesem Fall unser Button *mein_Button*. gtk ruft unsere Funktion also immer mit einem *Parameter* auf, und zwar unserem Button-Objekt *mein_Button*.

Die einzige Änderung, die noch fehlt, ist also, in der Definition unserer Funktion *zeig_mir_ein_Fenster* einen Parameter zu akzeptieren. Und fertig – unser Programm produziert jetzt auf Knopfdruck leere Fenster. Großartig, oder?! Meine Version findest du unter Programm 7.

3.3 Befehlsreihenfolge und Funktionen

Eine Anmerkung noch zur Reihenfolge der Befehle in unseren Programmen, über die wir uns schon im Abschnitt 2.4 unterhalten haben: In unserem Programm muss die Funktion *zeig_mir_ein_Fenster* über dem Hauptprogramm stehen – andersherum funktioniert es nicht. Das liegt daran, dass wir im Hauptprogramm die Funktion *zeig_mir_ein_Fenster* verwenden – indem wir sie als Parameter in die Funktion *connect* unseres Buttons geben. An dieser Stelle im Hauptprogramm, wo wir die Funktion *connect* benutzten, muss unsere Funktion *zeig_mir_ein_Fenster* also schon definiert sein – denn etwas, was es noch nicht gibt, können wir nicht benutzen (und also auch nicht als Parameter verwenden).

3.4 Aufgabe: „Fenster-Spam“

Ergänze das Programm aus dem letzten Abschnitt: Auch in den sich auf Knopfdruck öffnenden Fenstern soll sich wieder ein Button befinden. Klickt man diesen, soll sich ebenfalls ein neues Fenster öffnen, in dem sich wieder ein Button mit dieser Funktion befindet. Mit jedem Klick auf irgendeinen Button soll also wieder ein neues Fenster mit Button entstehen.

[6] Innerhalb der Funktion steht ein Parameter wie eine Variable zur Verfügung. Was Variablen sind, besprechen wir später in Abschnitt 6.2.

Programm 7: Neue Fenster auf Knopfdruck

```
1 def zeig_mir_ein_Fenster(mein_Parameter):
2   mein_Fenster = gtk.Window()
3   mein_Fenster.show()
4   mein_Fenster.set_title("Ich bin ein Fenster!")
5
6 import gtk
7
8 mein_Fenster = gtk.Window()
9 mein_Fenster.show()
10 mein_Fenster.set_title("Ich bin ein Fenster!")
11
12 mein_Button = gtk.Button()
13 mein_Button.show()
14 mein_Button.set_label("Ich bin ein Button!")
15 mein_Button.connect("clicked", zeig_mir_ein_Fenster)
16
17 mein_Fenster.add(mein_Button)
18
19 gtk.main()
```

3.5 Einschub: Ereignisorientierung

Diese Art von Ablaufsteuerung eines Programmes, die wir gerade programmiert haben, nennt man *ereignisorientierte Programmierung* oder mit englischem Fachbegriff *event driven programming*: Ein Programm reagiert auf bestimmte Ereignisse, die oft von außen kommen (wie in unserem Beispiel der Klick auf einen Button), aber auch vom Programm selbst ausgelöst werden können.

Ermöglicht wird das von unserer alten Bekannten, der Funktion *gtk.main*. Bisher sagten wir nur, wir rufen diese Funktion immer am Ende unseres Programmes auf, damit es sich nicht beendet und die von uns programmierten grafischen Benutzeroberflächen (Fenster etc.) sichtbar bleiben. Nun haben wir noch eine weitere Aufgabe dieser Funktion kennen gelernt: Sie prüft ständig, ob Ereignisse in unserem Programm eintreten, für die wir eine bestimmte Reaktion programmiert haben – wie in unserem Beispiel ein Klick auf den „=“-Button.

Die Ereignisorientierung ist ein bestimmtes Prinzip oder ein bestimmter Stil, wie man Programme aufbauen kann. Der Informatiker sagt, Ereignisorientierung ist ein bestimmtes *Programmier-Paradigma*.

Seit dem Anfang unseres Kurses haben wir bereits ein anderes Paradigma kennen gelernt: Die *Objektorientierung*, bei der ein Programm in viele kleine Objekte aufgeteilt wird, die alle ihre eigene kleine Aufgabe haben – so dass sich aus dem Zusammenspiel der einzelnen Objekte das Gesamtprogramm ergibt.

Wie wir jetzt sehen, können in einem Programm auch mehrere Paradigmen gleichzeitig zum Einsatz kommen. Objektorientierung und Ereignisorientierung gemeinsam sind typisch für die

Programmierung mit *gtk*.

4 Klassen

Den Begriff der Klasse haben wir bisher immer schon nebenbei verwendet. Wir haben erwähnt, dass eine Klasse der Typ oder die Art eines Objektes ist. Darauf gehen wir in diesem Abschnitt genauer ein.

Erinnern wir uns an unsere Definition eines Objekts: Ein Objekt ist ein konkretes Ding – wie ein bestimter Bildschirm in der realen Welt. Zwei Bildschirme sind zwei verschiedene Objekte. Ich kann entweder den einen Bildschirm anfassen, oder den anderen, oder beide gleichzeitig. Ein grauer Bildschirm ist kein blauer Bildschirm. Dein Bildschirm ist nicht mein Bildschirm. Die beiden Bildschirm-Objekte sind verschieden. Trotzdem gehören beide zur Klasse Bildschirm.

Eine Klasse ist eine abtrakte Idee und kann sozusagen der Bauplan für eine Art von Objekten sein[7]. Die Klasse „Bildschirm“ in der realen Welt können wir nicht anfassen. Trotzdem wissen wir, dass jeder Bildschirm grundsätzlich *das gleiche hat* und *das gleiche kann*. Jeder Bildschirm hat einen Anschluss, um ihn mit dem Computer zu verbinden und jeder Bildschirm hat einen Stromanschluss. Jeder Bildschirm kann ein Bild anzeigen. Jedes Objekt der Klasse Bildschirm hat diese Eigenschaften.

Ähnlich ist es mit Objekten und Klassen im Computer. Ein bestimmtes Fenster und ein bestimmter Button sind konkrete Objekte. Im Computer sind zwar, wie wir gelernt haben, nicht alle Objekte sichtbar – ein Fenster (also ein Objekt der gtk-Klasse *Window*), auf dem wir noch nicht die Funktion *show* aufgerufen haben, können wir (noch) nicht sehen. Ob wir ein nicht-sichtbares Fenster wirklich „anfassen“ können, ist eine Frage der persönlichen Vorstellungskraft – trotzdem ist auch ein (noch) nicht-sichtbares Fenster ein ganz bestimmtes, eigenes Objekt.

In unserem Programm 5, zwei Fenster mit je einem Button, haben wir z.B. zwei verschiedene Objekte der Klasse *Window*. Das sind zwei völlig unabhängige Objekte – sie kennen sich nicht einmal. Trotzdem gehören sie beide zur selben Klasse. Eine Klasse bestimmt die Art des Objekts. Das bedeutet zum Beispiel, dass beide *Window*-Objekte die gleichen Dinge *können*, also gleiche Funktionen haben, wie *show* und *set_title*. Die beiden *Window*-Objekte können auch beide gleichartige Dinge *haben* – zum Beispiel beide einen Titel und beide einen Button. Trotzdem haben die beiden konkreten *Window*-Objekte unterschiedliche Titel und unterschiedliche Buttons. Ganz am Anfang in Abschnitt 2 haben wir auch ein *Window*-Objekt kennen gelernt, das weder Titel noch Button hat.

4.1 Welche Klassen kennen wir bisher?

Sicher erkennst du sehr schnell, dass wir bisher die Klassen *Window* und *Button* verwendet haben.

[7] Erzeugt man ein Objekt von einer Klasse, benutzt man die Klasse also als Bauplan für ein neues Objekt, sagt man, man *instantiiert* diese Klasse. Es gibt aber auch sogenannte *abstrakte* Klassen, die nicht instantiiert werden können. Für diesen Kurs gehen solche Details zu weit.

Nicht ganz so offensichtlich ist, dass auch *gtk* ein Klasse ist. gtk ist eine Klasse, von der wir nie ein Objekt erzeugt habe. Diese Klasse ist also sozusagen nicht Bauplan, sondern nur eine abstrakte Idee. Trotzdem haben wir sie verwendet.

Auch eine Klasse *hat* und *kann* (manchmal) etwas – so ähnlich wie ein Objekt. Eine Klasse, die etwas kann, hat also *Funktionen*. So haben wir zum Beispiel die Funktion *main* der Klasse *gtk* kennengelernt – sie rufen wir immer am Ende unseres Programms auf mit:
`gtk.main()`

Gehört eine Funktion zu einer Klasse und nicht zu einem konkreten Objekt, bedeutet das auch, dass diese Funktion kein konkretes Objekt verändert. Beispiel: Die Funktion *set_title*, die zu einem Window-*Objekt* gehört, verändert dieses Objekt – sie verändert den Titel, den dieses Objekt *hat*. Wenn wir die Funktion *main* der Klasse *gtk* aufrufen, ist dagegen garantiert, dass kein konkretes Objekt verändert wird.[8]

4.2 Klassendiagramme

Klassen können andere Dinge *haben* – zum Beispiel andere Klassen! Die Klasse gtk **hat** die Klassen *Window* und *Button* – oder andersherum: Die Klassen *Window* und *Button* gehören zur Klasse *gtk*. Jetzt wird auch klarer, warum wir ein Window-Objekt mit dem Befehl
`mein_Fenster = gtk.Window()`
erstellen müssen: Wir wählen erst die Klasse *gtk*, und dann die in ihr enthaltene Klasse *Window*.

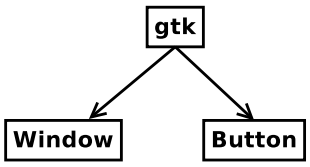

Abbildung 6: Klassendiagramm unserer bisher bekannten Klassen

Diese Beziehung zwischen verschiedenen Klasse kann man in einem *Klassendiagramm* darstellen. Es zeigt die betrachteten Klassen und ihre Verbindungen untereinander.[9] Das Klassendiagramm der drei Klassen, die wir bisher kennen (*gtk*, *Window* und *Button*), zeigt Abb. 6.

Anmerkung: Klassen können zwar andere Klassen besitzen, aber keine Objekte. Etwas so abstraktes wie eine Klasse kann nichts so konkretes wie ein Objekt besitzen. Umgekehrt kann ein Objekt auch keine Klasse besitzen. „Eine Klasse hat Objekte“ sagt man dagegen, wenn es Objekte gibt, die zu dieser Klasse gehören – also Objekte, für die diese Klasse der Bauplan war.

4.3 Konstruktoren

Schauen wir uns nochmal den Befehl, mit dem wir ein neues *Window*-Objekt konstruieren (erzeugen), genauer an:
`mein_Fenster = gtk.Window()`

[8] Zumindest so lange nicht, wie wir nicht z.B. durch den Klick auf einen Button etwas bestimmtes auslösen. Wie das geht, lernen wir in Kapitel 6. In diesem Fall funktioniert gtk.main() als Vermittler zwischen dem Klick auf ein Buton-*Objekt* und der darauf programmierten Reaktion – und in dieser Reaktion, die ja zu einem Button-*Objekt* gehört, können dann durchaus Objekte verändert werden.

[9] Außerdem kann man in einem Klassendiagramm noch Funktionen und Attribute angeben – also das, was Objekte der jeweiligen Klassen *können* und *haben*. Wir beschränken uns der Einfachkeit halber nur auf die Klassen selbst und ihre Beziehungen.

Wir haben gesagt, *Window* ist eine Klasse, die der Klasse *gtk* gehört: deshalb „gtk.Window“. Die runden Klammern am Ende zeigen uns jedoch, dass *Window()* auch eine Funktion ist. *Window()* ist die Funktion, die uns ein Objekt der Klasse *Window* konstruiert. Wir nennen *Window()* deshalb einen *Konstruktor.*

Mit der Klasse *Button* verhält es sich genauso.

In Python sind Konstruktoren immer Funktionen, die genauso heißen, wie die Klasse, von der sie ein Objekt konstruieren.

Übrigens: Auch vielen Konstruktoren kann man in den Funktionsklammern Parameter mitgeben. Der Konstruktor von *Button* erlaubt es beispielsweise, direkt den Text mit anzugeben, der auf dem Button stehen soll. Statt diesen umständlichen zwei Befehlen:

```
mein_Button = gtk.Button()
mein_Button.set_label("Ich bin ein Button!")
```

Können wir also auch einfach schreiben:

```
mein_Button = gtk.Button("Ich bin ein Button!")
```

Dem Konstruktor von *Window* können wir dagegen leider nicht direkt den Titel des Fensters mitgeben.

5 Mehr zu grafischen Benutzeroberflächen

Bisher haben unsere Fenster immer nur ein einziges Objekt enthalten. Zum Beispiel haben wir in Programm 3 mit dieser Anweisung einen Button zu einem Fenster hinzugefügt:

```
mein_Fenster.add(mein_Button)
```

Vielleicht hast du schon ausprobiert, einfach einen weiteren Button zu erzeugen und diesen auch mit der Funktion *add* eures Fensters hinzuzufügen. Das wäre logisch, funktioniert mit gtk aber leider nicht und sorgt für eine Fehlermeldung.

Die Logik dahinter ist, dass nicht eindeutig bestimmt ist, wo ein zweiter Button in einem Fenster plaziert werden soll. Enthält ein Fenster nur einen Button, ist die Sache eindeutig: Der Button ist einfach so groß wie das Fenster. Fügen wir einen zweiten Button hinzu, gibt es verschiedene Möglichkeiten: Wo soll den neue Button hin? Neben den ersten oder darunter?

5.1 Mehrere Objekte in einem Fenster

gtk stellt uns zwei Klassen zur Verfügung, mit denen wir genau diese Frage beantworten können: *HBox* („horizontale Box“) und *VBox* („vertikale Box“). In *HBox*- und *VBox*-Objekte können wir andere Objekte (z.B. Buttons) hineinstecken – so, wie wir es von einer Box erwarten. Dazu stellen beide, genau wie das Fenster, eine Funktion *add* zur Verfügung. Im Gegensatz zum Fenster, das immer nur ein anderes Objekt aufnehmen kann, können wir beliebig viele andere Objekte in eine *HBox* oder *VBox* stecken.

Ein *HBox*-Objekt sorgt dabei dafür, dass die enthaltenen Objekte *horizontal*, also nebeneinander platziert werden. Es ist also eine Art Zeile. Ein *VBox*-Objekt ordnet seine enthaltenen Objekte dagegen *vertikal*, also untereinander an. Es ist eine Art Spalte.

Programm 8: Zwei Buttons nebeneinander

```
1  import gtk
2
3  # Zuerst brauchen wir ein Fenster:
4  mein_Fenster = gtk.Window()
5  mein_Fenster.show()
6  mein_Fenster.set_title("Ich bin ein Fenster!")
7
8  # Dann brauchen wir ein HBox-Objekt:
9  zeile = gtk.HBox()
10 zeile.show()
11
12 # Die Zeile muss ins Fenster:
13 mein_Fenster.add(zeile)
14
15 # Jetzt brauchen wir zwei Buttons:
16 erster_Button = gtk.Button("Ich bin ein Button!")
17 erster_Button.show()
18
19 zweiter_Button = gtk.Button("Ich bin der zweite Button!")
20 zweiter_Button.show()
21
22 # Die Buttons kommen in die Zeile,
23 # dann sind sie nebeneinander:
24 zeile.add(erster_Button)
25 zeile.add(zweiter_Button)
26
27 gtk.main()
```

Da wir nun zwei neue Klassen von gtk kennen gelernt haben, können wir unser Klassendiagramm erweitern (Abb. 7).

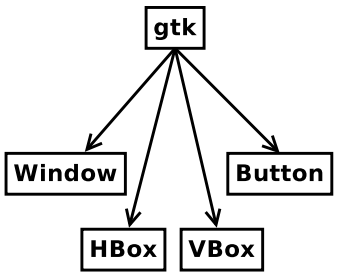

Abbildung 7: Klassendiagramm unserer bisher bekannten Klassen

Zwei Buttons nebeneinander

Unser nächstes Beispielprogramm soll zwei Buttons in einer Zeile enthalten. Wie gehen wir also vor?

1. Zunächst konstruieren wir ein Fenster und machen es sichtbar (Funktion *show*).
2. Dann konstruieren wir ein HBox-Objekt, das wir ebenfalls sichtbar machen.
3. Das HBox-Objekt fügen wir ins Fenster ein (Funktion *add* des Fensters).

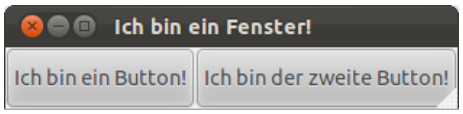

Abbildung 8: Zwei Buttons nebeneinander

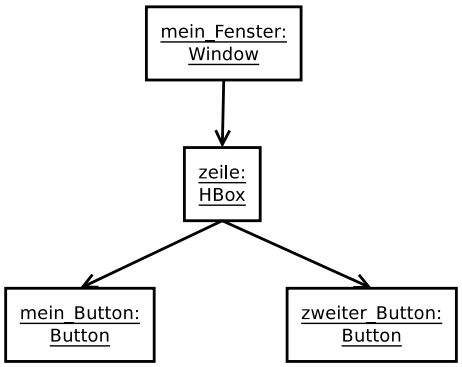

Abbildung 9: Zwei Buttons nebeneinander – das Objektdiagramm

4. Nun konstruieren wir noch beide Buttons und machen sie sichtbar.

5. Die beiden Buttons fügen wir ins HBox-Objekt ein (zweimal Funktion *add* des HBox-Objekts)

Fertig! Die einzelnen Anweisungen, um das zu programmieren, findest du unter *Programm 8*.

Auch zu diesem Programm zeichnen wir wieder ein Objektdiagramm (Abb. 9). Wir sehen, dass die Buttons nicht direkt mit dem Fenster verbunden sind. Das Fenster enthält das HBox-Objekt, und erst das HBox-Objekt enthält die Buttons.

Zum Schluss dieses Abschnitts eine kurze Fingerübung: Wie musst du das Programm verändern, so dass die beiden Buttons nicht mehr nebeneinander sondern untereinander sind?

5.2 Layouts

Der Begriff *Layout* (englisch „Aufbau“, „Aufmachung“) kommt aus dem Buch- und Zeitungsdruck und beschreibt, wie verschiedene Objekte angeordnet sind: In der Zeitung zum Beispiel also die Anordnung verschiedener Textspalten und Bilder auf einer Seite – in unserem Programmen die Anordnung von Buttons[10] in einem Fenster.

Bis jetzt haben wir nur zwei sehr einfache Layouts kennengerlernt: Die Zeile (mehrere Objekte in einem *HBox*-Objekt) und die Spalte (mehrere Objekte in einem *VBox*-Objekt). Aus diesen einfachen Bausteinen können wir fast alle denkbaren Layouts zusammenbauen, indem wir Boxen ineinanderschachteln. Wir können also z.B. in ein *VBox*-Objekt ein *HBox*-Objekt *add*-en oder umgekehrt.

Nehmen wir an, wir wollen ein einfaches Layout basteln, in dem vier Buttons im Quadrat angeordnet sind. Überlegen wir zuerst, wie wir dieses Layout auf „Spalten“ und „Zeilen“ herunterbrechen können. Es gibt zwei Möglichkeiten, als was wir diese vier Buttons im Quadrat sehen können: Entweder, das sind zwei Spalten à zwei Buttons, die sich nebeneinander befinden – oder aber zwei Zeilen à zwei Buttons, die sich untereinander befinden.

[10] Und, wie wir im nächsten Abschnitt sehen werden, natürlich auch anderen sichtbaren Objekten.

Wir nehmen uns für dieses Beispiel mal die zweite Sichtweise vor: Vier Buttons im Quadrat sind zwei Zeilen à zwei Buttons. In Abbildung 10 siehst du das noch einmal im Bild.

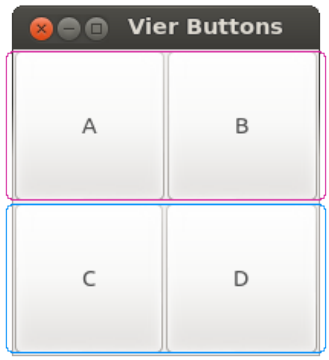

Abbildung 10: Vier Buttons in einem Fenster

Wir brauchen im Fenster also als erstes ein *VBox*-Objekt, um die beiden Zeilen untereinander anzuordnen. In dieses *VBox*-Objekt kommen dann zwei *HBox*-Objekte, die für die beiden Zeilen stehen. In jedes *HBox*-Objekt kommen zum Schluss zwei Buttons. Eine Beispiel-Lösung findet du in Programm 9.

5.3 Text im Fenster

Ein Fenster braucht natürlich noch andere Inhalte als immer nur Buttons. Um einfachen Text in ein Fenster zu schreiben, brauchen wir bei gtk ein Objekt der Klasse *Label* (englisch „Aufkleber“, „Aufschrift“). Ein solches *Label*-Objekt konstruieren wir nach genau dem gleichen Muster, wie wir das auch von Fenster und Buttons kennen. Wie bei Buttons können wir den gewünschten Text direkt dem Konstruktor als Parameter mitgeben.

Ein *Label*-Objekt können wir also zum Beispiel mit diesem Befehl konstruieren:

```
erklaerung = gtk.Label("Zum Starten den Button anklicken")
```

Natürlich muss auch ein *Label*-Objekt danach noch mit der Funktion *show* sichtbar gemacht und entweder direkt einem Fenster oder einer HBox oder VBox hinzugefügt werden.

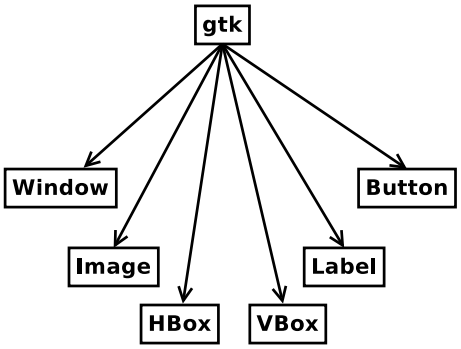

Abbildung 11: Klassendiagramm

5.4 Bilder

Um Bilder in unseren Programmen zu verwenden, brauchen wir die Klasse *Image* (englisch „Bild“). Auch *Image-Objekte* konstruieren wir genau so, wie wir das gewohnt sind, zum Beispiel so:

```
mein_Bild = gtk.Image()
```

Anschließend müssen wir dem Objekt noch mitteilen, welches Bild es anzeigen soll. Das machen wir mit der Funktion *set_from_file*. Diese verlangt als Parameter nach dem Dateinamen des Bildes:

```
mein_Bild.set_from_file("meinbild.jpg")
```

Das Bild muss dafür im gleichen Ordner wie dein Python-Programm liegen[11]. Wichtig ist auch, dass du wirklich den genauen Dateinamen einschließlich Endung (z.B. „.jpg“) angibst. Leider blendet Windows diese Endungen standardmäßig aus. Wenn du in einem Ordnerfenster im Menü *Extras* auf *Ordneroptionen* klickst und im sich dann öffnenden Fenster im Tab *Ansicht* das Häkchen *Erweiterungen bei bekannten Dateitypen ausblenden* entfernst, siehst du die Endungen wieder.

[11] Relative (oder – gar nicht empfehlenswert – auch absolute) Pfadangaben sind auch möglich.

Programm 9: Vier Buttons in einem Fenster

```
1  import gtk
2
3  fenster = gtk.Window()
4  fenster.show()
5  fenster.set_title("Vier Buttons")
6
7  # Unser Layout besteht aus zwei Zeilen untereinander!
8
9  untereinander = gtk.VBox()
10 untereinander.show()
11 fenster.add(untereinander)
12
13 # Hier kommt die obere Zeile mit den Buttons A und B
14
15 zeile1 = gtk.HBox()
16 zeile1.show()
17 untereinander.add(zeile1)
18
19 linksoben = gtk.Button("A")
20 linksoben.show()
21 zeile1.add(linksoben)
22
23 rechtsoben = gtk.Button("B")
24 rechtsoben.show()
25 zeile1.add(rechtsoben)
26
27 # Hier kommt die untere Zeile mit den Buttons C und D
28
29 zeile2 = gtk.HBox()
30 zeile2.show()
31 untereinander.add(zeile2)
32
33 linksunten = gtk.Button("C")
34 linksunten.show()
35 zeile2.add(linksunten)
36
37 rechtsunten = gtk.Button("D")
38 rechtsunten.show()
39 zeile2.add(rechtsunten)
40
41 gtk.main()
```

Programm 10: Programm mit Katze! Beispiel für *Label*- und *Image*-Objekte

```
1  import gtk
2
3  mein_Fenster = gtk.Window()
4  mein_Fenster.show()
5  mein_Fenster.set_title("Katzen sind toll")
6
7  untereinander = gtk.VBox()
8  untereinander.show()
9  mein_Fenster.add(untereinander)
10
11 mein_Bild = gtk.Image()
12 mein_Bild.show()
13 mein_Bild.set_from_file("katze.jpg")
14 untereinander.add(mein_Bild)
15
16 text = gtk.Label("Ist sie nicht suess?")
17 text.show()
18 untereinander.add(text)
19
20 gtk.main()
```

Denk daran, auch ein *Image*-Objekt mit der Funktion *show* sichtbar zu machen und einer HBox, einer VBox oder direkt einem Fenster hinzuzufügen.

In Programm 10 findest du ein Beispiel, das sowohl ein *Image*- als auch ein *Label*-Objekt benutzt. Wie dieses kleine Programm aussieht, siehst du in Abb. 12.

Da wir nun wieder zwei neue Unterklassen von *gtk* kennengelernt haben – *Label* und *Window* – wird es mal wieder Zeit, unser Klassendiagramm zu erweitern! Du findest es in Abb. 11.

Abbildung 12: Das Katzenprogramm
Foto: Matthew Maguire, lizensiert CC-BY-NC

5.5 Aufgabe: Eine interaktive Geschichte

Schreibe in Python mit gtk ein kleines Spiel: Mit Text und Bilder erzählst du den Anfang einer kleinen Geschichte (Beispiel: „Der Prinz reitet an einem wunderschönen Sonntagmorgen auf seinem Pferd aus seinem Schloss hinaus. Nach kurzer Zeit kommt er an eine Wegkreuzung.“). Mit verschiedenen Buttons soll der Benutzer über den weiteren Verlauf der Geschichte entscheiden können (zum Beispiel drei Buttons: „Er reitet nach links“, „Er reitet geradeaus“, „Er reitet nach rechts“).

Bei Klick auf einen Button soll sich ein neues Fenster öffnen, in dem die Geschichte – je nach

vom Benutzer angeklickten Button – weitererzählt wird. Auch soll der Benutzer dann wieder über mehrere Buttons entscheiden können, wie die Geschichte weiter geht.

6 Variablen und Datenverarbeitung

Computer sind Datenverarbeitungsmaschinen: Sie sind dafür da, uns das Leben einfacher zu machen, indem sie zum Beispiel Dinge berechnen oder Datensätze wie Teilnehmerlisten verwalten.

Abbildung 13: Das Addierfenster™

Bisher haben unsere Programme zwar wunderbar über grafische Oberflächen mit dem Benutzer „geredet“. Wirklich etwas gearbeitet haben sie aber noch nicht. Das ändert sich in diesem Kapitel. Wir werden lernen, wie wir in unseren Programmen Eingaben vom Benutzer entgegennehmen, sie verarbeiten und ein Ergebnis ausgeben.

6.1 Ein Programm, das rechnen kann

Dazu schreiben wir als erstes ein minimalistisches Rechenprogramm, das zwei Zahlen addieren kann. Unser Programmfenster soll so aussehen, wie du in Abb. 13 siehst.

Wie du siehst, brauchen wir dafür noch eine neue Art von grafischen Objekten, nämlich Eingabefelder. Diese stellt uns gtk mit der Klasse *Entry* (englisch „Eingabe“) bereit.

Entry ist vorerst die letzte Klasse grafischer gtk-Objekte, die wir im Rahmen dieses Kurses kennenlernen werden. Natürlich gibt es noch viel mehr! Eine vollständige Übersicht findest du auf der *pygtk*-Webseite unter `http://developer.gnome.org/pygtk/stable/`. Diese ist im für Informatiker typischen Stil einer Referenz gehalten und deshalb vielleicht verwirrend. Wenn du Lust hast, schau aber mal rein.

Für diesen Kurs reicht uns unser Wissen über grafische Oberflächen erst einmal. Wir erweitern mit der neu kennen gelernten Klasse *Entry* ein letztes Mal unser gtk-Klassendiagramm (Abb. 14) und konzentrieren uns ab jetzt darauf, Programme wirklich etwas tun, also Daten verarbeiten zu lassen.

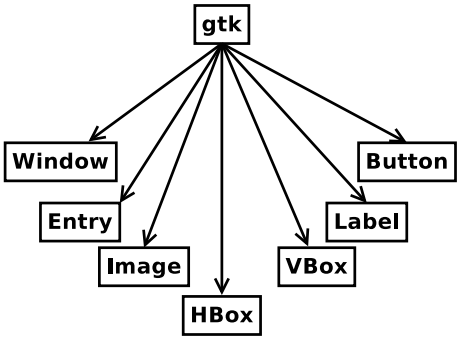

Abbildung 14: Klassendiagramm

Um unser Rechenprogramm zu schreiben, fangen wir zunächst mit der grafischen Oberfläche an. Wir brauchen wieder ein Fenster und, da die sichtbaren Objekte nebeneinander angeordnet werden sollen, auch ein *HBox*-Objekt, das wir dem Fenster hinzufügen. Nun brauchen wir ein Eingabefeld (*Entry*) für den ersten Summanden, ein Textfeld (*Label*) für das „+“, ein Eingabefeld (*Entry*) für den zweiten Summanden, einen „=“-Button und ein zunächst noch leeres Textfeld (*Label*), in das das Programm später das Ergebnis hineinschreiben soll. Diese erstellen wir uns fügen sie dem HBox-Objekt hinzu. Damit ergibt sich eine Objektstruktur, wie im Objektdiagramm in Abb. 15 gezeigt.

Falls du jetzt schonmal in meine Version des Addierprogramms hineinschauen willst, findest du den Quellcode unter Programm 11. In dieser Version schon enthalten ist auch der Code, der unserem Programm wirklich das rechnen beibringt – darum geht es jetzt!

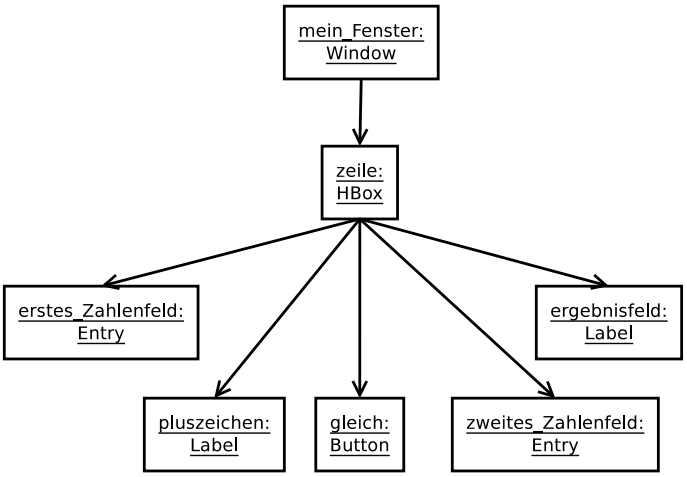

Abbildung 15: Addierprogramm, Objektdiagramm

6.2 Variablen

Um unserem Addierprogramm gleich das Rechnen beizubringen, benötigen wir den Begriff der *Variablen*. Zum Glück wissen wir eigentlich schon, was das ist – wir haben es nur nie genauer betrachtet. Schauen wir uns nochmal unseren guten, alten Befehl an, mit dem wir ein Window-Objekt konstruieren:

```
mein_Fenster = gtk.Window()
```

Den Teil links des Gleichheitszeichens haben wir bisher immer nur als „Namen“ des Objekts bezeichnet. Der korrekte Begriff dafür ist *Variable*.

Eine Variable steht für ein bestimmtes Stück Speicher im Rechner. Technisch gesehen muss unser neues Window-Objekt irgendwo im Speicher des Rechners abgelegt werden. Der Computer reserviert also eine gewisse Menge Speicher – genau so viel, wie für ein Window-Objekt notwendig ist – und konstruiert dann das Objekt in genau diesen Speicherbereich. Genau dieser Speicherbereich, in dem das Objekt dann liegt, wird durch eine *Variable* dargestellt. In unserem Beispiel steht also die Variable mein_Fenster für genau den Bereich im Speicher des Computers, in dem unser neues Fenster abgespeichert wurde.

Anschaulich könnte man sich den Speicher eines Computers als eine Art Schubladenschrank vorstellen. Für jedes neue Objekt wird eine Schublade reserviert und mit einem Namen beschriftet – dem Namen der Variable, wie zum Beispiel mein_Fenster. Abb. 16 zeigt, wie ein solcher „Speicherschrank“ aussehen könnte, nachdem wir die Objekte mein_Fenster und mein_Button angelegt haben. Wenn wir später unser Objekt mein_Fenster benutzen wollen, zum Beispiel mit dem Befehl *mein_Fenster.show()*, kann man sich vorstellen, dass der Rechner in seinem Speicherschrank nach der Variablen – also der Schublade – mit dem Namen mein_Fenster sucht, und auf dem Objekt in genau dieser Schublade dann die Funktion show() ausführt.

Dieses Bild – Speicher als Schrank und Variablen als Schubladen – ist aber mit Vorsicht zu genießen: Natürlich hat unser Computer nicht nur sieben Speicherplätze, und natürlich sind von seinen Speicherplätzen nicht fünf groß und zwei klein. Wenn wir eine neue Variable erstellen, reserviert der Computer genau so viel Speicher, wie für das anzulegende Objekt benötigt wird (in den Speicher passen also viele kleine oder weniger viele große Objekte)[12].

[12] Auch das ist noch eine vereinfachte Darstellung – wie Speicherverwaltung technisch funktioniert, ist ein

6.3 Mein Programm soll endlich rechnen!

Weiter geht's mit unserem angefangenen Addierprogramm: Es fehlt uns noch eine Funktion, die aufgerufen werden soll, wenn der Button gedrückt wird – die Funktion also, die wirklich rechnet.

Definieren wir also eine Funktion, der wir zum Beispiel den Namen *berechnen* geben. Wie wir aus Abschnitt 3.2 wissen, muss diese Funktion einen Parameter akzeptieren – nicht, weil wir ihn verwenden wollen, sondern weil gtk das erwartet.

```
def berechnen(parameter):
```

Wie gehen wir innerhalb dieser Funktion nun vor? Zunächst müssen wir den Inhalt der Eingabefelder auslesen – schließlich wollen wir mit diesen Werten rechnen. Beginnen wir mit dem ersten Eingabefeld, also der ersten Zahl.

Hierfür stellt uns die Klasse Entry (zu der die Eingabefelder ja gehören) die Funktion *get_text()* zur Verfügung. Das Ergebnis dieser Funktion wollen wir in einer Variable speichern, die wir *erste_Zahl_ausgelesen* nennen:

Abbildung 16: Variablen kann man sich als Schubladen vorstellen

Foto: Allen McBride, lizensiert CC-BY-SA

```
erste_Zahl_ausgelesen = erstes_Zahlenfeld.get_text()
```

Vielleicht fällt euch die Ähnlichkeit zu dem Befehl auf, mit dem wir uns immer ein neues Fenster erstellt haben (*mein_Fenster = gtk.Window()*). Links steht der Name einer Variablen – rechts eine Funktion, die uns ein bestimmtes Ergebnis gibt: Hier den Inhalt des Zahlenfelds, im Fall des Fensters das frisch konstruierte Window-Objekt. Das Gleichheitszeichen sagt: Das Ergebnis der Funktion auf der rechten Seite soll in den Speicher gelegt werden, für den die Variable auf der linken Seite steht. Da wir die Variable *erste_Zahl_ausgelesen* bisher nicht in unserem Programm verwendet haben, wird sie automatisch für uns angelegt.

Einen solchen Befehl, der etwas Bestimmtes in eine Variable schreibt, nennt man *Zuweisung*.

recht komplexes Thema. Stichworte wären zum Beispiel die Speicherstrukturen *Stack* und *Heap*. Für diesen Kurs belassen wir es aber bei der vereinfachten Sicht.

Ein Text ist keine Zahl

Sicher ist euch aufgefallen, dass wir den Wert des ersten Zahlenfeldes mit der Funktion *get_**text**()* ausgelesen haben – und mit Text können wir nicht rechnen. Warum handelt es sich bei den ausgelesen Daten um Text, wo der Benutzer doch eine Zahl eingeben soll?

Gibt der Benutzer zum Beispiel „42“ in das Textfeld ein, ist das für den Computer etwas anderes als die Zahl 42: Der Benutzer hat das Zeichen „4“ und das Zeichen „2“ getippt. Also stehen in dem Eingabefeld eben diese beiden Zeichen – und mehrere Zeichen sind ein Text (in Python heißen Texte *str*[13]). Auch Texte sind Objekt: In der Variable *erste_Zahl_ausgelesen* befindet sich nun also ein Objekt der Klasse *str*.

Diese Zeichen, die ja für eine Zahl stehen (in unserem Beispiel „42“), müssen wir nun noch in eine echte Zahl umwandeln, mit der der Computer rechnen kann. Wir haben die Wahl, ob wir den Inhalt des Eingabefeldes in eine Ganzzahl (Klasse *int*[14]), also eine Zahl ohne Komma, oder in eine reelle Zahl (Klasse *float*[15]), also eine Zahl mit Komma, umwandeln wollen. Wir entscheiden uns für die Form mit Komma, damit unser Programm nicht nur ganze Zahlen addieren kann – also *float*.

Also erstellen wir nun eine neue Variable, nennen wir sie *erste_Zahl_als_Zahl*, und legen dort die in eine *float*-Zahl umgewandelte erste Zahl ab. Dazu konstruieren wir ein *float*-Objekt:

```
erste_Zahl_als_Zahl = float(erste_Zahl_ausgelesen)
```

Bei obigem Befehl handelt es sich also – ebenso, wie wenn wir z.B. einen Button erstellen – um einen Konstruktoraufruf. Wenn dich das verwirren sollte, schau dir die Ähnlichkeit noch einmal an! Der Konstruktor der Klasse *float* erlaubt es, in den Funktionsklammern direkt einen Text mitzugeben, aus dem eine Kommazahl erzeugt wird – genauso wie der Konstruktor der Klasse *Button* es erlaubt, in den Funktionsklammern einen Text mitzugeben, der auf dem Button stehen soll.

Berechnen des Ergebnisses

Die erste Zahl haben wir nun also endlich in einer Form, in der wir mit ihr rechnen können. Das gleiche machen wir nun noch mit der zweiten Zahl:

```
zweite_Zahl_ausgelesen = zweites_Zahlenfeld.get_text()
zweite_Zahl_als_Zahl = float(zweite_Zahl_ausgelesen)
```

Jetzt können wir endlich das Ergebnis berechnen. Dazu nehmen wir eine neue Variable *ergebnis* und weisen ihr die Summe aus *erste_Zahl_als_Zahl* und *zweite_Zahl_als_Zahl* zu:

```
ergebnis = erste_Zahl_als_Zahl + zweite_Zahl_als_Zahl
```

Dieser Befehl ist – genau wie schon oben – eine Zuweisung, nur dass wir hier nicht das Ergebnis einer Funktion, sondern das Ergebnis einer Berechnung in die Variable schreiben.

[13] kurz für „String“ (englisch „Schnur“, „Kette“) – ein Text ist also eine „Kette“ aus Zeichen

[14] kurz für „integer“ (englisch „Ganzzahl“)

[15] „float“ (englisch „gleiten“) bedeutet *Gleitkommazahl*. Das ist eine Form Zahl zu speichern, hinter der deutlich mehr steckt, als nur, dass die Zahl ein Komma enthalten kann. Gleitkomma bedeutet zum Beispiel auch, dass Zahlen nicht exakt gespeichert werden, sondern um so *ungenauer*, je größer sie sind. In diesem Kurs gehen wir auf diese Details aber nicht weiter ein.

In der Variable Ergebnis befindet sich jetzt ein Objekt der Klasse *float*, also eine Zahl (mit Komma), da die Summe aus zwei Zahlen (mit Komma) wieder eine Zahl (mit Komma) ist. Um das Ergebnis ins Fenster, in unser Label-Objekt *ergebnisfeld*, schreiben zu können, benötigen wir aber wieder einen Text – ein Objekt der Klasse *str*. Also müssen wir nun den umgekehrten Weg wie oben gehen – ein *float*-Objekt (eine Zahl) in ein *str*-Objekt (einen Text) umwandeln. Dazu benutzen wir eine neue Variable *ergebnis_als_Text*.

Auch von der Klasse *str* können wir ein Objekt konstruieren, und der Konstruktor von *str* erlaubt auch das direkte Konstruieren eines *str*-Objekts aus einem *float*-Objekt. Wir schreiben also:

```
ergebnis_als_Text = str(ergebnis)
```

Schließlich müssen wir diesen Ergebnistext nur noch in unser *Label*-Objekt *ergebnisfeld* schreiben. Hierfür haben Label-Objekte die Funktion *set_text*:

```
ergebnisfeld.set_text(ergebnis_als_Text)
```

Unsere Funktion sieht jetzt also so aus:

```
def berechnen(parameter):
  erste_Zahl_ausgelesen = erstes_Zahlenfeld.get_text()
  erste_Zahl_als_Zahl = float(erste_Zahl_ausgelesen)

  zweite_Zahl_ausgelesen = zweites_Zahlenfeld.get_text()
  zweite_Zahl_als_Zahl = float(zweite_Zahl_ausgelesen)

  ergebnis = erste_Zahl_als_Zahl + zweite_Zahl_als_Zahl
  ergebnis_als_Text = str(ergebnis)
  ergebnisfeld.set_text(ergebnis_als_Text)
```

Unsere Funktion setzen wir jetzt über den Quellcode für die grafische Oberfläche des Addierfensters, den wir schon in Abschnitt 6.1 geschrieben haben. Denk auch daran, dass du deinen Button noch mit der Funktion *connect* mit dieser Funktion verbinden musst, wie wir das schon aus Abschnitt 3.1 kennen.

Programm 11: Das Addierprogramm, komplett

```
1  # Diese Funktion wird aufgerufen,
2  # sobald jemand den =-Knopf drueckt!
3  def berechnen(mein_Parameter):
4    # Eingabe fuer die erste Zahl holen und in Zahl verwandeln:
5    erste_Zahl_ausgelesen = erstes_Zahlenfeld.get_text()
6    erste_Zahl_als_Zahl = float(erste_Zahl_ausgelesen)
7
8    # Eingabe fuer die zweite Zahl holen und in Zahl verwandeln:
9    zweite_Zahl_ausgelesen = zweites_Zahlenfeld.get_text()
10   zweite_Zahl_als_Zahl = float(zweite_Zahl_ausgelesen)
11
12   # Ergebnis berechnen und in Text verwandeln:
13   ergebnis = erste_Zahl_als_Zahl + zweite_Zahl_als_Zahl
14   ergebnis_als_Text = str(ergebnis)
```

```
15
16    # Ergebnis ins Ergebnisfeld schreiben:
17    ergebnisfeld.set_text(ergebnis_als_Text)
18
19  # Ab hier kommt das Hauptprogramm:
20  import gtk
21
22  # Als erstes brauchen wir ein Fenster:
23  mein_Fenster = gtk.Window()
24  mein_Fenster.show()
25  mein_Fenster.set_title("Addierfenster")
26
27  # In dem Fenster sollen Elemente nebeneinander angeordnet werden
        .
28  # Das schreit nach einer HBox:
29  zeile = gtk.HBox()
30  zeile.show()
31  mein_Fenster.add(zeile) # zeile dem Fenster hinzufuegen
32
33  # Eingabefeld fuer die erste Zahl:
34  erstes_Zahlenfeld = gtk.Entry()
35  erstes_Zahlenfeld.show()
36  zeile.add(erstes_Zahlenfeld) # Eingabefeld in die Zeile
37
38  # Pluszeichen als Text:
39  pluszeichen = gtk.Label("+")
40  pluszeichen.show()
41  zeile.add(pluszeichen)
42
43  # Eingabefeld fuer die zweite Zahl:
44  zweites_Zahlenfeld = gtk.Entry()
45  zweites_Zahlenfeld.show()
46  zeile.add(zweites_Zahlenfeld)
47
48  # Gleichheitszeichen als Button:
49  gleich = gtk.Button("=")
50  gleich.show()
51  zeile.add(gleich)
52
53  # Wenn der Button geklickt wird, Funktion berechnen ausfuehren!
54  gleich.connect("clicked", berechnen)
55
56  # Textobjekt fuer Ergebnis:
57  ergebnisfeld = gtk.Label()
58  ergebnisfeld.show()
59  zeile.add(ergebnisfeld)
60  gtk.main()
```

6.4 Aufgabe: Ein richtiger Rechner

Bring unserem Addierprogramm weitere Rechenoperationen bei – zum Beispiel Subtraktion, Multiplikation und/oder Division!

6.5 Beispiel BMI-Rechner

Als Nächstes schreiben wir ein Programm zur Berechnung des *Body Mass Index* (BMI) einer Person. Das ist eine Zahl, die aus der Größe und dem Gewicht eines Menschen berechnet wird und als grobes Maß dafür gilt, ob diese Person Über- oder Untergewicht hat. Mehr Informationen zum BMI findest du zum Beispiel in Wikipedia unter `http://de.wikipedia.org/wiki/Body-Mass-Index`.

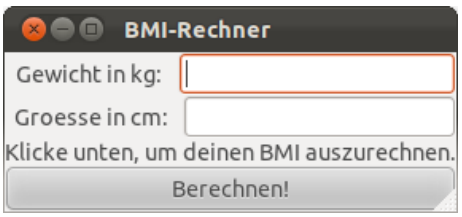

Abbildung 17: Der BMI-Rechner

Man berechnet den BMI, indem man die Körpergröße in Metern quadriert und dann das Gewicht in kg durch dieses Quadrat der Körpergröße teilt:

$$BMI = \frac{\text{Körpergewicht in kg}}{(\text{Körpergröße in m})^2}$$

Man geht grob davon aus, dass ein BMI zwischen 18,5 und 25 für Normalgewicht, ein BMI unter 18,5 für Untergewicht und ein BMI über 25 für Übergewicht steht.

Alles, was wir zum Schreiben dieses Programms brauchen, haben wir schon kennengelernt! Deshalb fassen wir hier nur noch kurz zusammen, wie wir vorgehen müssen. Dabei nehmen wir an, dass unser Programmfenster so aussehen soll wie in Abb.17 – du kannst natürlich auch ein anderes Layout wählen!

Erst einmal gestalten wir die grafische Benutzeroberfläche:

1. Wie immer erstellen wir zunächst ein Fenster und geben ihm einen Titel.
2. In dem Fenster sollen Dinge in mehreren Zeilen *untereinander* stehen. Also brauchen wir zunächst ein *VBox*-Objekt.
3. Für die erste Zeile (Eingabe des Gewichts) brauchen wir ein *HBox*-Objekt. In dieses HBox-Objekt stecken wir zunächst ein neues *Label*-Objekt (mit dem Text „Gewicht in kg:“) und dann ein *Entry*-Objekt (Eingabefeld fürs Gewicht).
4. Für die zweite Zeile (Eingabe der Größe) brauchen wir ebenfalls ein neues *HBox*-Objekt. Hier kommt zunächst wieder ein neues *Label*-Objekt hinein (mit dem Text „Groesse in cm:“) und dann wieder ein neues *Entry*-Objekt (Eingabefeld für die Größe).
5. Die dritte „Zeile“ besteht nur aus einem *Label*-Objekt. Dieses zeigt zunächst einen kurzen Infotext an („Klicke unten, um deinen BMI auszurechnen“) – später soll hier das Ergebnis stehen. Wir brauchen für die dritte Zeile also *kein* HBox-Objekt: Es sollen ja nicht mehrere Dinge nebeneinander stehen, sondern nur das eine Label-Objekt. Wir erstellen also ein Label und fügen es direkt unserem VBox-Objekt hinzu.

6. Auch die letzte Zeile besteht nur aus einem einzigen Objekt – nämlich dem „Berechnen¡'-Button. Also erstellen wir ein neues *Button*-Objekt und fügen es dem VBox-Objekt hinzu.

Anschließend erstellen wir die Funktion, die die Berechnung durchführt:

1. Wir entnehmen den beiden Eingabefeldern mit deren Funktion *get_text* ihre Daten und speichern sie in zwei neuen Variablen.
2. Diese Daten liegen uns als Text – als *str*-Objekte – vor. Wir wandeln sie also zunächst in Zahlen (mit Komma), in *float*-Objekte, um. Die umgewandelten Werte speichern wir wieder in zwei neuen Variablen.
3. Da uns die Eingaben nun als Zahlen vorliegen, können wir jetzt die eigentliche Berechnung nach der Formel durchführen. Vorsicht: In meiner Version des Programms habe ich den Benutzer aufgefordert seine Größe in cm einzugeben, die Formel erwartet aber die Größe in Metern. Die Eingabe des Benutzers müssen wir also vorab durch 100 teilen.
4. Das Ergebnis der Berechnung, das uns natürlich wieder als *float*-Zahl vorliegt, wandeln wir nun wieder in ein *str*-Objekt (also in Text) um.
5. Diese in Text umgewandelte Form des Ergebnisses schreiben wir schließlich mit der Funktion *set_text* ins Ergebnisfeld.

Probiere, den BMI-Rechner selbstständig zu programmieren! Meine Version findest du unter Programm 12.

Programm 12: Der BMI-Rechner

```
1  import gtk
2
3  def berechnen(parameter):
4    # Diese Funktion wird aufgerufen, wenn jemand den
5    # Berechnen-Knopf drueckt.
6    # Sie liest Gewicht und Groesse aus dem Fenster aus,
7    # berechnet den BMI, und schreibt den dann wieder ins Fenster.
8
9    # Erstmal das Gewicht holen und in eine Zahl umwandeln.
10   gewicht = gewichtEingabe.get_text()
11   gewichtAlsZahl = float(gewicht)
12
13   # Jetzt die Groesse holen und in eine Zahl umwandeln.
14   # Die Groesse muss durch 100 geteilt werden, weil der
15   # Benutzer sie in cm eingibt, wir fuer die Formel aber
16   # die Groesse in Metern brauchen.
17   groesse = groesseEingabe.get_text()
18   groesseAlsZahl = float(groesse) / 100
19
```

```
20    # BMI berechnen und wieder in einen Text umwandeln:
21    bmi = gewichtAlsZahl / (groesseAlsZahl * groesseAlsZahl)
22    bmiAlsText = str(bmi)
23
24    # Den fertig berechneten und umgewandelten BMI
25    # in das BMItext-Objekt unseres Fensters schreiben:
26    ergebnisWert.set_text(bmiAlsText)
27
28
29 # Wir brauchen ein Hauptfenster:
30 meinFenster = gtk.Window()
31 meinFenster.set_title("BMI-Rechner")
32
33 # Unser Programmfenster soll aus mehreren Zeilen von Buttons
34 # und anderen Elementen bestehen
35 untereinander = gtk.VBox()
36 untereinander.show()
37 meinFenster.add(untereinander)
38
39 # Erste Zeile: Hier soll der Benutzer sein Gewicht eingeben
40 zeileGewicht = gtk.HBox()
41 zeileGewicht.show()
42 untereinander.add(zeileGewicht)
43
44 gewichtText = gtk.Label("Gewicht in kg: ") # Erklaerungstext
45 gewichtText.show()
46 zeileGewicht.add(gewichtText)
47
48 gewichtEingabe = gtk.Entry() # Eingabefeld
49 gewichtEingabe.show()
50 zeileGewicht.add(gewichtEingabe)
51
52 # Zweite Zeile: Hier soll der Benutzer seine Groesse eingeben
53 zeileGroesse = gtk.HBox()
54 zeileGroesse.show()
55 untereinander.add(zeileGroesse)
56
57 groesseText = gtk.Label("Groesse in cm: ") # Erklaerungstext
58 groesseText.show()
59 zeileGroesse.add(groesseText)
60
61 groesseEingabe = gtk.Entry() # Eingabefeld
62 groesseEingabe.show()
63 zeileGroesse.add(groesseEingabe)
64
65 # Dritte Zeile: Hier soll spaeter das Ergebnis stehen,
66 # am Anfang nur ein Infotext
```

```
67 zeileErgebnis = gtk.HBox()
68 zeileErgebnis.show()
69 untereinander.add(zeileErgebnis)
70
71 ergebnisText = gtk.Label("Dein BMI ist: ")
72 ergebnisText.show()
73 zeileErgebnis.add(ergebnisText)
74
75 ergebnisWert = gtk.Label("  "); # am Anfang noch leer
76 ergebnisWert.show()
77 zeileErgebnis.add(ergebnisWert)
78
79 # Vierte Zeile: Der Berechnen-Button
80 buttonBerechnen = gtk.Button("Berechnen!")
81 buttonBerechnen.show()
82 untereinander.add(buttonBerechnen)
83 # Diese "Zeile" besteht nur aus einem Button,
84 # braucht also keine extra HBox-Objekt
85
86 # Button seine Funktion geben:
87 buttonBerechnen.connect("clicked", berechnen)
88
89 # Fenster anzeigen und Programmkontrolle an GTK abgeben
90 meinFenster.show()
91 gtk.main()
```

7 Verzweigungen

Wir haben in diesem Kurs zuerst gelernt, wie wir Programme schreiben, die einfach immer stur von oben nach unten durchlaufen. In Kapitel 3 haben wir außerdem kennen gelernt, wie wir auf Ereignisse (wie zum Beispiel den Klick auf einen Button) mit eigenen Funktionen reagieren können. Innerhalb dieser Funktionen laufen unsere Programme aber immer noch stur von oben nach unten ab.

Das ist viel zu unflexibel! Nehmen wir dazu wieder unser Beispiel *BMI-Rechner*: Dieser soll jetzt nicht mehr nur den BMI als Zahl anzeigen, sondern dem Benutzer auch sofort anzeigen, ob er mit diesem BMI Normalgewicht hat, oder ob er zu leicht oder zu schwer ist. Das lässt sich nicht sinnvoll mit Ereignissen machen – schließlich hängt diese Frage ja von dem Ergebnis unseres Programms selbst (nämlich dem berechneten BMI) und nicht etwa von einem Klick des Nutzers oder ähnlichem ab[16].

[16]Tatsächlich lässt sich auch ein solches Problem *ereignisorientiert* lösen – indem unser Programm nach der Berechnung des Ergebnisses selbst ein *Ereignis* auslöst, durch das dann wiederum eine Funktion aufgerufen wird. Sich immer nur auf die Ereignisorientierung zu verlassen, macht das Programmieren aber ziemlich umständlich und ist deshalb nicht üblich.

7.1 BMI-Rechner mit Auswertung

Wir wollen unseren BMI-Rechner (Programm 12) dazu bringen, uns zu sagen, ob wir zulegen oder abspecken müssen! Dazu fügen wir zunächst der grafischen Oberfläche irgendwo ein neues Label-Objekt hinzu, in dem wir die Empfehlung später anzeigen. (Das kriegst du alleine hin!)

Für den weiteren Text gehen wir davon aus, dass wir dieses Label-Objekt *auswertung* genannt haben. Diesem Label-Objekt müssen wir jetzt je nach errechnetem BMI – also je nachdem, welche Zahl in der Variablen *bmi* steht – einen anderen Text zuweisen: Unter 18,5 müssen wir so etwas wie „Du bist zu dünn!“ ausgeben, zwischen 18,5 und 25 zum Beispiel „Alles im grünen Bereich!“ und über 25 „Du hast Übergewicht!“.

Wir brauchen also drei verschiedene Befehle, die den Text unseres *auswertung*-Objekts setzen, von denen je nach Situation immer nur einer ausgeführt werden soll. Eine solche *Verzweigung* realisiert man mit dem Befehl **if** (englisch: „wenn“).

Wir schreiben ans Ende unserer *berechnen*-Funktion zunächst den Befehl **if**. Dahinter formulieren wir die Bedingung für Untergewicht: $bmi < 18.5$. Anschließend folgt ein Doppelpunkt. Alle Befehle, die nur unter eben dieser Bedingung ausgeführt werden sollen, müssen wir darunter *eingerückt* schreiben.

Das erinnert dich vielleicht an die Schreibweise, mit der wir Funktionen definiert haben: In beiden Fällen zeigt die Einrückung, dass diese Befehle besonders sind und nicht einfach stumpf von oben nach unten abgearbeitet werden sollen. Im Falle einer Funktion sollen sie erst bei deren Aufruf, im Falle der Verzweigung nur unter der genannten Bedingung ausgeführt werden.

Unsere erste Ergänzung zum BMI-Rechner sieht also so aus:

```
if bmi < 18.5:
  auswertung.set_label("Du bist zu duenn!")
```

Analog dazu fügen wir darunter noch die beiden weiteren möglichen Fälle ein. Für den „Normalgewicht“-Fall muss der BMI zwischen 18,5 und 25 liegen. Das bedeutet, zwei Bedingungen müssen gleichzeitig erfüllt sein: Der BMI muss größer/gleich 18,5 *und* kleiner/gleich 25 sein.

```
if bmi >= 18.5 and bmi <= 25:
  auswertung.set_label("Alles im gruenen Bereich!")
if bmi > 25:
  auswertung.set_label("Du hast Uebergewicht!")
```

Den BMI-Rechner mit diesen Ergänzungen findest du unter Programm 13.

Programm 13: Der BMI-Rechner, mit Auswertung

```
1 import gtk
2
3 def berechnen(parameter):
4   # Diese Funktion wird aufgerufen, wenn jemand den
```

```
5   # Berechnen-Knopf drueckt.
6   # Sie liest Gewicht und Groesse aus dem Fenster aus,
7   # berechnet den BMI, und schreibt den dann wieder ins Fenster.
8
9   # Erstmal das Gewicht holen und in eine Zahl umwandeln.
10  gewicht = gewichtEingabe.get_text()
11  gewichtAlsZahl = float(gewicht)
12
13  # Jetzt die Groesse holen und in eine Zahl umwandeln.
14  # Die Groesse muss durch 100 geteilt werden, weil der
15  # Benutzer sie in cm eingibt, wir fuer die Formel aber
16  # die Groesse in Metern brauchen.
17  groesse = groesseEingabe.get_text()
18  groesseAlsZahl = float(groesse) / 100
19
20  # BMI berechnen und wieder in einen Text umwandeln:
21  bmi = gewichtAlsZahl / (groesseAlsZahl * groesseAlsZahl)
22  bmiAlsText = str(bmi)
23
24  # Den fertig berechneten und umgewandelten BMI
25  # in das BMItext-Objekt unseres Fensters schreiben:
26  ergebnisWert.set_text(bmiAlsText)
27
28  # Auswertung ins Fenster schreiben:
29  if bmi < 18.5:
30    auswertung.set_label("Du bist zu duenn!")
31  if bmi >= 18.5 and bmi <= 25:
32    auswertung.set_label("Alles im gruenen Bereich!")
33  if bmi > 25:
34    auswertung.set_label("Du hast Uebergewicht!")
35
36 # Wir brauchen ein Hauptfenster:
37 meinFenster = gtk.Window()
38 meinFenster.set_title("BMI-Rechner")
39
40 # Unser Programmfenster soll aus mehreren Zeilen von Buttons
41 # und anderen Elementen bestehen
42 untereinander = gtk.VBox()
43 untereinander.show()
44 meinFenster.add(untereinander)
45
46 # Erste Zeile: Hier soll der Benutzer sein Gewicht eingeben
47 zeileGewicht = gtk.HBox()
48 zeileGewicht.show()
49 untereinander.add(zeileGewicht)
50
51 gewichtText = gtk.Label("Gewicht in kg: ") # Erklaerungstext
```

```
52 gewichtText.show()
53 zeileGewicht.add(gewichtText)
54
55 gewichtEingabe = gtk.Entry() # Eingabefeld
56 gewichtEingabe.show()
57 zeileGewicht.add(gewichtEingabe)
58
59 # Zweite Zeile: Hier soll der Benutzer seine Groesse eingeben
60 zeileGroesse = gtk.HBox()
61 zeileGroesse.show()
62 untereinander.add(zeileGroesse)
63
64 groesseText = gtk.Label("Groesse in cm: ") # Erklaerungstext
65 groesseText.show()
66 zeileGroesse.add(groesseText)
67
68 groesseEingabe = gtk.Entry() # Eingabefeld
69 groesseEingabe.show()
70 zeileGroesse.add(groesseEingabe)
71
72 # Dritte Zeile: Hier soll spaeter das Ergebnis stehen,
73 # am Anfang nur ein Infotext
74 zeileErgebnis = gtk.HBox()
75 zeileErgebnis.show()
76 untereinander.add(zeileErgebnis)
77
78 ergebnisText = gtk.Label("Dein BMI ist: ")
79 ergebnisText.show()
80 zeileErgebnis.add(ergebnisText)
81
82 ergebnisWert = gtk.Label("  "); # am Anfang noch leer
83 ergebnisWert.show()
84 zeileErgebnis.add(ergebnisWert)
85
86 # Vierte Zeile: Hier soll spaeter die Auswertung stehen
87 auswertung = gtk.Label()
88 auswertung.show()
89 untereinander.add(auswertung)
90 # Diese "Zeile" besteht nur aus einem Button,
91 # braucht also keine extra HBox-Objekt
92
93 # Fuenfte Zeile: Der Berechnen-Button
94 buttonBerechnen = gtk.Button("Berechnen!")
95 buttonBerechnen.show()
96 untereinander.add(buttonBerechnen)
97
98 # Button seine Funktion geben:
```

```
99 buttonBerechnen.connect("clicked", berechnen)
100
101 # Fenster anzeigen und Programmkontrolle an GTK abgeben
102 meinFenster.show()
103 gtk.main()
```

7.2 Einschub: Das imperative Paradigma

... haben wir gerade kennen gelernt! Schon in Kapitel 3.5 haben wir den Begriff „Paradigma“ benutzt, um die zwei Arten zu beschreiben, auf die wir bis dahin programmieren konnten: *Objektorientiert* (wir teilen unser Programm auf in viele kleine Objekte, die jeweils etwas ganz bestimmtes haben und können) und *ereignisorientiert* (wenn ein bestimmtes Ergeignis, wie z.B. ein Klick auf einen Button, eintritt, soll eine bestimmte Funktion ausgeführt werden).

In diesem Kapitel haben wir gemerkt, dass wir damit alleine bestimmte Probleme nicht (sinnvoll) lösen können. Mit Verzweigungen haben wir nun ein wesentliches Element der *imperativen* Programmierung kennen gelernt: Wir befehlen (vgl. Bedeutung des Wortes „Imperativ“: Befehlsform) dem Computer direkt, an einer ganz bestimmten Stelle eine ganz bestimmte Bedingung abzuprüfen und je nach Ergebnis ganz bestimmte Befehle auszuführen.

Die imperative Programmierung liegt im Gegensatz zu den anderen Paradigmen deutlich näher an dem, wie ein Computer intern funktioniert. Imperative Konstrukte lassen sich also am direktesten in Maschinensprache übersetzen.

An unserem Beispielprogrammen sehen wir, wie gut sich verschiedene Paradigmen ergänzen: Unser BMI-Rechner enthält nun sowohl objektorientierte und ereignisorientierte als auch imperative Programmier-Konstrukte.

7.3 Beispiel Morse-Codierer

Der Morsecode ist eine Form, in der Nachrichten bzw. Texte über Funk übermittelt werden können (und war besonders in der Anfangszeit der Funktechnik, bevor Sprachfunk sinnvoll möglich wurde, die einzige verwendete Form). Jeder Buchstabe wird durch eine eindeutige Folge von kurzen Tönen („Dit“) und langen Tönen („Dah“) dargestellt. Am Bekanntesten ist sicher das aus dem Morsecode hervorgegangene Notrufsignal SOS: Drei kurz, drei lang, drei kurz.

Mehr Informationen zum Morsecode und das Morsealphabet findest du zum Beispiel auf Wikipedia: `http://de.wikipedia.org/wiki/Morsecode`

Zur Übung wollen wir ein Programm schreiben, dass einen vom Benutzer eingegebenen Buchstaben in den Morsecode umwandelt. Im nächsten Kapitel werden wir dieses Programm dann ausbauen, so dass es ganze Texte in Morse übersetzen kann. Wie im Geschriebenen üblich soll das Ergebnis dabei mit einem Punkt („.“) für einen kurzen Ton und einem Strich („-“) für einen langen Ton auf dem Bilschirm angezeigt werden.

Wir beginnen wie immer mit einer grafischen Oberfläche: Wir brauchen ein Eingabefeld (*Entry*), ein Ausgabefeld (*Label* oder alternativ noch ein *Entry*) sowie einen Button, der die Umwandlung auslöst. Wie wir es kennen, verknüpfen wir anschließend den Button mit einer Funktion, die ausgeführt werden soll, wenn der Button geklickt wird.

Nehmen wir an, unsere Funktion heißt *codieren*, das Eingabefeld heißt ganz einfach *eingabefeld* und das Ausgabefeld heißt *ergebnisfeld*. Als erstes müssen wir in der Funktion wie immer die Eingabe auslesen. Unsere Funktion *codieren* beginnt also so:

Abbildung 18: Der Morse-Codierer

```
def codieren(parameter):
  eingabe =
eingabefeld.get_text()
```

Reden wir kurz über Datentypen: Wenn der Benutzer etwas in ein Eingabefeld schreibt, hat das, wie wir wissen, den Datentyp Zeichenkette – *str*. Für den Morsecodierer müssen wir im Gegensatz zum BMI-Rechner diese Eingabe *nicht* erst in eine Zahl umwandeln – wir wollen dieses Mal ja durchaus mit Buchstaben arbeiten. Allerdings können wir nur jeweils ein einzelnes Zeichen in Morsecode umwandeln und nicht eine ganze Zeichen*kette*. Der Benutzer könnte ja auch mehrere Zeichen eingegeben haben – zum Beispiel das Wort „Handtuch“. Aber selbst, wenn der Benutzer nur einen einzigen Buchstaben eingegeben hat, sind *Zeichenkette* und einzelnes *Zeichen* in Python grundsätzlich verschiedene Datentypen.

Wir müssen uns also aus dieser Zeichenkette, die sich jetzt in der Variablen *eingabe* befindet, das erste Zeichen herausnehmen. Das geht, indem wir hinter den Variablennamen – eingabe – in eckigen Klammern eine 0 schreiben: [0]. Die eckigen Klammern bedeuten dabei so viel wie „gib mir ein einzelnes Zeichen“ und die Null steht für das erste Zeichen (Informatiker sind komisch und fangen immer bei Null an zu zählen anstatt bei Eins). „[1]“ würde dementsprechend für das zweite Zeichen stehen.

Wir entnehmen also der Zeichenkette *eingabe* das erste Zeichen und weisen das Ergebnis einer Variablen zu:

```
  buchstabe = eingabe[0]
```

Jetzt sind wir endlich so weit, dass wir den eingegebenen Buchstaben in Morsecode umwandeln können! Hat der Benutzer zum Beispiel ein „A“ eingegeben, soll das Ergebnis unserer Codierung „.-“ sein. Ist es dagegen ein „B“, müssen wir uns „-...“ als Ergebnis merken. Der in unserem Programm jetzt folgende Umwandlungsbefehl hängt also davon ab, was eingegeben wurde: Wir müssen die Eingabe überprüfen und je nach Wert einen von verschiedenen Befehlen ausführen.

Wie wir in Abschnitt 7.1 gelernt haben, beginnt eine solche Prüfung immer mit dem Schlüsselwort **if** und dahinter einer zu prüfenden Bedingung. Fangen wir mit dem Fall an, dass ein „A“ eingegeben wurde: Dann lautet die Bedingung *buchstabe == "A"*.

Beachte das doppelte Gleichheitszeichen: Wir wir wissen, ist das einfache Gleichheitszeichen in Python schon mit etwas anderem belegt, nämlich der *Zuweisung*. Der *Vergleich* braucht dementsprechend ein anderes Zeichen, wofür das doppelte Gleichheitszeichen gewählt wurde. Ein einfaches „=“ bedeutet in Python also den Befehl „mach das gleich!“ während das doppelte „==“ für die Frage „ist das gleich?“ steht.

Um unserem Programm beizubringen, wie man ein A codiert, schreiben wir unsere Funktion also wie folgt weiter:

```
if buchstabe == "A":
  ergebnis = ".-")
```

Für die anderen Buchstaben von B bis Z sowie für die Ziffern gehen wir genauso vor – wir fügen also für jedes andere mögliche Zeichen einen weiteren *if*-Block zu unserer Funktion hinzu. Am Ende schreiben wir das Ergebnis schließlich ins Ergebnisfeld, wie wir das auch in unseren vorherigen Programmen schon immer gemacht haben.

Einen Schönheitsfehler beheben wir noch: Unser Programm wandelt im Moment nur Großbuchstaben um. Das liegt daran, dass z.B. ein großes „A“ und eine kleines „a“ unterschiedliche Zeichen sind. Wir fragen in unseren *if*-Bedingungen aber nur Großbuchstaben ab.

Es gibt also zwei Möglichkeiten: Die umständlichere ist, in jeder *if*-Bedingung einzeln nach dem Groß- und dem Kleinbuchstaben zu fragen – die Frage, die wir der Computer beantworten soll, wäre für das A also: „Ist der Buchstabe ein großes A oder *(englisch: „or“)* ist der Buchstabe ein kleines A“? In Python-Code:

```
if buchstabe == "A" or buchstabe == "a":
```

Die einfachere Lösung ist aber, einfach vor der Umwandlung in Morsecode die Eingabe komplett in Großbuchstaben zu verwandeln. Pyhon hilft uns dabei! Wir erinnern uns daran, dass auch Zeichenketten *Objekte* sind – nämlich Objekte der Klasse *str*. Und wie es sich für Objekte gehört, können auch str-Objekte nützliche Dinge: Sie besitzen die Funktion *upper* (kurz für uppercase, englisch: „in Großbuchstaben“). Diese erstellt ein neues str-Objekt mit dem gleichen Inhalt, aber komplett in Großbuchstaben. Der Befehl zum „groß machen“ sieht also so aus:

```
gross = eingabe.upper()
```

Natürlich müssen wir dann daran denken, uns auch wirklich den ersten Buchstaben des großgeschriebenen Textes für die Morse-Codierung zu holen – und nicht etwa weiterhin den ersten Buchstaben der ursprünglichen Eingabe. In dem Befehl, der den Teil „eingabe[0]“ enthält, müssen wir stattdessen also „gross[0]“ schreiben.

Zusammengefasst sieht unsere Funktion jetzt also so aus:

```
def codieren(parameter):
  eingabe = eingabefeld.get_text()
  gross = eingabe.upper()
  buchstabe = gross[0]
  if buchstabe == "A":
    ergebnis = ".-"
  if buchstabe == "B":
    ergebnis = "-..."
  if buchstabe == "C":
```

```
    ergebnis = "-.-."
  # ...
  # und so weiter fuer alle Buchstaben und Ziffern
  ergebnisfeld.set_text(ergebnis)
```

Das komplette Programm findest du unter Programm 14.

Programm 14: Morse-Codierer für einzelne Zeichen

```
1  def codieren(parameter):
2    eingabe = eingabefeld.get_text()
3    gross = eingabe.upper()
4    buchstabe = gross[0]
5    if buchstabe == "A":
6      ergebnis = ".-"
7    if buchstabe == "B":
8      ergebnis = "-..."
9    if buchstabe == "C":
10     ergebnis = "-.-."
11   if buchstabe == "D":
12     ergebnis = "-.."
13   if buchstabe == "E":
14     ergebnis = "."
15   if buchstabe == "F":
16     ergebnis = "..-."
17   if buchstabe == "G":
18     ergebnis = "--."
19   if buchstabe == "H":
20     ergebnis = "...."
21   if buchstabe == "I":
22     ergebnis = ".."
23   if buchstabe == "J":
24     ergebnis = ".---"
25   if buchstabe == "K":
26     ergebnis = "-.-"
27   if buchstabe == "L":
28     ergebnis = ".-.."
29   if buchstabe == "M":
30     ergebnis = "--"
31   if buchstabe == "N":
32     ergebnis = "-."
33   if buchstabe == "O":
34     ergebnis = "---"
35   if buchstabe == "P":
36     ergebnis = ".--."
37   if buchstabe == "Q":
38     ergebnis = "--.-"
39   if buchstabe == "R":
```

```
40      ergebnis = ".-."
41    if buchstabe == "S":
42      ergebnis = "..."
43    if buchstabe == "T":
44      ergebnis = "-"
45    if buchstabe == "U":
46      ergebnis = "..-"
47    if buchstabe == "V":
48      ergebnis = "...-"
49    if buchstabe == "W":
50      ergebnis = ".--"
51    if buchstabe == "X":
52      ergebnis = "-..-"
53    if buchstabe == "Y":
54      ergebnis = "-.--"
55    if buchstabe == "Z":
56      ergebnis = "--.."
57    if buchstabe == "0":
58      ergebnis = "-----"
59    if buchstabe == "1":
60      ergebnis = ".----"
61    if buchstabe == "2":
62      ergebnis = "..---"
63    if buchstabe == "3":
64      ergebnis = "...--"
65    if buchstabe == "4":
66      ergebnis = "....-"
67    if buchstabe == "5":
68      ergebnis = "....."
69    if buchstabe == "6":
70      ergebnis = "-...."
71    if buchstabe == "7":
72      ergebnis = "--..."
73    if buchstabe == "8":
74      ergebnis = "---.."
75    if buchstabe == "9":
76      ergebnis = "----."
77    ergebnisfeld.set_text(ergebnis)
78
79  import gtk
80
81  fenster = gtk.Window()
82  fenster.show()
83  fenster.set_title("Morse-Codierer")
84
85  untereinander = gtk.VBox()
86  untereinander.show()
```

```
87 fenster.add(untereinander)
88
89 zeile = gtk.HBox()
90 zeile.show()
91 untereinander.add(zeile)
92
93 eingabefeld = gtk.Entry()
94 eingabefeld.show()
95 zeile.add(eingabefeld)
96
97 go = gtk.Button("Codieren")
98 go.show()
99 zeile.add(go)
100 go.connect("clicked", codieren)
101
102 ergebnisfeld = gtk.Label("Gib ein Zeichen ein und klicke auf
      Codieren")
103 ergebnisfeld.show()
104 untereinander.add(ergebnisfeld)
105
106 gtk.main()
```

8 Die *for*-Schleife

Ausgehend von unserem Beispiel *Morse-Codierer* in Kapitel 7.3 ist es leicht zu erklären, was der Programmierer unter einer Schleife versteht: Bisher kann unser Programm nur ein einzelnes Zeichen in Morse codieren. Besser wäre aber ein Programm, das ein ganzes Wort oder sogar einen ganzen Text am Stück codieren kann.

Um das zu erreichen, müssen wir eigentlich nicht viel mehr programmieren, als wir schon haben: Ein Text besteht aus einzelnen Zeichen, und jedes einzelne Zeichen kann unser Programm ja schon codieren. Wir müssen ihm also nur beibringen, diese Umwandlung mehrfach hintereinander durchzuführen – für jedes eingegebene Zeichen einmal. Genau das leistet die *for*-Schleife[17].

Der Befehl für eine *for*-Schleife sieht so aus:

```
for buchstabe in zeichenkette:
```

Dabei muss *zeichenkette* ein existierendes *str*-Objekt[18] sein. Diesen Befehl nennt man den *Schleifenkopf*.

Wie bei Funktionen und *if*-Verzweigungen werden auch hier alle folgenden Befehle, die zur Schleife gehören sollen, *eingerückt* geschrieben. Alle Befehle, die zur Schleife gehören, bilden den *Schleifenkörper*.

[17] Falls du schon andere Programmiersprachen wie z.B. C++, Java oder PHP kennst: Die *for*-Schleife in Python funktioniert ein bisschen anders. Sie entspricht der *foreach*-Schleife in PHP.

[18] Genauer gesagt funktioniert das mit jedem Sequenz-Typ, also z.B. auch mit einer Liste oder einem Array.

Alle Befehle in der Schleife (also der Schleifenkörper) werden jetzt für jedes Zeichen in der *zeichenkette* einmal ausgeführt. Bei jedem Durchlauf stellt die Schleife dabei ein anderes Zeichen der *zeichenkette* unter dem Variablennamen *buchstabe* zur Verfügung – erst das erste, dann das zweite und so weiter.

Wenn wir diese Schleife in unseren Morse-Codierer einbauen, bedeutet das also: Unsere Umwandlungs-*if*-Anweisungen werden so oft ausgeführt, wie es Zeichen im Eingabetext gibt. In jedem Durchlauf steht in der Variablen *buchstabe* (in die wir in der alten Version immer fest den ersten Buchstaben gespeichert haben) ein anderer Buchstabe des Eingabetextes. Buchstabe für Buchstabe codiert das Programm also den Text.

Übrigens: Die Variablen-Namen im Schleifenkopf sind natürlich frei wählbar. Statt *zeichenkette* kannst du jede Variable benutzen, die eine Zeichenkette enthält. Statt *buchstabe* kannst du dir einen beliebigen Namen aussuchen – du solltest nur im Schleifenkörper den gleichen verwenden!

8.1 Der Morse-Codierer für ganze Texte

Bauen wir die Schleife also in unseren Morse-Codierer ein! Zwei Dinge müssen wir dabei noch beachten und im Programm ändern.

Es ist sicher nicht sinnvoll, der Variablen *ergebnis* in jedem Durchlauf einen komplett neuen Wert zuzuweisen – und damit die davor codierten Zeichen alle wieder wegzuschmeißen. Stattdessen beginnen wir *vor* der Schleife mit einer leeren *ergebnis*-Zeichenkette und fügen ihr *in* der Schleife in jedem Durchlauf nur das gerade erzeugte Ergebnis hinzu. Eine leere Zeichenkette erzeugen wir vor der Schleife so:

```
ergebnis = ""
```

Zeichenketten lassen sich in Python ganz einfach mit + aneinanderhängen. In der Schleife fügen wir der Variablen *ergebnis* also wie folgt etwas hinzu, hier am Beispiel eines codierten „A“:

```
    ergebnis = ergebnis + ".-"
```

Eine Schönheitskorrektur sollten wir außerdem noch vornehmen: Damit man das Ergebnis auch lesen kann (wenn man denn Morse kann), setzen wir hinter jedes Morsezeichen noch ein Leerzeichen. Außerdem „codieren“ wir ein eingegebenes Leerzeichen mit drei Leerzeichen.

Unten siehst du die fertige Version der Funktion *codieren*, mit eingebauter Schleife und den besprochenen Änderungen. Überlege zuerst selbst, bevor du dir die Lösung durchliest!

```
def codieren(parameter):
  eingabe = eingabefeld.get_text()
  gross = eingabe.upper()
  ergebnis = ""
  for buchstabe in gross:
    if buchstabe == "A":
      ergebnis = ergebnis + ".- "
```

```
    if buchstabe == "B":
      ergebnis = ergebnis + "-... "
    if buchstabe == "C":
      ergebnis = ergebnis + "-.-. "
    # ...
    # und so weiter fuer alle Buchstaben und Ziffern
    if buchstabe == " ":
      ergebnis = ergebnis + "   "
  ergebnisfeld.set_text(ergebnis)
```

In Kapitel 10 werden wir noch eine weitere Art der Schleife kennenlernen.

9 Listen

Im letzten Abschnitt (8.1) haben wir am Beispiel des Morsecodierers festgestellt, dass Objekte aus mehreren anderen Objekten bestehen können: Ein Text (ein *str*-Objekt) besteht aus mehreren Zeichen.

Listen sind im Prinzip das gleiche in allgemeiner: Eine Liste ist ein Objekt, das aus (beliebigen) anderen Objekten besteht – genau so, wie eine Liste, die wir auf ein Blatt Papier schreiben, aus mehreren Punkten besteht.

Anmerkung: Wenn du schon andere Programmiersprachen gelernt hast, hast du vielleicht schon einmal etwas von *Arrays* gehört. *Listen* in Python sind so etwas ähnliches[19].

Auch eine Liste ist in Python ein Objekt und kann ganz normal einer Variablen zugewiesen werden. Nehmen wir an, wir wollen ein Listen-Objekt mit dem Namen *meine_Liste* erstellen, dass die Zahlen 13, 37 und 42 enthält. Der Befehl dazu ist Folgender:

```
meine_Liste = [13, 37, 42]
```

Listen werden also durch eckige Klammern eingeschlossen, in denen die einzelnen Listen-Elemente durch Kommata getrennt aufgezählt werden. Listen können alle Arten von Objekten enthalten, also zum Beispiel auch Buchstaben, Texte oder sogar komplexe Objekte wie Fenster.

```
buchstaben_Liste = ["h", "a", "n"]
verrueckte_Liste = [42, "hallo welt", mein_Fenster]
```

Zugriff auf Listenelemente

Genau so, wie wir in Abschnitt 7.3 auf einen einzelnen Buchstaben in einem *str*-Objekt zugegriffen haben, können wir auch auf einzelne Elemente einer Liste zugreifen. Hierzu schreiben wir hinter den Namen der Liste eckige Klammern und darin die Nummer des Elements, das wir haben möchten. Denk daran, dass Informatiker – komische Leute – bei Null anfangen zu zählen (und nicht etwa bei eins).

[19] Arrays sind besondere Listen mit ganz bestimmten Eigenschaften: Bei einem Array müssen alle Elemente hintereinander im Speicher liegen und gleich groß sein (d.h. gleich viel Speicher belegen). Das hat technisch den großen Vorteil, dass der Computer sehr schnell auf einzelne Array-Elemente zugreifen kann. Der Nachteil ist, dass es nicht ohne weiteres möglich ist, die Größe eines Arrays zu ändern.

```
erstes_Element = meine_Liste[0]
zweites_Element = meine_Liste[1]
drittes_Element = meine_Liste[2]
```

In diesem Beispiel steht jetzt in der Variablen *erstes_Element* die Zahl 13, in *zweites_Element* die Zahl 37 und in *drittes_Element* die Zahl 42.

Erweitern von Listen

Wenn du eine Liste erstellst und später feststellst, dass da doch noch etwas mehr rein muss - kein Problem! Listen (die ja auch Objekte sind) haben die Funktion *append* (englisch: „anhängen“), mit der sich ein weiteres Element ans Ende der Liste hinzufügen lässt.

```
meine_Liste.append(1337)
```

In unserem Beispiel besteht die Liste *meine_Liste* jetzt aus den Elementen 13, 37, 42 und 1337.

Listen können auch leer sein! Hierzu schreiben wir beim Erstellen der Liste einfach nichts in die eckigen Klammern:

```
leere_Liste = []
```

Vielleicht erinnerst du dich daran, dass wir so ähnlich in Abschnitt 8.1 auch eine leere Zeichenkette erstellt haben (mit dem Befehl *ergebnis* = ””). Wie gesagt sind Listen so etwas ähnliches wie Zeichenketten – nur eben nicht nur für Zeichen, sondern allgemeiner.

Natürlich können wir auch eine leere Liste nachträglich erweitern:

```
leere_Liste.append(110)
leere_Liste.append(112)
```

In unserem Beispiel besteht die „*leere_Liste*“ (die ja gar nicht mehr leer ist) jetzt aus den Zahlen 110 und 112.

9.1 Beispiel Statistik-Tool

Als nächstes schreiben wir ein Programm, das statistische Daten auswerten soll. Der Benutzer soll beliebig viele Zahlen eingeben können (die wir natürlich in einer Liste speichern) und das Programm berechnet daraus das arithmetische Mittel (den „Durchschnitt“) und die Standardabweichung (s). Außerdem soll es anzeigen, wie viele Werte insgesamt eingegeben wurden (n).

Abbildung 19: Das Statistik-Tool

Für was braucht man das?

Bevor es los geht, ein kleiner Ausflug in die
Welt der Statistik. Das arithmetische Mittel hast du schon benutzt, wenn du den Notendurchschnitt von einer Arbeit ausgerechnet hast. Man addiert einfach alle Werte und teilt sie dann durch die Anzahl der Werte. Nochmal am Beispiel des Notendurchschnitts: Nehmen wir an, es gab in der letzten Arbeit zweimal die 1, dreimal die 2 und einmal die 4. Man summiert alle Noten (also $1 + 1 + 2 + 2 + 2 + 4 = 12$) und teilt das Ergebnis durch die Anzahl der Noten (also $12 / 6 = 2$). Der Durchschnitt ist 2!

Das arithmetische Mittel ($\overline{x}$) als Formel:

$$\overline{x} = \frac{x_1+x_2+...+x_n}{n} = \frac{1}{n} * \sum_{i=1}^{n} x_i$$

x_1, x_2 und so weiter bis x_n stehen dafür für den ersten, zweiten und so weiter bis zum letzten eingegebenen Wert. n steht für die Anzahl der eingegebenen Werte. Die Schreibweise rechts mit dem Summen-Sigma (Σ) kennt ihr wahrscheinlich nicht – sie ist aber nichts anderes als die mathematisch saubere Form dessen, was wir gerade besprochen haben.

Die Standardabweichung gibt an, wie gut die tatsächlichen Werte durch den Durchschnitt repräsentiert werden – also ob das arithmetische Mittel in diesem Fall überhaupt etwas sinnvolles aussagt. Beispiel: Nehmen wir an, in einer Arbeit schreibt die eine Hälfte der Klasse eine 1 und die andere Hälfte eine 6. Der Durchschnitt wäre 3,5, obwohl in Wirklichkeit niemand auch nur annähernd im 3er-Bereich gelandet ist. In diesem Fall würde eine hohe Standardabweichung darauf hinweisen, dass hier etwas nicht stimmt.

Die Berechnung der Standardabweichung (s) ist etwas komplizierter und benutzt das arithmetische Mittel: Für jeden einzelnen eingegebenen Wert rechnen wir zuerst das arithmetische Mittel minus diesen Wert und quadrieren, was dabei herauskommt. Diese Ergebnisse, die wir nun für jeden eingegebenen Wert haben, summieren wir und teilen sie durch die Anzahl der Werte minus 1. Hiervon ziehen wir schließlich noch die Wurzel – fertig ist die Standardabweichung!

Die Standardabweichung als Formel:

$$s = \sqrt{\frac{(\overline{x}-x_1)^2+(\overline{x}-x_2)^2+...+(\overline{x}-x_n)^2}{n-1}} = \sqrt{\frac{1}{n-1} * \sum_{i=1}^{n} (\overline{x} - x_i)^2}$$

Keine Panik!

Zur Beruhigung, falls du gerade entsetzt auf diese Formeln starrst und nur Bahnhof verstehst: Du musst die Mathematik dahinter jetzt auch gar nicht unbedingt verstehen. Sieh das einfach als zwei Zahlen, die Statistiker unheimlich toll finden. Wir als Informatiker haben jetzt einfach nur den Job, ihnen diese Zahlen zu liefern – auch, wenn wir damit selbst vielleicht nichts anfangen können.

Übrigens: Arithmetisches Mittel und Standardabweichung berechne ich auch, wenn ich die Evaluation ausgewerte, in der ihr am Ende des Kurses den Unterricht bewerten könnt. Ich könnte also der „Kunde“ für dieses Programm sein.

Wie programmieren wir das?

Wir beginnen wie immer mit einer grafischen Oberfläche. Wir brauchen ein Eingabefeld, in das der Benutzer nacheinander jeweils einen Wert eingeben kann, und einen OK-Button. Dann benötigen wir drei Label-Objekte zur Ausgabe – eines für das arithmetische Mittel, eines für die Standardabweichung, und eines für die Anzahl der eingegebenen Werte.

Um später die für die Berechnung der Standardabweichung Wurzeln ziehen zu können, werden wir die Klasse *math* benutzen (denn die kann jede Menge Mathe-Krams, so dass wir nicht das Rad neu erfinden müssen). Diese müssen wir genauso importieren, wie wir das mit *gtk* seit der ersten Stunde machen: *import math*

Um die Eingaben alle speichern zu können, definieren wir uns irgendwo im Hauptprogramm eine leere Liste (z.B. nach den *import*-Befehlen). In meinem Beispielprogramm heißt diese Liste *Liste_der_Eingaben*.

In der Funktion, die beim Klick auf den OK-Button aufgerufen wird, machen wir nun nacheinander folgende Dinge:

1. Neue Eingabe auslesen und abspeichern

   Ein Klick auf OK bedeutet, dass der Benutzer eine neue Zahl eingegeben hat. Wir lesen die Eingabe also aus dem Eingabefeld aus, wandeln sie in eine *float*-Zahl um und hängen sie an unsere *Liste_der_Eingaben* an.

2. Anzahl der Eingaben abfragen

   Wie viele Eingaben es insgesamt gibt, soll im Fenster angezeigt werden. Diese Anzahl zu bestimmen, geht ganz einfach: Wir lassen Python einfach für uns nachzählen, wie lang unsere Liste ist. Das geht mit *len(Liste_der_Eingaben)*. Das Ergebnis wandeln wir wie immer wieder in einen Text (*str*) um, um es ins passende Ergebnisfeld im Fenster schreiben zu können.

3. Artihmetisches Mittel berechnen

   Um das arithmetische Mittel berechnen zu können, müssen wir alle Eingaben in einer Schleife aufsummieren. Dazu definieren wir uns zuerst, vor der Schleife, eine Variable *Summe* und setzen sie gleich 0.

   Danach kommt eine *for*-Schleife über alle Eingabewerte, in der wir jeweils einen Wert zu dieser Summe dazu-addieren:

   ```
   Summe = 0
   for Zahl in Liste_der_Eingaben:
     Summe = Summe + Zahl
   ```

   Die so berechnete Summer teilen wir schließlich noch durch die Anzahl der Eingaben. Auch das arithmetische Mittel wandeln wir nach der Berechnung natürlich wieder in Text um und schreiben es ins Fenster.

4. Standardabweichung berechnen

Auch für die Berechnung der Standardabweichung müssen wir Dinge aufsummieren. Wir gehen also im Prinzip so vor, wie schon beim arithmetischen Mittel: Als erstes definieren wir uns eine Summen-Variable, in einer neuen *for*-Schleife über alle Eingabewerte addieren wir dann immer unsere Zwischenergebnisse auf diese Summen-Variable drauf.

In der Schleife rechnen wir zuerst arithmetisches Mittel minus dem jeweiligen Eingabewert. Das Ergebnis multiplizieren wir mit sich selbst (denn wir müssen es ja quadrieren) und addieren es, wie gesagt, auf die Summen-Variable drauf.

Nach der Schleife teilen wir unsere Summen-Variable durch die Anzahl der Eingaben und ziehen aus diesem Ergebnis noch die Wurzel. Das geht mit der Funktion *sqrt* aus der Klasse *math*.

```
Summe_fuer_s = 0
for Zahl in Liste_der_Eingaben:
  dieser_Summand = (Mittelwert - Zahl)
  dieser_Summand = dieser_Summand * dieser_Summand
  Summe_fuer_s = Summe_fuer_s + dieser_Summand

Standardabweichung = Summe_fuer_s / Anzahl_der_Eingaben
Standardabweichung = math.sqrt(Standardabweichung)
```

Das war's! Versuche, das Statistik-Tools mit diesen Hinweisen selbst zu schreiben. Meine Version findest du unter Programm 15. Wenn du noch etwas verwirrt sein solltest: Lege dir mein Programm direkt neben die Erklärungen oben. Vielleicht hilft das dabei, es nachzuvollziehen.

Programm 15: Statistik-Tool

```
1  Liste_der_Eingaben = []
2  # Zu Beginn des Programms ist die Liste leer.
3  # Bei jeder Eingabe eines neuen Wertes wird dieser
4  # an die Liste angehaengt.
5
6  def berechnen(parameter):
7    # Neu eingegebenen Wert auslesen, in Zahl umwandeln
8    # und in die Liste packen:
9    ausgelesen = Eingabefeld.get_text()
10   ausgelesen_Zahl = float(ausgelesen)
11   Liste_der_Eingaben.append(ausgelesen_Zahl)
12
13   # Die Liste nach der Anzahl der Eingaben fragen und
14   # die Antwort ins Fenster schreiben:
15   Anzahl_der_Eingaben = len(Liste_der_Eingaben)
16   Anzahl_der_Eingaben_Text = str(Anzahl_der_Eingaben)
17   Anzahl_Ausgabe.set_text(Anzahl_der_Eingaben_Text)
18
19   # Berechnen des arithmetischen Mittels:
20   # Erst alle Eingaben zusammenrechnen...
21   Summe = 0
```

```
22    for Zahl in Liste_der_Eingaben:
23      Summe = Summe + Zahl
24
25    # ... und dann durch die Anzahl teilen.
26    # Ergebnis kommt ins Fenster!
27    Mittelwert = Summe / Anzahl_der_Eingaben
28    Mittelwert_Text = str(Mittelwert)
29    Mittelwert_Ausgabe.set_text(Mittelwert_Text)
30
31    # Jetzt die Standardabweichung berechnen:
32    # Zunaechst fuer jede Eingabe "Mittel minus diese Eingabe"
33    # rechnen und Ergebnis summieren.
34    Summe_fuer_s = 0
35    for Zahl in Liste_der_Eingaben:
36      # Mittelwert minus der jeweiligen Eingabe rechnen,
37      # quadrieren und zusammenrechnen
38      dieser_Summand = (Mittelwert - Zahl)
39      dieser_Summand = dieser_Summand * dieser_Summand
40      Summe_fuer_s = Summe_fuer_s + dieser_Summand
41
42    # Diese Summe durch die Anzahl der Eingaben minus 1 teilen
43    # und aus dem Ergebnis die Wurzel ziehen.
44    # Fertig ist die Standardabweichung!
45    Standardabweichung = Summe_fuer_s / (Anzahl_der_Eingaben - 1)
46    Standardabweichung = math.sqrt(Standardabweichung)
47
48    Standardabweichung_Text = str(Standardabweichung)
49    Standardabweichung_Ausgabe.set_text(Standardabweichung_Text)
50
51  import gtk
52  import math
53
54  Fenster = gtk.Window()
55  Fenster.show()
56  Fenster.set_title("Statistik-Tool")
57
58  # Die grafische Oberflaeche soll wieder
59  # aus mehreren Zeilen bestehen:
60  untereinander = gtk.VBox()
61  untereinander.show()
62  Fenster.add(untereinander)
63
64  # Erste Zeile: Eingabe der Werte
65  Zeile1 = gtk.HBox()
66  Zeile1.show()
67  untereinander.add(Zeile1)
68
```

```
69 Eingabe_Label = gtk.Label("Neuen Wert eingeben: ")
70 Eingabe_Label.show()
71 Zeile1.add(Eingabe_Label)
72
73 Eingabefeld = gtk.Entry()
74 Eingabefeld.show()
75 Zeile1.add(Eingabefeld)
76
77 OK = gtk.Button("OK")
78 OK.show()
79 OK.connect("clicked", berechnen)
80 Zeile1.add(OK)
81
82 # Zweite bis letzte Zeile: Ausgabe der Ergebnisse
83 Zeile2 = gtk.HBox()
84 Zeile2.show()
85 untereinander.add(Zeile2)
86
87 Mittelwert_Label = gtk.Label("Arithmetisches Mittel: ")
88 Mittelwert_Label.show()
89 Zeile2.add(Mittelwert_Label)
90
91 Mittelwert_Ausgabe = gtk.Label("    ")
92 Mittelwert_Ausgabe.show()
93 Zeile2.add(Mittelwert_Ausgabe)
94
95 Zeile3 = gtk.HBox()
96 Zeile3.show()
97 untereinander.add(Zeile3)
98
99 Standardabweichung_Label = gtk.Label("Standardabweichung: ")
100 Standardabweichung_Label.show()
101 Zeile3.add(Standardabweichung_Label)
102
103 Standardabweichung_Ausgabe = gtk.Label("    ")
104 Standardabweichung_Ausgabe.show()
105 Zeile3.add(Standardabweichung_Ausgabe)
106
107 Zeile4 = gtk.HBox()
108 Zeile4.show()
109 untereinander.add(Zeile4)
110
111 Anzahl_Label = gtk.Label("Anzahl Werte: ")
112 Anzahl_Label.show()
113 Zeile4.add(Anzahl_Label)
114
115 Anzahl_Ausgabe = gtk.Label("    ")
```

```
116 Anzahl_Ausgabe.show()
117 Zeile4.add(Anzahl_Ausgabe)
118
119 gtk.main()
```

10 Die *while*-Schleife

Kommt noch! :-)